## Dylan Language Extensions

1 Dylan Language Extensions
1.1 Language differences .................................................. 3
1.2 Function Definition .................................................. 5
1.3 Extensions to the FOR iteration construct ......................... 6
1.4 Weak tables ............................................................ 7
1.5 Inlining adjectives for methods, constants, functions, and slots 7
1.6 Integers .................................................................... 8
1.7 object-with-elements ................................................... 26
1.8 Macro System Extensions ............................................. 26
1.9 Parser Expansions ....................................................... 29
1.10 Alternative Curry Syntax ............................................ 30

## LID File Format

2 LID File Format
2.1 LID files .................................................................... 33
2.2 LID keyword statements ............................................. 33
2.3 Open Dylan’s LID extensions ....................................... 35
2.4 File naming conventions ............................................. 38
2.5 Application example .................................................. 39

## The c-ffi Library

3 The c-ffi Library
3.1 Overview .................................................................... 42
3.2 Notes on Linking ......................................................... 47
3.3 Tracing FFI Calls ....................................................... 47
3.4 Terminology ................................................................ 48
3.5 Basic options in C-FFI macros .................................... 48
3.6 Designator classes ...................................................... 49
3.7 Defining types ............................................................ 65
3.8 Functions ................................................................. 74
3.9 Allocating and deallocating C storage ......................... 84
3.10 Utility designator classes .......................................... 86

## The collections Library

4 The collections Library
4.1 The bit-set Module ....................................................... 91
4.2 The bit-vector Module ................................................ 95
4.3 The collectors Module ................................................ 100
4.4 The plists Module ....................................................... 100
4.5 The set Module ........................................................ 104
4.6 The table-extensions Module .................................... 104

## The coloring-stream Library

5 The coloring-stream Library

### Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Dylan Language Extensions</td>
<td>3</td>
</tr>
<tr>
<td>1.1 Language differences</td>
<td>3</td>
</tr>
<tr>
<td>1.2 Function Definition</td>
<td>5</td>
</tr>
<tr>
<td>1.3 Extensions to the FOR iteration construct</td>
<td>6</td>
</tr>
<tr>
<td>1.4 Weak tables</td>
<td>7</td>
</tr>
<tr>
<td>1.5 Inlining adjectives for methods, constants, functions, and slots</td>
<td>7</td>
</tr>
<tr>
<td>1.6 Integers</td>
<td>8</td>
</tr>
<tr>
<td>1.7 object-with-elements</td>
<td>26</td>
</tr>
<tr>
<td>1.8 Macro System Extensions</td>
<td>26</td>
</tr>
<tr>
<td>1.9 Parser Expansions</td>
<td>29</td>
</tr>
<tr>
<td>1.10 Alternative Curry Syntax</td>
<td>30</td>
</tr>
<tr>
<td>2 LID File Format</td>
<td>33</td>
</tr>
<tr>
<td>2.1 LID files</td>
<td>33</td>
</tr>
<tr>
<td>2.2 LID keyword statements</td>
<td>33</td>
</tr>
<tr>
<td>2.3 Open Dylan’s LID extensions</td>
<td>35</td>
</tr>
<tr>
<td>2.4 File naming conventions</td>
<td>38</td>
</tr>
<tr>
<td>2.5 Application example</td>
<td>39</td>
</tr>
<tr>
<td>3 The c-ffi Library</td>
<td>41</td>
</tr>
<tr>
<td>3.1 Overview</td>
<td>42</td>
</tr>
<tr>
<td>3.2 Notes on Linking</td>
<td>47</td>
</tr>
<tr>
<td>3.3 Tracing FFI Calls</td>
<td>47</td>
</tr>
<tr>
<td>3.4 Terminology</td>
<td>48</td>
</tr>
<tr>
<td>3.5 Basic options in C-FFI macros</td>
<td>48</td>
</tr>
<tr>
<td>3.6 Designator classes</td>
<td>49</td>
</tr>
<tr>
<td>3.7 Defining types</td>
<td>65</td>
</tr>
<tr>
<td>3.8 Functions</td>
<td>74</td>
</tr>
<tr>
<td>3.9 Allocating and deallocating C storage</td>
<td>84</td>
</tr>
<tr>
<td>3.10 Utility designator classes</td>
<td>86</td>
</tr>
<tr>
<td>4 The collections Library</td>
<td>91</td>
</tr>
<tr>
<td>4.1 The bit-set Module</td>
<td>91</td>
</tr>
<tr>
<td>4.2 The bit-vector Module</td>
<td>95</td>
</tr>
<tr>
<td>4.3 The collectors Module</td>
<td>100</td>
</tr>
<tr>
<td>4.4 The plists Module</td>
<td>100</td>
</tr>
<tr>
<td>4.5 The set Module</td>
<td>104</td>
</tr>
<tr>
<td>4.6 The table-extensions Module</td>
<td>104</td>
</tr>
<tr>
<td>5 The coloring-stream Library</td>
<td>111</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
</tr>
<tr>
<td>---------</td>
<td>--------------------------------------------</td>
</tr>
<tr>
<td>5.1</td>
<td>Usage</td>
</tr>
<tr>
<td>5.2</td>
<td>The coloring-stream Module</td>
</tr>
<tr>
<td>6</td>
<td>The command-line-parser Library</td>
</tr>
<tr>
<td>6.1</td>
<td>Quick Start</td>
</tr>
<tr>
<td>6.2</td>
<td>Reference</td>
</tr>
<tr>
<td>7</td>
<td>The common-dylan Library</td>
</tr>
<tr>
<td>7.1</td>
<td>The byte-vector Module</td>
</tr>
<tr>
<td>7.2</td>
<td>The common-extensions Module</td>
</tr>
<tr>
<td>7.3</td>
<td>The machine-words Module</td>
</tr>
<tr>
<td>7.4</td>
<td>The simple-format Module</td>
</tr>
<tr>
<td>7.5</td>
<td>The simple-profiling Module</td>
</tr>
<tr>
<td>7.6</td>
<td>The simple-random Module</td>
</tr>
<tr>
<td>7.7</td>
<td>The simple-timers Module</td>
</tr>
<tr>
<td>7.8</td>
<td>The transcendentals Module</td>
</tr>
<tr>
<td>8</td>
<td>The dylan Library</td>
</tr>
<tr>
<td>8.1</td>
<td>The finalization Module</td>
</tr>
<tr>
<td>8.2</td>
<td>The dylan-primitives Module</td>
</tr>
<tr>
<td>8.3</td>
<td>The threads Module</td>
</tr>
<tr>
<td>9</td>
<td>The hash-algorithms Library</td>
</tr>
<tr>
<td>9.1</td>
<td>The hash-algorithms Module</td>
</tr>
<tr>
<td>10</td>
<td>The io Library</td>
</tr>
<tr>
<td>10.1</td>
<td>The format Module</td>
</tr>
<tr>
<td>10.2</td>
<td>The format-out Module</td>
</tr>
<tr>
<td>10.3</td>
<td>The print and pprint Modules</td>
</tr>
<tr>
<td>10.4</td>
<td>The streams Module</td>
</tr>
<tr>
<td>10.5</td>
<td>The standard-io Module</td>
</tr>
<tr>
<td>11</td>
<td>The logging Library</td>
</tr>
<tr>
<td>11.1</td>
<td>Overview</td>
</tr>
<tr>
<td>11.2</td>
<td>Quick Start</td>
</tr>
<tr>
<td>11.3</td>
<td>Errors</td>
</tr>
<tr>
<td>11.4</td>
<td>Log Levels</td>
</tr>
<tr>
<td>11.5</td>
<td>Logging Functions</td>
</tr>
<tr>
<td>11.6</td>
<td>Logs</td>
</tr>
<tr>
<td>11.7</td>
<td>Log Targets</td>
</tr>
<tr>
<td>11.8</td>
<td>Log Formatting</td>
</tr>
<tr>
<td>12</td>
<td>The network Library</td>
</tr>
<tr>
<td>12.1</td>
<td>Overview</td>
</tr>
<tr>
<td>12.2</td>
<td>Utilities</td>
</tr>
<tr>
<td>12.3</td>
<td>Internet addresses</td>
</tr>
<tr>
<td>12.4</td>
<td>Sockets</td>
</tr>
<tr>
<td>12.5</td>
<td>Socket conditions</td>
</tr>
<tr>
<td>13</td>
<td>The progress-stream Library</td>
</tr>
<tr>
<td>14</td>
<td>The regular-expressions Library</td>
</tr>
<tr>
<td>14.1</td>
<td>Overview</td>
</tr>
<tr>
<td>14.2</td>
<td>Quick Start</td>
</tr>
<tr>
<td>14.3</td>
<td>Reference</td>
</tr>
</tbody>
</table>
This reference describes the libraries provided with Open Dylan, either as part of the “opendylan” repository or included therein as submodules.

It also documents Dylan language extensions and the LID file format.
The Dylan language is described in The Dylan Reference Manual by Andrew Shalit (Addison-Wesley, 1996). We call this book “the DRM” hereafter.

Open Dylan provides an implementation of the Dylan language described by the DRM, with a few exceptions that are documented in Language differences.

Open Dylan provides the Dylan language in the dylan module of the dylan library.

This document is an introduction to Open Dylan’s extensions to the Dylan language. These extensions are either built in to the dylan library or are available in a separate library, common-dylan.

The majority of the extensions are in the common-extensions module of the common-dylan library. That library also exports a number of smaller modules that contain other basic facilities such as simplified formatting, pseudo-random integer generation, and object finalization.

Open Dylan provides a convenience library, common-dylan, that combines the dylan and common-extensions modules to provide a convenient “dialect” of Dylan, exported from the module common-dylan:

```dylan
define library common-dylan
  use dylan, export: all;
  use common-extensions, export: all;
  export common-dylan;
end module;

define module common-dylan
  use dylan, export: all;
  use common-extensions, export: all;
end module;
```

This section describes the common language extensions, that is, extensions made to the Dylan library as it is defined in DRM. These extensions are available to applications in the dylan library's dylan module.

### 1.1 Language differences

Note that in addition to the intentional differences documented below, there are several known bugs related to compliance with the language standard as documented in the DRM.
1.1.1 Table Protocol

For efficiency, Open Dylan adopts a slightly different table protocol to that described by the DRM. Hashing functions take an additional hash-state argument and merge it into the hash-state result. The function `merge-hash-codes` is replaced by `merge-hash-ids` because hash-states are merged as part of the hashing process. The constant `$permanent-hash-state` is no longer required; the same effect can be achieved by returning the argument `hash-state` unchanged as the result `hash-state`. Finally, `object-hash` has been altered to use the new protocol.

This section describes the items that have been changed. We also provide a Table-extensions module, which you can read about in *The table-extensions Module*.

table-protocol Open Generic function

Returns functions used to implement the iteration protocol for tables.

**Signature**

\[
\text{table-protocol} \quad \text{table} \Rightarrow \text{test-function hash-function}
\]

**Parameters**

- \textit{table} – An instance of `<table>`.

**Values**

- \textit{test-function} – An instance of `<function>`.
- \textit{hash-function} – An instance of `<function>`.

**Discussion**

Returns the functions used to iterate over tables. These functions are in turn used to implement the other collection operations on `<table>`.

The `test-function` argument is for the table test function, which is used to compare table keys. It returns true if, according to the table’s equivalence predicate, the keys are members of the same equivalence class. Its signature must be:

\[
\text{test-function} \; \text{*key1* \; *key2*} \Rightarrow \text{*boolean*}
\]

The `hash-function` argument is for the table hash function, which computes the hash code of a key. Its signature must be:

\[
\text{hash-function} \; \text{*key* \; *initial-state*} \Rightarrow \text{*id* \; *result-state*}
\]

In this signature, `initial-state` is an instance of `<hash-state>`. The hash function computes the hash code of `key`, using the hash function that is associated with the table’s equivalence predicate. The hash code is returned as two values: an integer `id` and a hash-state `result-state`. This `result-state` is obtained by merging the `initial-state` with the hash-state that results from hashing `key`. The `result-state` may or may not be `==` to `initial-state`. The `initial-state` could be modified by this operation.

merge-hash-ids Function

Returns a hash ID created by merging two hash IDs.

**Signature**

\[
\text{merge-hash-ids} \; \text{id1 \; id2 \; #key \; ordered} \Rightarrow \text{merged-id}
\]

**Parameters**

- \textit{id1} – An instance of `<integer>`.
- \textit{id2} – An instance of `<integer>`.
- \textit{ordered} – An instance of `<boolean>`. Default value: `#f`.

**Values**
• merged-id – An instance of `<integer>`.

Discussion

Computes a new hash ID by merging the argument hash IDs in some implementation-dependent way. This can be used, for example, to generate a hash ID for an object by combining hash IDs of some of its parts.

The `id1, id2` arguments and the return value `merged-id` are all integers.

The `ordered` argument is a boolean, and determines whether the algorithm used to the merge the IDs is permitted to be order-dependent. If false (the default), the merged result must be independent of the order in which the arguments are provided. If true, the order of the arguments matters because the algorithm used need not be either commutative or associative. It is best to provide a true value for `ordered` when possible, as this may result in a better distribution of hash IDs. However, `ordered` must only be true if that will not cause the hash function to violate the second constraint on hash functions, described on page 123 of the DRM.

object-hash Function

The hash function for the equivalence predicate `==`.

Signature

```
object-hash object initial-state => hash-id result-state
```

Parameters

• `object` – An instance of `<integer>`.
• `initial-state` – An instance of `<hash-state>`.

Values

• `hash-id` – An instance of `<integer>`.
• `result-state` – An instance of `<hash-state>`.

Discussion

Returns a hash code for `object` that corresponds to the equivalence predicate `==`.

This function is a useful tool for writing hash functions in which the object identity of some component of a key is to be used in computing the hash code.

It returns a hash ID (an integer) and the result of merging the initial state with the associated hash state for the object, computed in some implementation-dependent manner.

1.2 Function Definition

The `define function` definition macro provides a convenient way to define functions that have no generic properties and hence are not suitable for definition with `define generic` or `define method`. This extension has been accepted as part of the language since the DRM was published.

The `define function` macro provides a way of defining a function that says clearly to other programmers that the function is not part of any generic operation; furthermore, the function will not be extended as a generic function, and calling it need not involve any generic dispatch. Without this macro, programmers who wanted to do so would have to turn to `define constant`. With `define function`, programmer intent is more explicit and it relays more information to future maintainers of a piece of code.

The language definition of `define function` explicitly does not specify what it expands into, so that Dylan implementations have latitude to support this definer in the best way suited to the implementation.

**define function Defining Macro**

Defines a constant binding in the current module and initializes it to a new function.
Macro Call

```dylan
define (*adjective* )* function *name* *parameter-list*
[ *body* ]
end [ function ] [ *name* ]
```

Parameters

- **adjective** – A Dylan unreserved-name bnf.
- **name** – A Dylan variable-name bnf.
- **parameter-list** – A Dylan parameter-list bnf.
- **body** – A Dylan body bnf.

Discussion

Creates a constant module binding with the name `name`, and initializes it to a new function
described by `parameter-list`, `options`, and any adjectives.

The adjectives permitted depend on the implementation.

The `parameter-list` describes the number and types of the function’s arguments and return values.
It is an error to supply `#next` in the parameter list, and there is no implicit `#next` parameter.

Operations

The following functions return the same values as they would if the function had been
defined as a bare method with the same signature:

- `function-specializers`
- `function-arguments`
- `function-return-values`

Calling some of the following reflective operations on a function defined with `define`
function may be an error:

- `generic-function-methods`
- `add-method`
- `generic-function-mandatory-keywords`
- `sorted-applicable-methods`
- `find-method`
- `remove-method`
- `applicable-method?`

### 1.3 Extensions to the FOR iteration construct

We have also made two extensions to the `for` iteration construct: a `keyed-by` clause and `in ... using` clauses.

The `keyed-by` clause allows iteration over table elements:

```dylan
for (my-element keyed-by my-key in my-table)
...
end;
```

The `in ... using` clause allows you to specify a iteration protocol other than the default
(`forward-iteration-protocol`):
\texttt{for (element in my-sequence using backward-iteration-protocol)}
\texttt{\ldots}
\texttt{end;}

\section*{1.4 Weak tables}

We have extended \texttt{define table} to incorporate \textit{weak references} through keys and values.

A weak reference is an reference that the garbage collector treats as irrelevant to establishing whether the object referred to is live. If an object has only weak references to it, the garbage collector can delete the reference and recycle the object’s memory. We call a normal reference a \textit{strong reference}.

Weak references are a useful tool for building data structures where you do not want the garbage collector to preserve objects in the structure on account of certain references merely used to build up the structure.

Typically, this level of control is not required in a language like Dylan, which does not expose memory references to programs. But without the ability to tell the garbage collector to disregard certain kinds of reference, data structures such as tables could be bloated unnecessarily by the garbage collector preserving entries (a key/value pair) solely because the table object itself has a reference to the entry’s key or value.

Open Dylan provides weakness options for instances of \texttt{<table>}. A table can have \textit{weak keys} or \textit{weak values}:

\begin{verbatim}
make(<table>, weak: "key");  // makes a weak-key table
make(<table>, weak: "value"); // makes a weak-value table
\end{verbatim}

In a weak-keyed table, if a key is no longer referenced from anywhere else in the program (apart from weak references, including from the same table), then the entry (key and value) can be deleted from the table. After that, the key object will be recycled. The value will also be recycled unless it has strong references from elsewhere in the program.

Weak-valued tables are much the same, except that the focus is values and not keys. In a weak-valued table, if a value is no longer referenced from anywhere else in the program (apart from weak references, including from the same table), then the entry (value and key) can be deleted from the table. After that, the value object will be recycled. The key will also be recycled unless it has strong references from elsewhere in the program.

Weak tables are useful for implementing many sorts of cache, where the cached data is recomputable and yet both expensive to compute and also expensive to keep for a long time. For example, consider something like a font cache for an X Window System server, or a printer. Fonts might be looked up by name, so the strings would be the keys of the table. The values would be the bitmaps for the font. While the X server is using a font, the cache will be kept alive — so any further requests to select the font will find the data already present. However, if the font is not used then you would eventually expect the garbage collector to clean it out. Any future request would then have to re-load all the bitmaps.

\section*{1.5 Inlining adjectives for methods, constants, functions, and slots}

To \texttt{inline} a value is to replace, at compile time, a reference to a variable with the value of that variable. Such inlining often allows compile-time evaluation (“constant folding”) or partial evaluation.

The Open Dylan compiler can perform inlining on generic function methods, constants, class slots, and functions (created with \texttt{define function} — see \textit{Function Definition}). We have extended the Dylan language specification of \texttt{define method}, \texttt{define constant}, and class slots with inlining definition adjectives and have included those same adjectives in our language extension \texttt{define function}. The adjectives are:

- \texttt{not-inline} Never inline this item.
• **default-inline** (default) Inline this item within a library, at the compiler’s discretion. Never inline a cross-library reference.

• **may-inline** Inline this item within or between libraries, at the compiler’s discretion.

• **inline** Inline this item wherever the compiler can do so.

In addition, define constant and define function permit the adjective **inline-only**, which forces every reference to the constant or function to be inlined.

**Note:** If you export from a library any variables created with may-inline, inline, or inline-only, and then change the values of the variables, client libraries may need to be recompiled.

### 1.6 Integers

This document describes the Open Dylan implementation of arithmetic functions, especially integer arithmetic. It describes a number of extensions to the Dylan language, which are available from the Dylan library. It also describes a generic arithmetic facility that, through the use of other libraries, allows you to extend arithmetic to special number types, such as “big” (64-bit) integers.

Throughout this document, arguments are instances of the class specified by the argument name (ignoring any numeric suffixes), unless otherwise noted. Thus, the arguments *integer*, *integer1*, and *integer2* would all be instances of the class `<integer>`.

The goals of the extensions to the Dylan language described in this document are as follows:

• Provide arithmetic operations that are closed over small integers.
  
  This allows type inference to propagate small integer declarations more widely, because there is no possibility of automatic coercion into some more general format.

• Make the arithmetic operations that are closed over small integers easily accessible to programmers.

• Allow the Dylan library to be described in such a way that only small integers are present by default, moving support for infinite precision integer arithmetic to the Big-Integers library, which must be explicitly used.

• Support infinite precision integer arithmetic through the Big-Integers library.

**Note:** Using that library in another library does not have a negative effect on the correctness or performance of other libraries in the same application that do not use it.

• Maintain compatibility with the DRM specification.

  In particular, the extensions support the production of efficient code for programs written to be portable with respect to the DRM specification. Use of implementation-specific types or operations in order to get reasonable efficiency is not required. This precludes relegating the `<integer>` class and `limited-<integer>` types to inefficient implementations.

**Note:** When there are several distinct interfaces with the same name but in different modules, the notation `interface # module` is used in this document to remove ambiguity.

• Specify that the class `<integer>` has a finite, implementation-dependent range, bounded by the constants `$minimum-integer` and `$maximum-integer`. 
The representation for integers must be at least 28 bits, including the sign. That is, the minimum conforming value for $\text{maximum-integer}$ is $2^{27} - 1$ and the maximum conforming value for $\text{minimum-integer}$ is $-2^{27}$.

**Note:** **Rationale:** Restricting `<integer>` in this way allows the programmer to stay in the efficient range without requiring exact knowledge of what that range might be. The full generality of extended precision integers is provided by the Big-Integers library, for programmers who actually need that functionality.

- Define the type `<machine-number>` to be the type union of `<float>` and `<integer>`.

The Dylan library provides implementations of the generic functions and functions described in this document. If the result of one of these operations is specified to be an instance of `<integer>` and the mathematically correct result cannot be represented as an `<integer>` then an error is signaled. This removes fully generic arithmetic from the Dylan library. In particular, it removes extended integers, ratios, and rectangular complex numbers.

### 1.6.1 Extensions to the Dylan Library

This section describes the extensions to the Dylan library that provide the arithmetic operations available as standard to your applications. You do not have to explicitly use any additional libraries to have access to any of the functionality described in this section. Note that this section only describes extensions to the Dylan library; for complete descriptions, you should also refer to the *Dylan Reference Manual*.

Note that the Common-Dylan library also has these extensions because it uses the Dylan library.

**Ranges**

The initialization arguments for `<range>` must all be instances of `<machine-number>` rather than `<real>`.

**Specific constructors**

The following specific constructors are available for use with the class `<integer>`.

#### limited Generic function

Defines a new type that represents a subset of the class `<integer>`.

**Signature**  
limited integer-class #key min max => limited-type

**Parameters**

- **integer-class** – The singleton(<integer>).
- **min** – The lower bound of the range. The default is $\text{minimum-integer}$.
- **max** – The upper bound of the range. The default is $\text{maximum-integer}$.

**Discussion** The `integer-class` argument is the class `<integer>`, and all other arguments are instances of `<integer>`. The range of `<integer>` is bounded by default.

#### range Function

This function is used to specify ranges of numbers.

**Signature**  
range (#key from:, to:, above:, below:, by:, size:) => <range>

**Discussion** All of the supplied arguments must be instances of `<machine-number>`.
Equality comparisons

The = function compares two objects and returns #t if the values of the two objects are equal to each other, that is of the same magnitude.

= Open Generic function
Tests its arguments to see if they are of the same magnitude.

Signature = object1 object2 => boolean

=(<complex>) Sealed Method
Tests its arguments to see if they are of the same magnitude.

Signature = complex1 complex2 => boolean

=(<machine-number>) Method
Tests its arguments to see if they are of the same magnitude.

Signature = machine-number1 machine-number2 => boolean

Magnitude comparisons

The Dylan library provides the following interfaces for testing the magnitude of two numbers:

< Open Generic function
Returns #t if its first argument is less than its second argument.

Signature < object1 object2 => boolean

<(complex>) Sealed Method
Returns #t if its first argument is less than its second argument.

Signature < complex1 complex2 => boolean

<(machine-number>) Method
Returns #t if its first argument is less than its second argument.

Signature < machine-number1 machine-number2 => boolean

Properties of numbers

Various number properties can be tested using the following predicates in the Dylan library:

odd? Open Generic function
Tests whether the argument supplied represents an odd value.

Signature odd? object => boolean

odd?(<complex>) Sealed Method
Tests whether the argument supplied represents an odd value.

Signature odd? complex => boolean

odd?(<integer>) Method
Tests whether the argument supplied represents an odd value.

Signature odd? integer => boolean

even? Open Generic function
Tests whether the argument supplied represents an even value

Signature even? object => boolean
even?(<complex>) Sealed Method
Tests whether the argument supplied represents an even value

Signature even? complex => boolean

even?(<integer>) Method
Tests whether the argument supplied represents an even value

Signature even? integer => boolean

zero? Open Generic function
Tests whether the argument supplied represents a zero value.

Signature zero? object => boolean

zero?(<complex>) Sealed Method
Tests whether the argument supplied represents a zero value.

Signature zero? complex => boolean

zero?(<machine-number>) Method
Tests whether the argument supplied represents a zero value.

Signature zero? machine-number => boolean

positive? Open Generic function
Tests whether the argument supplied represents a positive value.

positive?(<complex>) Sealed Method
Tests whether the argument supplied represents a positive value.

Signature positive? complex

positive?(<machine-number>) Method
Tests whether the argument supplied represents a positive value.

negative? Open Generic function
Tests whether the argument supplied represents a negative value.

negative?(<complex>) Sealed Method
Tests whether the argument supplied represents a negative value.

Signature negative? complex => boolean

negative?(<machine-number>) Method
Tests whether the argument supplied represents a negative value.

Signature negative? machine-number => boolean

integral? Open Generic function
Tests whether the argument supplied represents an integral value.

integral?(<complex>) Sealed Method
Tests whether the argument supplied represents an integral value.

Signature integral? complex

integral?(<machine-number>) Method
Tests whether the argument supplied represents an integral value.

Signature integral? machine-number => boolean

1.6. Integers
Arithmetic operations

The following arithmetic operations are available in the Dylan library:

+ 
Open generic function
+ object1 object2 => #rest object
+
Sealed domain
+ complex1 complex 2
+
G.f. method
+ integer1 integer2 => integer
+
G.f. method
+ machine-number1 machine-number2 => machine-number

Returns the sum of the two supplied arguments. The actual type of the value is determined by the contagion rules when applied to the arguments.

- 
Open generic function
- object1 object2 => #rest object
-
Sealed domain
- complex1 complex2
-
G.f. method
- integer1 integer2 => integer
-
G.f. method
- machine-number1 machine-number2 => machine-number

Returns the result of subtracting the second argument from the first. The actual type of the value is determined by the contagion rules when applied to the arguments.

* 
Open generic function
* object1 object2 => #rest object
*
Sealed domain
* complex1 complex2
* G.f. method
* integer1 integer 2 => integer
*
* G.f. method
* machine-number1 machine-number2 => machine-number

Returns the result of multiplying the two arguments. The actual type of the value is determined by the contagion rules when applied to the arguments.

//
Open generic function
// object1 object2 => #rest object
//
Sealed domain
// complex1 complex2
//
* G.f. method
// float1 float 2 => float

Returns the result of dividing the first argument by the second. The actual type of the value is determined by the contagion rules when applied to the arguments.

negative
Open generic function
negative object => #rest negative-object
negative
Sealed domain
negative complex
negative
negative
G.f. method
negative integer => negative-integer
negative
G.f. method
negative float => negative-float

Negates the supplied argument. The returned value is of the same float format as the supplied argument.

floor
Function

floor machine-number => integer machine-number  floor integer => integer integer floor float => integer float

1.6. Integers
Truncates a number toward negative infinity. The integer part is returned as `integer`, the remainder is of the same float format as the argument.

`ceiling` Function

`ceiling machine-number => integer machine-number`  `ceiling integer => integer integer`  `ceiling float => integer float`

Truncates a number toward positive infinity. The integer part is returned as `integer`, the remainder is of the same float format as the argument.

`round` Function

`round machine-number => integer machine-number`  `round integer => integer integer`  `round float => integer float`

Rounds a number toward the nearest mathematical integer. The integer part is returned as `integer`, the remainder is of the same float format as the argument. If the argument is exactly between two integers, then the result `integer` will be a multiple of two.

`truncate` Function

`truncate machine-number => integer machine-number`  `truncate integer => integer integer`  `truncate float => integer float`

Truncates a number toward zero. The integer part is returned as `integer`, the remainder is of the same float format as the argument.

`floor/` Function

```
floor/ *machine-number1* *machine-number2* => *integer* *machine-number*
floor/ *integer1* *integer2* => *integer* *integer*
floor/ *machine-number1* *machine-number2* => *integer* *machine-number*
```

Divides the first argument into the second and truncates the result toward negative infinity. The integer part is returned as `integer`, the type of the remainder is determined by the contagion rules when applied to the arguments.

`ceiling/` Function

```
ceiling/ *machine-number1* *machine-number2* => *integer* *machine-number*
ceiling/ *integer1* *integer2* => *integer* *integer*
ceiling/ *machine-number1* *machine-number2* => *integer* *machine-number*
```

Divides the first argument into the second and truncates the result toward positive infinity. The integer part is returned as `integer`, the type of the remainder is determined by the contagion rules when applied to the arguments.

`round/` Function

```
round/ *machine-number1* *machine-number2* => *integer* *machine-number*
round/ *integer1* *integer2* => *integer* *integer*
round/ *machine-number1* *machine-number2* => *integer* *machine-number*
```
Divides the first argument into the second and rounds the result toward the nearest mathematical integer. The integer part is returned as `integer`, the type of the remainder is determined by the contagion rules when applied to the arguments.

```
truncate/ *machine-number1* *machine-number2* => *integer* *machine-number*
truncate/ *integer1* *integer2* 2 => *integer* *integer*
truncate/ *machine-number1* *machine-number2* => *integer* *machine-number*
```

Divides the first argument into the second and truncates the result toward zero. The integer part is returned as `integer`, the type of the remainder is determined by the contagion rules when applied to the arguments.

```
modulo **machine-number1** **machine-number2** => **machine-number**
modulo **integer1** **integer2** => **integer**
modulo **machine-number1** **machine-number2** => **machine-number**
```

Returns the second value of `floor/ (arg1, arg2)`. The actual type of the second value is determined by the contagion rules when applied to the arguments.

```
remainder **machine-number1** **machine-number2** => **machine-number**
remainder **integer1** **integer2** => **integer**
remainder **machine-number1** **machine-number2** => **machine-number**
```

Returns the second value of `truncate/ (arg1, arg2)`. The actual type of the second value is determined by the contagion rules when applied to the arguments.

```
^ **object1** **object2** => #rest **object**
```

Sealed domain

```
^ **complex1** **complex2**
```

G.f. method

```
^ **integer1** **integer2** => **integer**
```

G.f. method

```
^ **float1** **integer2** => **float**
```

Returns the first argument raised to the power of the second argument. The value is of the same float format as the first argument. An error is signalled if both arguments are 0.

```
abs
```

---

1.6. Integers

15
Open generic function
abs object => #rest object
abs
Sealed domain
abs complex
abs
G.f. method
abs integer => integer
abs
G.f. method
abs float => float
Returns the absolute value of the argument. The value is of the same float format as the argument.
logior
Function
logior #rest integers => integer
Returns the bitwise inclusive OR of its integer arguments.
logxor
Function
logxor #rest integers => integer
Returns the bitwise exclusive OR of its integer arguments.
logand
Function
logand #rest integers => integer
Returns the bitwise AND of its integer arguments.
lognot
Function
lognot integer1 => integer2
Returns the bitwise NOT of its integer arguments.
logbit?
Function
logbit? index integer => boolean
Tests the value of a particular bit in its integer argument. The index argument is an instance of <integer>.
ash
Function
ash integer1 count => integer
Performs an arithmetic shift on its first argument.
lcm
Function
lcm integer1 integer2 => integer
Returns the least common multiple of its two arguments.
gcd
Function
gcd integer1 integer2 => integer
Returns the greatest common divisor of its two arguments.

Collections

The keys for sequences are always instances of `<integer>`. This means that certain kinds of collections cannot be sequences; very large (or unbounded) sparse arrays are an example.

The Table Protocol

The following functions in the Dylan library are extended. Note that the hash IDs for tables are always instances of `<integer>`.

merge-hash-codes
Function

merge-hash-codes id1 state1 id2 state2 #key ordered? => merged-id merged-state
Returns a hash code created from the merging of two argument hash codes. The id arguments are hash IDs, and the state arguments are hash states (instances of `<object>`). The ordered? argument is an instance of `<boolean>`. The returned merged values are instances of `<integer>` and `<object>`, as determined by the name of each argument.

object-hash
Function

object-hash object => hash-id hash-state
The hash function for the equivalence predicate `=`. The return values are of the same types as the return values of merge-hash-codes.

Iteration Constructs

for
Statement macro

The start, bound, and increment expressions in a numeric clause must evaluate to instances of `<machine-number>` for this macro.

1.6.2 The generic-arithmetic Library

The Generic-Arithmetic library exports the functions described in this section from an exported module called `generic-arithmetic`.
The Generic-Arithmetic library provides a fully extensible version of all arithmetic operations. If an application only uses Generic-Arithmetic, these versions of the operators reduce themselves to be equivalent to those in the Dylan library. But when you use additional implementation libraries, the arithmetic operators are extended.

The Big-Integers library is one such implementation library. It provides a 64-bit implementation of <integer>.

The standard integer implementation in the Dylan library is actually part of the following class hierarchy:
<abstract-integer>
<integer>
<big-integer>
<double-integer>

(The classes <big-integer> and <double-integer> are implementation classes. You do not need to use them.)

The modules in the Generic-Arithmetic library export <abstract-integer> with the name <integer>. They also export a full set of arithmetic operators that use instances of <abstract-integer> rather than instances of <integer> (in the Dylan library naming scheme). However, those operators just fall back to the Dylan library operators until you include an implementation library, such as Big-Integers, in your application.

When you use the Big-Integers library, the arithmetic operators exported by Generic-Arithmetic are enhanced to extend their results to 64-bit integers. If a result is small enough to fit in a Dylan library <integer>, it will be fitted into one.

Note that the Generic-Arithmetic library uses the same naming conventions for arithmetic operators as used by the Dylan library. This means that some renaming is required in modules that require access to both the basic Dylan interfaces and the interfaces supplied by the Generic-Arithmetic library. As described earlier, the notation interface # module is used to denote different interfaces of the same name, where interface is the name of the interface, and module is the name of the module it is exported from.

See Using special arithmetic features for an example of how to use an implementation library with Generic-Arithmetic.

Ranges

The Generic-Arithmetic library defines the class <range>, which is in most respects functionally equivalent to <range>#Dylan, but uses generic arithmetic operations in its implementation so that the initialization arguments can be instances of <real>, rather than being restricted to <machine-number>.

Classes

The class <abstract-integer> is imported and re-exported under the name <integer>#generic-arithmetic.

Specific constructors

range

Function

range #key from to above below by size => range

This function is identical to the function range#Dylan, except that all of the supplied arguments must be instances of <real>. 18 Chapter 1. Dylan Language Extensions
Arithmetic operations

The following functions all apply function #Dylan to the arguments and return the results, where function is the appropriate function name. See Arithmetic operations for descriptions of each function as implemented in the Dylan library.

- object1 object2 => #rest object
- object1 object2 => #rest object
- * object1 object2 => #rest object
- / object1 object2 => #rest object
- negative object => #rest negative-object
- floor real1 => abstract-integer real
- ceiling real1 => abstract-integer real
- round real1 => abstract-integer real
- truncate real1 => abstract-integer real
- floor/ real1 real2 => abstract-integer real
- ceiling/ real1 real2 => abstract-integer real
- round/ real1 real2 => abstract-integer real
- truncate/ real1 real2 => abstract-integer real
- modulo real1 real2 => real
- remainder real1 real2 => real
- ^ object1 object2 => #rest object
- abs object1 => #rest object
- logior #rest abstract-integer1 => abstract-integer
- logxor #rest abstract-integer1 => abstract-integer
- logand #rest abstract-integer1 => abstract-integer
- lognot abstract-integer1 => abstract-integer
- logbit? integer abstract-integer => boolean
- ash abstract-integer1 integer => abstract-integer
- lcm abstract-integer1 abstract-integer2 => abstract-integer
- gcd abstract-integer1 abstract-integer2 => abstract-integer

Iteration constructs

While a programmer could make use of generic arithmetic in a for loop by using explicit-step clauses, this approach leads to a loss of clarity. The definition of the for macro is complex, so a version that uses generic arithmetic in numeric clauses is provided, rather than requiring programmers who want that feature to reconstruct it.

for
Statement macro
The start, bound, and increment expressions in a numeric clause must evaluate to instances of `<machine-number>` for this macro. Otherwise, this macro is similar to `for#Dylan`.

Exported Modules from the generic-arithmetic Library

The Generic-Arithmetic library exports several modules that are provided for the convenience of programmers who wish to create additional modules based on the `dylan` module plus various combinations of the arithmetic models.

The `dylan-excluding-arithmetic` Module

The Dylan-Excluding-Arithmetic module imports and re-exports all of the interfaces exported by the `dylan` module from the Dylan library, except for the following excluded interfaces:

- `<integer>`
- `range`
- `+ * /`
- `negative`
- `floor ceiling round truncate`
- `floor/ ceiling/ round/ truncate/`
- `modulo remainder`
- `^`
- `abs`
- `logior logxor logand lognot`
- `logbit?`
- `ash`
- `lcm gcd`
- `for`

The `dylan-arithmetic` Module

The Dylan-Arithmetic module imports and re-exports all of the interfaces exported by the `dylan` module from the Dylan library which are excluded by the `dylan-excluding-arithmetic` module.

The `generic-arithmetic-dylan` Module

The Generic-Arithmetic-Dylan module imports and reexports all of the interfaces exported by the `dylan-excluding-arithmetic` module and the `generic-arithmetic` module.

The `dylan-excluding-arithmetic`, `dylan-arithmetic`, and `generic-arithmetic` modules provide convenient building blocks for programmers to build the particular set of global name bindings they wish to work with. The purpose of the `generic-arithmetic-dylan` module is to provide a standard environment in which generic arithmetic is the norm, for those programmers who might want that.
1.6.3 Using Special Arithmetic Features

As noted in *The Generic-Arithmetic library*, the Generic-Arithmetic library provides an extensible protocol for adding specialized arithmetic functionality to your applications. By using the Generic-Arithmetic library alongside a special implementation library, you can make the standard arithmetic operations support number types such as big (64-bit) integers, or complex numbers.

This section provides an example of extending the basic Dylan arithmetic features using the Generic-Arithmetic library and the Big-Integers implementation library.

To use special arithmetic features, a library's `define library` declaration must use at least the following libraries:

- `common-dylan`
- `generic-arithmetic`
- `special-arithmetic-implementation-library`

So for Big-Integers you would write:

```dylan
define library foo
  use common-dylan;
  use generic-arithmetic;
  use big-integers;
  ...
end library foo;
```

Next you have to declare a module. There are three ways of using big-integer arithmetic that we can arrange with a suitable module declaration:

1. Replace all integer arithmetic with the big-integer arithmetic
2. Use both, with normal arithmetic remaining the default
3. Use both, with the big-integer arithmetic becoming the default

To get one of the three different effects described above, you need to arrange the `define module` declaration accordingly. To replace all integer arithmetic with big-integer arithmetic, include the following in your `define module` declaration:

```dylan
use generic-arithmetic-common-dylan;
```

(Note that the module definition should not use the Big-Integers module. The Big-Integers library is used as a side-effects library only, that is, it is referenced in the library definition so that it will be loaded. Its definitions extend the Generic-Arithmetic library.)

If you replace all integer arithmetic with big-integer arithmetic in this way, there will be performance hits. For instance, loop indices will have to be checked at run-time to see whether a normal or big integer representation is being used, and a choice must be made about the representation for an incremented value.

You can take a different approach that reduces the cost of big-integer arithmetic. Under this approach you leave normal integer arithmetic unchanged, and get access to big-integer arithmetic when you need it. To do this, use the same libraries but instead of using the `common-dylan-generic-arithmetic` module, include the following in your `define module` declaration:

```dylan
use common-dylan;
use generic-arithmetic, prefix: "ga/"; // use any prefix you like
```

This imports the big-integer arithmetic binding names, but gives them a prefix `ga/`, using the standard renaming mechanism available in module declarations. Thus you gain access to big arithmetic using renamed classes and operations like:
The operations take either instances of `<integer>` or `ga/<integer>` (a subclass of `<integer>`) and return instances of `ga/<integer>`.

Note that having imported the big-integer operations under new names, you have to use prefix rather than infix syntax when calling them. For example:

```
> ga/+ (5, 4);

not:

> 5 ga/+ 4;
```

The existing functions like `+` and `–` will only accept `<integer>` instances and `ga/<integer>` instances small enough to be represented as `<integer>` instances.

Under this renaming scheme, reduced performance will be confined to the `ga/` operations. Other operations, such as loop index increments and decrements, will retain their efficiency.

Finally, you can make big-integer arithmetic the default but keep normal arithmetic around for when you need it. Your `define module` declaration should contain:

```
use generic-arithmetic-common-dylan;
use dylan-arithmetic, prefix: "dylan/"; //use any prefix you like
```

### 1.6.4 The big-integers Library

The Big-Integers library exports a module called `big-integers`, which imports and re-exports all of the interfaces exported by the `generic-arithmetic` module of the Generic-Arithmetic library.

The Big-Integers library modifies the behavior of functions provided by the Dylan library as described in this section.

#### Specific Constructors

The Big-Integers library extends the functionality of specific constructors in the Dylan library as follows:

**limited**

G.f. method

limited `abstract-integer-class` #key min max => limited-type

Returns a limited integer type, which is a subtype of `<abstract-integer>`, whose instances are integers greater than or equal to `min` (if specified) and less than or equal to `max` (if specified). If no keyword arguments are specified, the result type is equivalent to `<abstract-integer>`. The argument `abstract-integer-class` is the class `<abstract-integer>`.

If both `min` and `max` are supplied, and both are instances of `<integer>`, then the result type is equivalent to calling `limited` on `<integer>` with those same bounds.

The Limited Integer Type Protocol is extended to account for limited `<abstract-integer>` types.

Instances and subtypes in the Big-Integers library
:: todo Fix header style here—

This is true if and only if...

... all these clauses are true

**instance?** (x, limited(<abstract-integer>, min: y, max: z))

**instance?**(x, <abstract-integer>) (y <= x) (x <= z)

**instance?** (x, limited(<abstract-integer>, min: y))

**instance?**(x, <abstract-integer>) (y <= x)

**instance?** (x, limited(<abstract-integer>, max: z))

**instance?**(x, <abstract-integer>) (x <= z)

**subtype?** (limited(<abstract-integer>, min: w, max: x), limited(<abstract-integer>, min: y, max: z))

(w >= y) (x <= z)

**subtype?** (limited(<abstract-integer>, min: w . . .), limited(<abstract-integer>, min: y))

(w >= y)

**subtype?** (limited(<abstract-integer>, max: x . . .), limited(<abstract-integer>, max: z))

(x <= z)

Type-equivalence in the Big-Integers library :: todo Fix header style here—

This is type equivalent to...

... this, if and only if...

... this is true

**limited** (<abstract-integer>, min: y, max: z)

**limited** (<integer>, min: y, max: z)

y and z are both instances of <integer>.

**limited** (<abstract-integer>, min: y, max: $maximum-integer)

**limited** (<integer>, min: y)

y is an instance of <integer>.

**limited** (<abstract-integer>, min: $minimum-integer, max: z)

**limited** (<integer>, max: z)

z is an instance of <integer>.

Type disjointness is modified as follows to account for limited <abstract-integer> types.

A limited integer type is disjoint from a class if their base types are disjoint or the class is <integer> and the range of the limited integer type is disjoint from the range of <integer> (that is, from $minimum-integer to $maximum-integer).

**Equality comparisons**

The behavior of equality comparisons in the Dylan library is modified by the Big-Integers library as follows:
Magnitude comparisons

The behavior of magnitude comparisons in the Dylan library is modified by the Big-Integers library as follows:

\[
< \text{abstract-integer1} \text{abstract-integer2} \Rightarrow \text{boolean} \\
< \text{abstract-integer} \text{float} \Rightarrow \text{boolean} \\
< \text{float} \text{abstract-integer} \Rightarrow \text{boolean}
\]

Properties of Numbers

The behavior of number property tests in the Dylan library is modified by the Big-Integers library as follows:

\[
\text{odd? abstract-integer} \Rightarrow \text{boolean} \\
\text{even? abstract-integer} \Rightarrow \text{boolean} \\
\text{zero? abstract-integer} \Rightarrow \text{boolean} \\
\text{positive? abstract-integer} \Rightarrow \text{boolean} \\
\text{negative? abstract-integer} \Rightarrow \text{boolean} \\
\text{integral? abstract-integer} \Rightarrow \text{boolean}
\]

Arithmetic Operations

The Big-Integers library modifies the behavior of the functions provided by the Generic-Arithmetic library as described below.

The actual type of the return value for all the following interfaces is determined by the contagion rules when applied to the arguments.

\[
+ \text{abstract-integer1} \text{abstract-integer2} \Rightarrow \text{abstract-integer} \\
+ \text{abstract-integer} \text{float1} \Rightarrow \text{float} \\
+ \text{float1} \text{abstract-integer} \Rightarrow \text{float} \\
- \text{abstract-integer1} \text{abstract-integer2} \Rightarrow \text{abstract-integer} \\
- \text{abstract-integer} \text{float1} \Rightarrow \text{float} \\
- \text{float1} \text{abstract-integer} \Rightarrow \text{float} \\
\times \text{abstract-integer1} \text{abstract-integer2} \Rightarrow \text{abstract-integer} \\
\times \text{abstract-integer} \text{float1} \Rightarrow \text{float} \\
\times \text{float1} \text{abstract-integer} \Rightarrow \text{float}
\]

The return value of the following interface is of the same float format as the argument:

\[
\text{negative abstract-integer} \Rightarrow \text{negative-abstract-integer}
\]

The second return value of all the following interfaces is of the same float format as the argument:

\[
\text{floor abstract-integer} \Rightarrow \text{abstract-integer1} \text{abstract-integer2} \\
\text{floor float1} \Rightarrow \text{abstract-integer} \text{float} \\
\text{ceiling abstract-integer} \Rightarrow \text{abstract-integer1} \text{abstract-integer2}
\]
The second return value of all the following interfaces is of the same float format as the first argument:

floor/ *abstract-integer1* *abstract-integer2* => *abstract-integer3* *abstract-integer4*
floor/ *float1* *abstract-integer1* => *abstract-integer2* *float2*

ceilings/ *abstract-integer1* *abstract-integer2* => *abstract-integer3* *abstract-integer4*
ceilings/ *float1* *abstract-integer1* => *abstract-integer2* *float2*

round/ *abstract-integer1* *abstract-integer2* => *abstract-integer3* *abstract-integer4*
round/ *float1* *abstract-integer1* => *abstract-integer2* *float2*

truncate/ *abstract-integer1* *abstract-integer2* => *abstract-integer3* *abstract-integer4*
truncate/ *float1* *abstract-integer1* => *abstract-integer2* *float2*

The second return value of the following interfaces is of the same float format as the second argument:

floor/ *abstract-integer1* *float1* => *abstract-integer2* *float2*
ceiling/ *abstract-integer1* *float1* => *abstract-integer2* *float2*
round/ *abstract-integer1* *float1* => *abstract-integer2* *float2*
truncate/ *abstract-integer1* *float1* => *abstract-integer2* *float2*

The return value of the following interfaces is of the same float format as the first argument:

modulo *float1* *abstract-integer* => *float*
remainder *float1* *abstract-integer* => *float*

modulo *abstract-integer1* *abstract-integer2* => *abstract-integer*
modulo *abstract-integer* *float1* => *float*
remainder *abstract-integer1* *abstract-integer2* => *abstract-integer*
remainder *abstract-integer* *float1* => *float*

The behavior of the following miscellaneous interfaces is also modified by the Big-Integers library:

^ *abstract-integer1* *integer* => *abstract-integer*
abs *abstract-integer1* => *abstract-integer*
logior #rest *abstract-integer1* => *abstract-integer*
logxor #rest *abstract-integer1* => *abstract-integer*
logand #rest *abstract-integer1* => *abstract-integer*
lognot *abstract-integer1* => *abstract-integer*
logbit? *integer* *abstract-integer* => *boolean*
ash *abstract-integer1* *integer* => *abstract-integer*
lcm *abstract-integer1* *abstract-integer2* => *abstract-integer*
gcd *abstract-integer1* *abstract-integer2* => *abstract-integer*
1.7 object-with-elements

In the DRM, the generics element and element-setter are specified to operate on instances of <collection>.

In the Open Dylan C-FFI, <C-statically-typed-pointer> would be much more useful if element and element-setter could be used with it, especially via the sugar syntax using [], but they needn’t support any other collection operations such as do or map.

To accommodate this, Open Dylan provides 2 additional classes in the hierarchy provided by the standard library:

- <object-with-elements>
- <mutable-object-with-elements>

<object-with-elements> Open Abstract Class
An invented superclass of collections and any other object to which element is applicable.

Superclasses <object>

Operations
• element

See also
• <mutable-object-with-elements>

<mutable-object-with-elements> Open Abstract Class
An invented superclass of mutable collections and any other object to which element-setter is applicable.

Superclasses <object-with-elements>

Operations
• element-setter

See also
• <object-with-elements>

1.8 Macro System Extensions

1.8.1 Traced Macros

Macros can be defined with an adjective, traced. This causes the macro expander to output some useful information during compilation:

```dylan
define traced macro when2
  { when2 (?cond:expression) ?:body end } => { if (?cond) ?body end } end;

when2 (23)
  format-out("Hello, world!
  "); end;
```

Compiling this will result in this output from the compiler:

```dylan
{ when2 } > when2 (23) format-out ("Hello, world!
  "); end
{ when2 } < if (23) format-out ("Hello, world!
  "); end
```
1.8.2 Template Calls

Note: This extension was originally described in an appendix to D-Expressions: Lisp Power, Dylan Style. This text is largely copied from there.

A number of limitations of local rewrite rules require resorting to top-level auxiliary macros. The first problem is that local rewrite rules do not have access to pattern variables defined in main rule sets. The usual solution is to employ an auxiliary macro which takes those needed variables as extra macro arguments. For example, suppose we want to add a prefix option to allow the creation of properties with a common prefix. We need to introduce an auxiliary macro so that the prefix variable can be in scope when iterating over the properties. For example, consider the following:

```dylan
define macro properties-definer
{ define properties ?kind:name
    prefixed-by ?prefix:name ?properties:* end }
=> { define variable ?kind
    = concatenate
    (prefixed-props (?)prefix"))?properties end ) }
{ define properties ?kind:name ?properties:* end }
=> { define variable ?kind
    = concatenate
    (prefixed-props ("")?properties end ) }
end macro;

define macro prefixed-props
{ prefixed-props (?prefix:name) end }
=> { #() }
{ prefixed-props (?prefix:name) ?prop:name; ?more:* end }
=> { list (?prefix #?prop),
    prefixed-props (?prefix) ?more end ) }
end macro;
```

Auxiliary macros are also needed when pattern variables must be walked in two different ways. Consider a macro, called iterate, for creating internally recursive procedures that are as convenient as loops. It has the following basic form:

```dylan
iterate name (variable = init, ...)
body
iterate
```

and has the semantics of evaluating the body with the variables initially bound to the inits and such that any call to name inside the body recursively calls the procedure with the variables bound to the arguments of the call. In writing a macro for iterate, the parenthesized fragments must be walked once to extract variables and another time to extract initial values. Unfortunately, with the current macro system, there is no way to walk the same pattern variable with more than one set of local rewrite rules. Instead, an extra copy of the pattern variable must be made and passed on to an auxiliary macro:

```dylan
define macro iterate
{ iterate ?loop:name (?args:*) ?:body end }
=> { iterate-aux ?loop (?args) (?args)
    ?body
    end }
end macro iterate;

define macro iterate-aux
{ iterate-aux ?loop:name (?args) (?inits) ?:body end }
```

(continues on next page)
Both of these reasons for needing auxiliary macros are somewhat artificial because in fact local rewrite rules are really like local functions and should allow extra arguments and should be directly callable on any pattern variable. The problem lies in the fact that the local rewrite rule is artificially tied to one pattern variable by virtue of its name.

In order to overcome this problem, we introduce a direct template call, obviating the need for auxiliary macros in many cases. This leads to a much more elegant solution to these more complicated macros. A template auxiliary rule set call has the following form:

`@rule-name{ <arbitrary-template-stuff> }`

where a new macro punctuation `@` marks a template call. For example, in the prefixed `properties-definer` macro, we can now directly invoke a local rewrite rule set with extra arguments:

```
define macro properties-definer
  { define properties ?kind:name
      prefixed-by ?prefix:name ?properties:* end }
  => { define variable ?kind
         = concatenate(?@prefixed-properties{ ?"prefix"; ?properties }) }
  { define properties ?kind:name ?properties:* end }
  => { define variable ?kind
         = concatenate(?@prefixed-properties{ ""; ?properties }) }

prefixed-properties:
  { ?prefix:name }
  => { #() }
  { ?prefix:name; ?property:name; ?more:* }
  => { list(?prefix ## ?property),
         ?@prefixed-properties{ ?prefix; ?more } }
end macro;
```

Similarly, `iterate` can now be written without auxiliary macros using two template calls:

```
define macro iterate2
  { iterate2 ?:name (?bindings:* ) ?:body end }
  => { local method ?name (?@vars{ ?bindings }) ?body end;
       ?name(?@inits{ ?bindings }) }
vars:
  { }
```

(continues on next page)
We can also introduce a template macro call

```dylan
@ { <arbitrary-template-stuff-that-forms-a-macro-call> }
```

which acts as a kind of shorthand for the `:macro` constraint and permits the definition of macros for use as shared rewriting tools. For example:

```dylan
define traced macro mcreverse
  { mcreverse(?list:* ) } => { ?list }
end macro;

define traced macro user
  { user(?stuff:* ) } => { list(@ { mcreverse(?stuff) } ) }
end macro;
```

where the `traced` modifier causes macro expansion to be traced. For example, here’s the trace for `user(1, 2, 3)`:

```
{ user } > user(1, 2, 3)
{ mcreverse } > mcreverse(1, 2, 3)
{ mcreverse } < 3,2,1
{ user } < list(3,2,1)
```

Like normal macro calls, a new hygiene context is created for `@ { }` calls, so you could define `gensym` thusly:

```dylan
define macro gensym
  { gensym() } => { gensymed-name }
end macro;
```

### 1.9 Parser Expansions

**Warning:** This is a highly experimental extension. It is not well specified, is not supported by any of the editor integrations or syntax highlighters, and may change or disappear in the future.

The syntax:
The `<text>` part can be either delimited or undelimited. Undelimited text can contain anything but commas, semicolons, brackets of any kind, and whitespace. There is no \ escape processing. All the following are valid:

```dylan
#:http://opendylan.org/
#:time:12:30am
#:date:12/3/2000
#:file:D:\dylan\sources\dylan-lang@googlegroups.com
```

In the delimited form, you have a choice of delimiters: ". . .", (...) [, ] {}, `...`, `...`. Within the delimiters, only the matching close delimiter must be escaped. The selection allows you to choose the delimiter requiring least escaping for the enclosed data. The text (less the delimiters) is passed to the parsing function. Examples:

```dylan
#:file:"C:\Program Files\Open Dylan\."  
#:html:{<html>
  <head><title>Foo</title></head>
  <body bgcolor="#FFFFFF">
  </body>
  </html>}
```

An example parser:

```dylan
define method html-parser
  (text :: <byte-string>)
  => (doc :: <html-document>)
  make(<html-document>, text: text)
end method;
```

If an appropriate function isn’t defined, you get a standard unbound variable reference message indicating the # literal.

### 1.10 Alternative Curry Syntax

**Warning:** This is an experimental extension. It is not well specified and may change or disappear in the future. This documentation is largely in the format of a DEP in case we want to propose it in the future.

#### 1.10.1 Abstract

An alternative syntax for currying functions has been provided within Open Dylan.

#### 1.10.2 Copyright

This document is public domain.
1.10.3 Specification

Functions called with an “omitted” or placeholder parameter return a curried function as if `curry` and `rcurry` had been used.

There may be multiple placeholder parameters and they can be in any parameter position. A curried function that has 2 placeholder parameters will need 2 parameters when called.

This can be used with operators as well.

1.10.4 Motivation

There are situations in which using `curry` and `rcurry` can result in confusing code that is difficult to understand at a quick glance. The alternative syntax provides an easy to understand way to set up a curried function.

1.10.5 Rationale

The alternative curry syntax was implemented in 1999 by Keith Playford and has been in the compiler since then. It does not break any existing code.

1.10.6 Examples

In this example, the two curried functions will have the same result:

```dylan
let to-string-1 = curry(as, <byte-string>);
let to-string-2 = as(<byte-string>, _);
```

As indicated earlier, multiple arguments being omitted is supported:

```dylan
define function adder(a, b, c)
  a + b + c
end;
let a = adder(_, 2, _);
let r = a(1, 3); // r == 6
```

Placeholder arguments can be used with operators as well:

```dylan
let incr = _ + 1;
let double = _ * 2;
```

These can be used inline with functions that take functions as arguments:

```dylan
format-out("\%=", map(_ + 1, #(1, 2, 3, 4));)
```

Taking some examples from the Open Dylan compiler, some simplifications are possible:

```dylan
map(method (o) print-condition(o, #t) end, o.subnotes)
let outer-values
  = map(method (variable :: <string>)
       jam-variable(jam, variable, default: #f)
       end,
       variables);
```
could be rewritten as:

```dylan
map(print-condition(_, #t), o.subnotes)
let outer-values = map(jam-variable(jam, _, default: #f), variables);
```

### 1.10.7 Backwards Compatibility

There are no backwards compatibility considerations for this functionality as the `_` is not a valid variable name in the language as specified by the DRM.

### 1.10.8 Reference Implementation

This functionality has been present within Open Dylan since 1999. The code is largely focused in these methods in `dfmc/conversion/convert.dylan`:

- curried-arguments?
- convert-curried-function-call

All the other language extensions are described in *The common-dylan Library*. 
LID FILE FORMAT

The DRM defines an interchange format for Dylan source files, but does not define an interchange format for Dylan libraries. Without such an agreed format, different Dylan development environments would find it difficult to import and build libraries developed using another Dylan vendor’s environment. It would also be impossible to automate the process of importing a library into another environment.

LID solves this problem. It allows you to describe Dylan library sources in a form that any Dylan environment should be able to understand. Harlequin and other Dylan vendors have adopted LID to make it easier to port applications from one environment to another.

Note: LID is a convention, and not an extension to the Dylan language.

Note: The Open Dylan environment can convert LID files to its own internal project file format, the .hdp file. It can also save project files as LID files with the File > Save As command in the project window.

2.1 LID files

LID works by supplementing each set of library sources with a LID file. A LID file describes a Dylan library using a set of keyword statements. Together, these statements provide enough information for specifying and locating the information necessary to build a library from its source. This means all Dylan libraries designed for interchange consist of at least two files: a LID file, and one or more files containing the library source code.

Thus a LID file performs a similar function to the makefile used in some C and C++ development environments. LID files have the file extension .lid. Every file referred to by a LID file must reside in the same folder (directory) as the LID file.

2.2 LID keyword statements

A LID file consists of a series of keyword/value statements, just like the Dylan source file interchange format. In this section, we describe the standard LID keywords.

2.2.1 Library:

LID file keyword
2.2.2 Files:

Library: *library-name*

Names the Dylan library being described. The *library-name* must be the name of the Dylan library that the LID file describes. This keyword is required in every LID file, and it may appear only once per LID file.

Files: *file-designators*

Associates a set of files with the library named by the Library: keyword. This keyword can appear one or more times per LID file. Every file specified is considered to be associated with the library.

A file designator is something that can be mapped to a concrete file name. Only one file designator can appear per line of the LID file. See File naming conventions for a description of the format of file designators and how they are mapped to concrete file names.

The order in which the designated source files are specified with the Files: keyword in the LID file determines the initialization order across the files within the defined library.

All the files must be specified relative to the same folder (directory) as the LID file.

2.2.3 Synopsis:

Synopsis: *arbitrary text*

A concise description of the library.

2.2.4 Keywords:

Keywords: *comma-separated phrases*

Any number of phrases, separated by commas, that are relevant to the description or use of the library.

2.2.5 Author:

Author: *arbitrary text*

The name of the library’s author.

2.2.6 Major-Version:
The current major version number of the library.

2.2.7 Minor-Version:

LID file keyword

The current minor version number of the library.

2.2.8 Description:

LID file keyword

A description of the library. The intention of this keyword is to provide a fuller, less concise description than that given by the Synopsis: keyword.

2.2.9 Comment:

LID file keyword

Any additional comments about the library.

2.3 Open Dylan’s LID extensions

This section contains extensions to LID that Open Dylan supports.

2.3.1 Specifying foreign files and resource files

The following keywords allow you to specify that files of foreign source code and resource files are a part of the library.

C-Source-Files:

LID file keyword

Identifies one or more C source files which are to be included as part of the library. Dylan environments copy these files to their build area and ensure that they are compiled by the appropriate batch file. The filenames specified must include the .c suffix.
C-Header-Files:
LID file keyword

Identifies one or more C header files included as part of the library. Dylan environments copy these files to their build area and ensure that they are compiled by the appropriate batch file. Any files specified using the C-Source-Files: or RC-Files: keywords depend on these header files in order to decide when they need to be recompiled. The file names given here must include the .h suffix.

C-Object-Files:
LID file keyword

Identifies one or more C object files included as part of the library. Dylan environments copy these files to their build area and ensure that they are compiled by the appropriate batch file and included in the final output as .DLL or .EXE files. The file names given here must include the .obj suffix on Windows or .o on other platforms, except when using this keyword in conjunction with a static library.

In some situations, a static library needs to be copied into the build area and linked into the project. This is typically when writing a binding for an external library written in C. In this situation, the C-Object-Files keyword may be useful.

RC-Files:
LID file keyword

Identifies one or more resource files to be included as part of the library. Dylan environments copy these files to their build area and ensure that they are compiled by the appropriate batch file. The resulting resource object files are included in the .DLL or .EXE built for the library. The file names given here must include the .rc suffix.

C-Libraries:
LID file keyword

Identifies one or more C libraries to be included in the link phase when building the .DLL or .EXE for the library. Paths to search for libraries can also be added with this keyword. Arbitrary linker options can not be specified using this keyword.

On Windows, this value for this keyword is passed directly to the linker. However, on Unix and macOS, the requirements are a bit more stringent and the arguments should be passed one per line and be one of the following:

- **-L path**: Add a path to the search path for shared libraries.
- **-llibrary**: Link against the specified shared library. This should be either in the regular linker search path or have a path specified via a -L flag.
library.a: Link against the specified static library.

Note: Note that this may cause a problem if you are using this to link a static library that hasn’t been built with -fPIC into a shared library on the x86_64-linux platform.

Note: In general, you don’t want to use this keyword to link a static library into a shared library since this keyword propagates to dependent libraries as discussed below.

-F path: Add a path to the search path for frameworks. (macOS only)

-framework framework: Link against the specified shared library. This should be either in the regular linker search path or have a path specified via a -F flag. (macOS only)

Unlike the other keywords described in this section, the C-Libraries: keyword propagates to dependent libraries. For example, suppose library A uses library B, and the LID file or library B specifies

C-Libraries: foo.lib

In this case, both library A and library B are linked against foo.lib.

2.3.2 Specifying compilation details

The following keywords control aspects of compilation for the library.

LID:

LID keyword

LID: *file-name.lid*

Specifies the name of a LID file to process and includes the settings contained in that file into the current LID file.

This is commonly used to share common definitions and settings between platform or OS specific LID files.

Jam-Includes:

LID keyword

Jam-Includes: *file-name.jam*

Specifies the name of a JAM file to process. This is typically used when integrating with a third party library and needing custom flags for the C compiler or linker.

An example JAM (for a library, not an executable) file might look like:

```dylan
{
local _dll = [ FDLLName $(image) ] ;
LINKLIBS on $(._dll) += `pkg-config --libs gtk+-3.0` ;
CCFLAGS += `pkg-config --cflags gtk+-3.0` ;
}
```

The use of backticks `...` will execute the command enclosed within and return the output of that command.

2.3. Open Dylan’s LID extensions
Executable:

LID keyword

Executable: *name*

Specifies the name of the executable (that is, .DLL or .EXE) file to be generated for this library.
The suffix (.DLL, .EXE) should not be included in the name as the appropriate suffix will be added automatically.
If this keyword is not specified, the compiler generates a default name for the executable from the name of the library.
With some library names, particularly when you are building a DLL, you may need to specify this keyword to override the default name and avoid conflicts with other DLLs from a third party.

Base-Address:

LID keyword

Base-Address: *address*

Specifies the base address of the DLL built from this Dylan library. The address must be a hexadecimal value. For convenience, you can use either Dylan (#xNNNNNNNN) or C (0xNNNNNNNN) notations when specifying the address.
This base address is ignored when building a .EXE file.
If this keyword is not specified, the compiler will compute a default base address for the library. However, it is possible for more than one library to end up with the same default base address. If an application uses any of these libraries, all but one of them will have to be relocated when the application starts. This process is automatic, but cuts down on the amount of sharing, increases your application’s memory footprint, and slows down load time. In such circumstances, you may want to give one or more libraries an explicit base address using this keyword.

Note: This keyword is only used on Windows and is ignored on other platforms.

Linker-Options:

LID keyword

linker-Options: *options*

Specifies additional options and libraries to be passed to the linker when building this DLL or EXE. Unlike the C-Libraries: keyword, the options and libraries specified here apply only to this Dylan library; they are not propagated to any libraries which use this library.

2.4 File naming conventions

In practice, importing a source distribution into a Dylan program involves unpacking the source distribution into its own subtree and then informing the environment of the location of the tree root. The environment then walks the entire subtree locating LID files, which describe the libraries in the distribution by giving a name to, and designating the source files associated with, each library.
Importing a Dylan program into the environment in this way requires two things:

1. That the LID files in the distribution can be identified.
2. That the file designators supplied to the Files: keyword in LID files can be mapped to the corresponding source filenames on disk.

If you are importing files from a platform that does not insist on, or conventionally use, standard filename suffixes to identify the filetype (such as MacOS), then you must rename your source files as follows:

- LID files must be given filenames with the suffix .lid.
- Dylan source files must be given filenames with the suffix .dylan.

The file designators that appear in LID files may be a string of characters of any length, constructed from the set of hyphen, underscore, and the mixed-case alphanumeric characters. Note that you do not have to specify the source filename suffix as part of the filename designator. This ensures that the LID files themselves do not need to be edited when importing source code from a platform, such as MacOS, that does not insist on particular filename suffixes to specify the file type.

The name of a LID file is not significant, and in particular need not be the same as the library name. Hierarchical directory structure can be used to organize multi-library systems as long as the files directly associated with each library are in a single directory.

### 2.5 Application example

This section contains an example of a complete Dylan application that uses a generic factorial calculation routine to return the value of the factorial of 100. Two libraries are defined: the factorial library provides an implementation of the generic factorial routine, and the factorial-application library provides a method that calls the generic routine and returns the appropriate result.

File: fact.lid. LID file describing the components of the factorial library.

```
Library: factorial
Synopsis: Provides a naive implementation of the factorial function
Keywords: factorial, integer, simple, recursive
Files: library
    fact
```

File: library.dylan. Defines the factorial library and its one module.

```
Module: dylan-user

define library factorial
    use dylan;
    export factorial;
end;

define module factorial
    export fact;
end;
```

File: fact.dylan. Defines the method for calculating a factorial.

```
Module: factorial

define generic fact(n);

define method fact(n == 0) 1;
```

(continues on next page)
end;

define method fact(n)
  n * fact(n - 1);
end;

File: *app.lid*. LID file describing the components of the *factorial-application* library.

Library: factorial-application
Synopsis: Computes factorial 100
Files: library
  app
Start-Module: factorial-application
Start-Function: main

File: *library.dylan*. Defines the *factorial-application* library and its one module.

Module: dylan-user

define library factorial-application
  use dylan;
  use factorial;
end library;

define module factorial-application
  use dylan;
  use factorial;
end module;

File: *app.dylan*. Defines a routine that calls the factorial routine.

Module: factorial-application

define method main (#rest ignore)
  fact(100);
end method;

The following example demonstrates how files of foreign source code and resource files can be integrated into a Dylan library:

Library: app-with-foreign-code
Synopsis: Uses some C code and resources
Files: dylan-code
  C-Source-Files: first.c
    second.c
  C-Header-Files: headers.h
RC-Files: extra-resources.rc
The C-FFI (C foreign function interface) library provides a means of interfacing a Dylan application with code written in the C language. The C-FFI library is available to applications as the module C-FFI in the library C-FFI.

The C-FFI library consists of macros, classes, and functions that you can use to write a complete description of the Dylan interface to a C library. Compiling this description generates a set of Dylan classes and functions through which Dylan code can manipulate the C library’s data and call its functions. Interface descriptions can also allow C code to call into Dylan; compiling such a description generates entry points compatible with C’s calling conventions.

Contents

• The c-ffi Library
  – Overview
    ∗ C types in Dylan
    ∗ C functions in Dylan
    ∗ C variables in Dylan
  – Notes on Linking
    ∗ Linking against a Library
    ∗ Linking against a macOS Framework
    ∗ Using pkg-config
  – Tracing FFI Calls
  – Terminology
  – Basic options in C-FFI macros
  – Designator classes
    ∗ Designator type properties
    ∗ Designator class basics
    ∗ Fundamental numeric type designator classes
    ∗ Pointer designator classes and related functions
    ∗ Structure types
    ∗ Union types
    ∗ Notes on C type macros
3.1 Overview

This section is an overview of the C-FFI library, introducing the most commonly used constructs as the basis for examples.

The C-FFI library provides a set of macros that can be used to describe a C interface in a form that the Open Dylan compiler can understand; we call these macros the C interface definition language.

The C interface definition language provides macros that correspond to each of C’s type, function, variable, and constant defining forms. These macros define Dylan classes that designate and encapsulate instances of C types, Dylan methods through which to manipulate C variables and call out to C functions, and functions with C-compatible entry points through which to call in to Dylan from C.

In addition to the interface definition language, the C-FFI library provides run-time methods and functions for allocating, manipulating and destroying instances of C data structures. For example, using these facilities you can allocate C structs and arrays, and access and set their elements.

3.1.1 C types in Dylan

When you use the interface definition language to describe a C type to the Dylan compiler, the compiler generates a new Dylan class. This class is said to designate the C type, which means that it carries with it the essential properties of the C type such as its size and alignment.

You can use this designator class in subsequent interface definition forms to specify which elements involve the designated C type. A designator class also carries with it the information on how to interpret the untyped C data as a tagged Dylan object.

The C-FFI library contains predefined designator classes for C’s fundamental types like int and double. The names of these predefined Dylan classes are formed from the C name of the fundamental type being designated. The designator class name for a particular C type formed using Dylan’s standard class-naming convention; it is prefixed with “C-”, hyphenated if it contains more than one word, and enclosed in angle brackets. For example, the C-FFI library provides the class <C-int> to designate the C type int; it designates double by the class <C-double>, and unsigned long by the class <C-unsigned-long>.
Note: Since Dylan variable names are compared without sensitivity to case, the capitalization of the “C” in the names above, and in other Dylan names appearing in this document, is not binding and can safely be ignored.

The C-FFI library also provides predefined classes designating pointers to C’s fundamental numeric types. To do so, it adds a * to the fundamental C type designator. For example <C-double*> designates the C type double*.

The following is an example of defining and using designator classes. Suppose we have the following C struct:

```c
typedef struct {
    unsigned short x_coord;
    unsigned short y_coord;
} Point;
```

We describe C structs to Dylan using the macro `define C-struct`:

```dylan
define C-struct <Point>
    slot x-coord :: <C-unsigned-short>;
    slot y-coord :: <C-unsigned-short>;
end C-struct;
```

This form defines a new designator class `<Point>` for a structure type corresponding to the C type `Point`. We designate the types of the slots of `<Point>` using the Dylan classes designating the C types used in the definition of `Point`. In this case, both slots are of the C type `unsigned short` which is designated by the predefined class `<C-unsigned-short>`. The information about the C type `unsigned short` carried by this designator class allows the compiler to compute the size, alignment, and layout of the struct. The compiler records the struct’s size and alignment and associates them with `<Point>`. The designator class `<Point>` can then be used in the definition of other types, functions, and variables. For example, we could describe

```c
typedef struct {
    Point start;
    Point end;
} LineSegment;
```

like this:

```dylan
define C-struct <LineSegment>
    slot start :: <Point>;
    slot end :: <Point>;
end C-struct;
```

As well as acting as a static information carrier for use in other FFI definitions, a designator class can also be instantiable, in which case Dylan uses an instance of the designator class to represent an object of the C type it designates when that object is passed from the “C world” to the “Dylan world”.

Note: Only classes that designate C pointer types can be instantiated in this way. Instances of C’s fundamental numeric value types like `int`, `char`, and `double` are just converted to an equivalent Dylan object with the same value. The `<Point>` class is not an instantiable class in Dylan because there is nothing in Dylan that corresponds to a C struct. However, the C-FFI does provide a Dylan representation of a pointer to a C struct.

To illustrate, here is an example interaction involving a C struct containing some pointer-typed slots and some slots whose types are fundamental numeric types:

```dylan
define C-struct <Example>
    slot count :: <C-int>;
    slot statistic :: <C-double>;
    slot data :: <C-char*>;
end C-struct;
```
This example defines the two designator types `<Example>` and `<Example*>`; the slots `count` and `statistic` have fundamental numeric types while `data` and `next` have pointer types. The getter and setter methods for the slots are defined for instances of `<Example*>`.

Suppose there is a function `current-example` that returns an initialized `<Example*>` struct. The following transactions illustrate what you get when you read the slots of the structure it returns:

\[
\begin{align*}
&? \text{define variable } \text{example} = \text{current-example}(); \\
&// \text{ Defined example} \\
&? \text{example.count}; \\
&4 \\
&? \text{instance?(example.count, <integer>)}; \\
&\#t \\
&? \text{example.statistic}; \\
&10.5 \\
&? \text{instance?(example.statistic, <float>)}; \\
&\#t
\end{align*}
\]

The interactions above show that if we access structure slots that were defined as being of one of C’s fundamental numeric types, we get a Dylan number of the equivalent value. The same thing happens if an imported C function returns a fundamental numeric type: a Dylan number with the same value appears in Dylan. Similarly, when setting slots in structs expecting numbers or passing objects out to C functions expecting numeric arguments, you should provide a Dylan number, and the C-FFI will convert it automatically to its C equivalent.

\[
\begin{align*}
&? \text{example.data}; \\
&\{<\text{C-char}> \text{ pointer } \#xff5e00\} \\
&? \text{instance?(example.data, <C-char*>)}; \\
&\#t \\
&? \text{example.next}; \\
&\{<\text{Example}> \text{ pointer } \#xff5f00\} \\
&? \text{instance?(example.next, <Example*>)}; \\
&\#t
\end{align*}
\]

The interactions above show that accessing structure slots with a pointer type results in an instance of the Dylan class that designates that type. Again, the same thing happens if an imported C function returns a pointer type: an instance of the corresponding designator class is created. Similarly, when setting slots in structs expecting pointers or passing objects out to C functions expecting pointer arguments, you should provide an instance of the Dylan designator class for that pointer type, and the C-FFI will convert it automatically to the raw C pointer value.

Later sections describe all the macros available for defining C types and the functions available for manipulating them.

### 3.1.2 C functions in Dylan

When you use the interface definition language to describe a C function to the Dylan compiler, the compiler generates a new Dylan function. This wrapper function accepts Dylan arguments and returns Dylan results. It converts each of its arguments from a Dylan object to a corresponding C value before calling the C function it wraps. The C-FFI converts any results that the C function returns into Dylan objects before returning them to the caller.

In order for Dylan to be able to call into C correctly, C functions must be described to Dylan in the same detail a C header file would provide a calling C program. Specifically, for every function we must provide the C name and the type of its arguments and results. As with struct definitions, these types are indicated by naming the designator classes corresponding to the C types involved in the C-FFI description of the C function.
The following is an example of defining and using wrapper functions. Suppose we have the following `extern C` function declaration:

```c
extern double cos (double angle);
```

We describe C functions to Dylan using the C-FFI macro `define C-function`:

```dylan
define C-function C-cos
    parameter angle :: <C-double>;
    result cos :: <C-double>;
    c-name: "cos"
end C-function;
```

The name appearing immediately after the `define C-function` is the name we want to give to the Dylan variable to which our wrapper function will be bound. We call it C-cos. We also give the actual C name of the function we want to wrap as the value of the keyword `c-name:`.

Once we have compiled the definition — and assuming the compiled version of the C library implementing `cos` has been linked in with the Dylan application — we can call the wrapper function just like any other Dylan function:

```
? C-cos(0.0);
1.0
```

As we noted above, when values are passed back and forth between Dylan and C, the C-FFI performs automatic conversions. In this case, the type of the parameter and the result are both fundamental numeric types which means that the C-FFI will accept and return Dylan floats, converting to and from raw C floats as necessary.

As well as making C functions available to Dylan code, the C-FFI allows us to make Dylan functions available to call from C code. We do this by defining a `C-callable` wrapper function. A C-callable wrapper is a Dylan function that a C program can call. The C-callable wrapper has a C calling convention. When a C program calls a C-callable wrapper, the C-FFI performs the necessary data conversions and then invokes a Dylan function.

You can pass C-callable wrappers into C code for use as callbacks. You can also give them names visible in C, so that C clients of Dylan code can call into Dylan directly by invoking a named function.

The argument and result conversions performed by C-callable wrappers are just like those done within Dylan wrapper functions. The macro that defines C-callable wrappers is called `define C-callable-wrapper` and we describe it in detail later. For now, consider the following simple example. Suppose we have a C `extern` function declaration `AddDouble`:

```c
extern double AddDouble (double x, double y);
```

This function is intended to return the sum of two `double` values. Instead of implementing the function in C, we can implement it in Dylan using Dylan's generic function `+`. All we need to do is define a C-callable wrapper for `+`, as follows:

```dylan
define C-callable-wrapper AddDoubleObject of 
    parameter x :: <C-double>;
    parameter y :: <C-double>;
    c-name: "AddDouble"
end C-callable-wrapper;
```

We can now call `AddDouble` in C. Our wrapper will be invoked, the C arguments will be converted and passed to Dylan's `+` generic function, and then the result of the computation will be converted and passed back to C:

```
{ extern double AddDouble (double x, double y);
  double result;
}(continues on next page)
result = AddDouble(1.0, 2.0);
}

The C-FFI binds the Dylan variable AddDoubleObject to a Dylan object representing the function pointer of the C-callable wrapper. This reference allows the C-callable wrapper to be passed to a C function expecting a callback argument.

### 3.1.3 C variables in Dylan

When you use the interface definition language to describe a C variable to the Dylan compiler, the compiler generates new Dylan getter and setter functions for reading and setting the variable’s value from Dylan. If the variable is constant, it defines a getter function only.

The getter function converts the C value to a Dylan value before returning it according to the variable’s declared type. Similarly, the setter function converts its argument, as Dylan value, into a C value before setting the C variable. These conversions happen according to the same rules that apply to other C-Dylan world transition points, such as argument passing or structure slot access.

In order for Dylan to be able to access a C variable correctly, we must describe the variable to Dylan in the same detail that a C header file would give to a C program that uses it. Specifically, we must provide the C name and the type of the variable. As with struct and function definitions, we indicate C types by naming the appropriate Dylan designator classes.

Here is an example of defining and using C variables. Suppose we have the following `extern` C variable declaration:

```c
extern double mean;
```

We describe C variables to Dylan using the C-FFI macro `define C-variable`:

```dylan
define C-variable C-mean :: <C-double>
c-name: "mean";
end C-variable;
```

The name immediately after the `define C-variable` is the name of the Dylan variable to which the getter for our C variable will be bound. In this case it is `C-mean`.

We give the C name of the variable as the value of the keyword `c-name:`. Once we have compiled the definition — and assuming the compiled version of the C library defining `mean` has been linked in with the Dylan application — we can call the getter function just like any other Dylan function:

```dylan
? C-mean();
1.5
```

By default, the C-FFI also defines a setter function for the variable. The setter name uses Dylan’s convention of appending “-setter” to the getter name.

```dylan
? C-mean() := 0.0;
0.0
? C-mean();
0.0
```

As described above, when values are passed back and forth between Dylan and C, the C-FFI performs automatic conversions. In this case, the type of the variable is a fundamental numeric type which means that the C-FFI accepts and returns Dylan floats, converting to and from raw C floats as necessary.
Note: We could achieve the same result by using the `define C-address` macro, which defines a constant that is a pointer to the storage allocated for the C variable.

### 3.2 Notes on Linking

When using C-FFI, you will typically need to link in an existing library or framework. *LID files* provide many options for controlling the compilation and linking of the project depending on what exactly is required.

#### 3.2.1 Linking against a Library

This can be done in a *LID file* using the `C-Libraries` keyword. This supports both static and shared libraries. It also supports specifying a search path. For example:

```text
C-Libraries: -lGL
```

#### 3.2.2 Linking against a macOS Framework

Just as with a regular shared library, the `C-Libraries` keyword in a *LID file*. For example:

```text
C-Libraries: -framework OpenGL
```

#### 3.2.3 Using pkg-config

Libraries that use “pkg-config” are slightly more complicated to work with in that they require using the `Jam-Includes` keyword and an additional file within the project. The GTK+ bindings provide multiple examples of this.

In the LID file, you would include the additional Jam file:

```text
Jam-Includes: gtk-dylan.jam
```

And you would provide the additional Jam file:

```jaml
{  local _dll = [ FDLLName $(image) ] ;
   LINKLIBS on $(/_dll) += 'pkg-config --libs gtk+-3.0' ;
   CCFLAGS += 'pkg-config --cflags gtk+-3.0' ;
}
```

### 3.3 Tracing FFI Calls

When working with the C-FFI, it is very useful to be able to trace what is happening, what is getting called, what the arguments are, and what the return value is. To aid in this, C-FFI enables the programmer to enable tracing.

To do this, you will need to exclude the default implementation of tracing when importing the `c-ffi` module and define your own implementation.
In your library.dylan, you would change your module declaration:

```dylan
to:

```dylan
Note that we’ve used the format-out module from the io library in addition to the exclusion.

After doing that, you can define your own implementation of tracing such that your implementation is in the same lexical scope as the C-function definitions that you want to trace:

```dylan
When this is run, you will see output like:

```dylan
3.4 Terminology

For the rest of this chapter, we adopt the following terminology, hopefully not too inconsistent with common C terminology:

- **Base type** Basic units of data storage (C’s variously sized integers, characters, and floating point numbers) and aggregate records (structs and unions).
- **Derived type** A type based on some other type (C’s pointer, array, and function types).
- **Fundamental numeric type** One of C’s integer or floating point types. This does not include pointer types, structure types, or union types.

3.5 Basic options in C-FFI macros

The defining macros of the C-FFI share a consistent core set of options which are worth describing here:
• A *c-name* argument. Every defining form allows you to specify the corresponding C entity through the keyword *c-name*. It is optional in some forms but required in others. You can define types that have no named opposite number in C, and the *c-name* option is always optional in type definitions. On the other hand, you must always name an imported C function or variable so that Dylan knows the correct name from the compiled C library to link with.

In general, any C entity you can declare in C using *extern* can only be found by the C-FFI if you pass a *c-name* argument to the corresponding C-FFI definition.

The sole exception to this is the *define objc-selector* form which instead takes a *selector*: keyword.

• A *pointer-type-name* argument. All the type-defining forms allow you to name the type for a pointer to the type being defined. This is normally specified throughout the *pointer-type-name*: keyword option.

### 3.6 Designator classes

As *Overview* explained, the C-FFI defines some Dylan classes to designate C types and to describe how they are passed to and from Dylan. These *designator classes* carry with them static information about the C type they designate.

The C-FFI library provides an initial set of designator classes corresponding to C’s fundamental types, as well as macros for generating designator classes corresponding to C’s pointer types and for extending the translation between C data and Dylan objects.

Designator classes that correspond to fundamental numeric types are not instantiable. When you pass a numeric value to Dylan from C, the C-FFI simply generates a Dylan number with the same value. Similarly, a Dylan number passed to C is converted to a C number of the appropriate type and value.

Each of the fundamental designator classes indicate a conversion to or from a unique Dylan class. The conversions that take place are described in detail in the documentation for each designator class.

The main reasons for this design are increased efficiency, simplified implementation, and added convenience when working with numeric values. The designator classes for the numeric types could have been made instantiable and placed beneath the appropriate number protocol classes in Dylan, but these extra representations in such a fundamental area could cause problems for Dylan compilers. In addition, to make these instantiable designator classes convenient to work with, the C-FFI would also have to define methods on the standard arithmetic and comparison operators. It is simpler to represent these fundamental types with existing Dylan objects.

However, the designator classes that correspond to pointer types *are* instantiable. When you pass a pointer from C to Dylan, the C-FFI constructs an instance of the appropriate designator class that contains the raw address. A wrapped pointer like this can be passed out to some C code that is expecting a compatible pointer — the C-FFI extracts the raw address before handing it to C code. The documentation for the abstract class `<C-pointer>` describes the compatibility rules for pointer types.

This feature of pointer designator classes allows Dylan code to be typed to a specific kind of pointer. For example, you can define methods that specialize on different kinds of pointer on the same generic function.

#### 3.6.1 Designator type properties

To understand how designator classes work, it is useful to know about their properties. A few of these properties are accessible programatically, but others are implicit and only really exist in the compiler. Some of the properties may be empty.

A *referenced type* is the designator type to which a pointer refers. A designator’s *referenced-type* only has a value for subtypes of `<C-statically-typed-pointer>`. Programs can access the referenced type through the function *referenced-type*. 

#### 3.6.2 Designator class properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>referenced-type</em></td>
<td>The designator type to which a pointer refers. It only has a value for</td>
</tr>
<tr>
<td></td>
<td>subtypes of <code>&lt;C-statically-typed-pointer&gt;</code></td>
</tr>
<tr>
<td><em>compatibility</em></td>
<td>The rules for the compatibility of pointer types.</td>
</tr>
<tr>
<td><em>instantiable</em></td>
<td>Whether the designator class is instantiable.</td>
</tr>
<tr>
<td><em>pointer-rate</em></td>
<td>The rate at which the designator class is used as a pointer.</td>
</tr>
<tr>
<td><em>pointer-class</em></td>
<td>The class that represents the designator as a pointer.</td>
</tr>
</tbody>
</table>

3.6. Designator classes 49
A designator class’s `pointer-type` only has a value for each of those types that has a pointer designator type that refers to it. Most of the constructs that define a new designator type also define a pointer-type for that designator. Many of the macros that define designators accept a `pointer-type-name` keyword to bind the `pointer-type` of the defined designator to a given variable. The pointer-type is not programmatically available because it may not have been defined. You can assure that there is a pointer-type for a particular designator by using the macro `define c-pointer-type`.

A designator class’s `import type` and `export type` are instantiable Dylan types that describe the Dylan instantiation of a designator class when it is used in a position that imports values from C, or exports values to C.

Nearly all of the C-FFI’s designators have import and export types that are equivalent. Some, such as `<C-string>`, have different import and export types because it is possible to pass a pointer to a Dylan object to C directly without creating a C pointer object, or copying the underlying data, but when importing a string from C it is not practical to copy the contents and create a Dylan string. By default, the import and export types for any subtype of `<C-pointer>` are the class itself. You can override this by defining a new subclass with the macro `define C-mapped-subtype`.

You can define a designator’s `import-function` and `export-function` by using the macro `define C-mapped-subtype`. These functions are merely the procedural specifications for translating the C data to Dylan and back. The `import` and `export` functions are inherited when you define a subclass for a designator.

### 3.6.2 Designator class basics

**<C-value> Abstract Sealed Class**

**Discussion** The abstract superclass of all designator classes. It is a subclass of `<object>`. It has neither an `export-type` nor an `import-type`, so you cannot use it when designating a transition between C and Dylan.

**<C-void> Abstract Sealed Class**

**Discussion**

The abstract superclass of all designator classes. It is a subclass of `<C-value>`. It has neither an `export-type` nor an `import-type`, so you cannot use it when designating a transition between C and Dylan.

This class is only useful in that it is the `referenced-type` for `<C-void*>`.

**size-of Function**

Takes a designator class and returns the size of the C type that the class designates.

**Signature** `size-of designator-class => size`

**Parameters**

- `designator-class` – A subclass of `<C-value>`.

**Values**

- `size` – An instance of `<integer>`.

**Discussion**

Takes a designator class and returns the size of the C type that the class designates.

The `size-of` function can be applied to any designator class. However, if it is applied to `<C-void>`, `<C-value>`, or `<C-struct>`, it returns zero. It corresponds to C’s `sizeof` operator and returns an integer, `size`, in the same units as `sizeof` does on the target platform. It can be useful when allocating a C object whose declared size is not accurate and has to be adjusted manually.

**alignment-of Function**

Takes a designator class and returns the alignment of the C type that the class designates.
Signature alignment-of designator-class => alignment

Parameters

- **designator-class** – A subclass of <C-value>.

Values

- **alignment** – An instance of <integer>.

Discussion Takes a designator class and returns the alignment of the C type that the class designates. The alignment-of function can be applied to any designator class. It returns the alignment as an integer, in the same units as size-of does.

### 3.6.3 Fundamental numeric type designator classes

This section describes the pre-defined designator classes for fundamental C numeric types. On page Designator classes we saw that none of these designator types are instantiable: a number on one side of the interface is converted to a number on the other side with the same value.

There are some additional details to note about integer representations. Because Dylan’s integer representations do not match C’s exactly, for each of the C integer types there are three designator classes that can be used to translate Dylan representations to that C integer. The categories are plain, unsafe, and raw integers.

*Plain* integer designators — of which the class <C-unsigned-short> is an example — translate C integer values to instances of <integer>. If the integer being translated is too big for the destination, the C-FFI signals an error. There are two ways this can happen.

- On export, the C-FFI signals an error if the Dylan value has more significant bits than the C integer.
  
  This can happen if, for example, the designator is <C-unsigned-short>, and the Dylan value is negative, or if unsigned short on that platform is 16 bits wide, but the Dylan integer has more than 16 significant bits. The check will be omitted if the compiler can determine that no Dylan value outside the safe range can reach there. This can be done using a limited integer type.

- On import into Dylan, the C-FFI signals an error if it cannot represent the C value using a Dylan <integer>.
  
  This can happen with any C integer type that is more than 30 bits wide. The size of a Dylan <integer> depends on the particular platform, but it is guaranteed to be at least 30 bits in length.

The C-FFI never signals an error for the *unsafe* designator classes — of which the class <C-unsafe-unsigned-short> is an example — but if the destination is too small for the value, the most significant bits of the value are chopped off to fit into the destination. Because there is no checking, using the unsafe designator classes brings a very small performance improvement, but nonetheless you should not use them unless you are certain you will not lose any bits.

*Raw* designator classes — of which the class <C-raw-unsigned-int> is an example — represent the integer on the Dylan side as a <machine-word>. An instance of <machine-word> is guaranteed to have enough bits to represent any C long value, or any C void* value. Note that a <machine-word> value may still have more significant bits than some C integer types, and so the C-FFI may still signal an overflow error if the <machine-word> value, interpreted as indicated by the designator, has more significant bits than may be held in the indicated C type.

The integer designator classes and their mappings. shows all raw, plain, and unsafe integer designator types exported from the C-FFI module.

Table 1: The integer designator classes and their mappings.

<table>
<thead>
<tr>
<th>Designator name</th>
<th>C type</th>
<th>Dylan type(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;C-int&gt;</td>
<td>int</td>
<td>&lt;integer&gt;</td>
</tr>
</tbody>
</table>

Continued on next page
For each of the fundamental integer designator types, `<C- xxx >`, there is also a type designating pointers to that type called `<C- xxx *>`. In addition, the C-FFI defines methods for `pointer-value` and `pointer-value-setter`, with appropriate translation behavior for each of the types designating pointers to the fundamental integer designator types.

### <C-number> Abstract Sealed Class

**Superclasses** `<C-value>`

**Discussion** The abstract superclass of all classes that designate a fundamental numeric C type.

### <C-float> Abstract Sealed Class

**Discussion** The class of C floating point values.

### <C-double> Abstract Sealed Class

**Discussion** The class of C double-precision values.

---

<table>
<thead>
<tr>
<th>Designator name</th>
<th>C type</th>
<th>Dylan type(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;C-raw-int&gt;</code></td>
<td>int</td>
<td><code>&lt;machine-word&gt;</code></td>
</tr>
<tr>
<td><code>&lt;C-unsafe-int&gt;</code></td>
<td>int</td>
<td><code>&lt;integer&gt;</code></td>
</tr>
<tr>
<td><code>&lt;C-raw-signed-int&gt;</code></td>
<td>signed int</td>
<td><code>&lt;machine-word&gt;</code></td>
</tr>
<tr>
<td><code>&lt;C-unsafe-signed int&gt;</code></td>
<td>signed int</td>
<td><code>&lt;integer&gt;</code></td>
</tr>
<tr>
<td><code>&lt;C-signed-int&gt;</code></td>
<td>signed int</td>
<td><code>&lt;integer&gt;</code></td>
</tr>
<tr>
<td><code>&lt;C-raw-unsigned-int&gt;</code></td>
<td>unsigned int</td>
<td><code>&lt;machine-word&gt;</code></td>
</tr>
<tr>
<td><code>&lt;C-unsafe-unsigned-int&gt;</code></td>
<td>unsigned int</td>
<td><code>&lt;integer&gt;</code></td>
</tr>
<tr>
<td><code>&lt;C-unsigned-int&gt;</code></td>
<td>unsigned int</td>
<td><code>&lt;integer&gt;</code></td>
</tr>
<tr>
<td><code>&lt;C-unsigned-long&gt;</code></td>
<td>unsigned long</td>
<td><code>&lt;integer&gt;</code></td>
</tr>
<tr>
<td><code>&lt;C-signed-long&gt;</code></td>
<td>signed long</td>
<td><code>&lt;integer&gt;</code></td>
</tr>
<tr>
<td><code>&lt;C-unsigned-long&gt;</code></td>
<td>unsigned long</td>
<td><code>&lt;integer&gt;</code></td>
</tr>
<tr>
<td><code>&lt;C-raw-unsigned-long&gt;</code></td>
<td>unsigned long</td>
<td><code>&lt;machine-word&gt;</code></td>
</tr>
<tr>
<td><code>&lt;C-raw-signed-long&gt;</code></td>
<td>signed long</td>
<td><code>&lt;machine-word&gt;</code></td>
</tr>
<tr>
<td><code>&lt;C-unsigned-short&gt;</code></td>
<td>unsigned short</td>
<td><code>&lt;integer&gt;</code></td>
</tr>
<tr>
<td><code>&lt;C-signed-short&gt;</code></td>
<td>signed short</td>
<td><code>&lt;integer&gt;</code></td>
</tr>
<tr>
<td><code>&lt;C-unsigned-short&gt;</code></td>
<td>signed short</td>
<td><code>&lt;integer&gt;</code></td>
</tr>
<tr>
<td><code>&lt;C-raw-unsigned-short&gt;</code></td>
<td>unsigned short</td>
<td><code>&lt;machine-word&gt;</code></td>
</tr>
<tr>
<td><code>&lt;C-raw-signed-short&gt;</code></td>
<td>signed short</td>
<td><code>&lt;machine-word&gt;</code></td>
</tr>
<tr>
<td><code>&lt;C-unsigned-char&gt;</code></td>
<td>unsigned char</td>
<td><code>&lt;integer&gt;</code></td>
</tr>
<tr>
<td><code>&lt;C-signed-char&gt;</code></td>
<td>signed char</td>
<td><code>&lt;integer&gt;</code></td>
</tr>
<tr>
<td><code>&lt;C-unsafe-unsigned-char&gt;</code></td>
<td>unsigned char</td>
<td><code>&lt;integer&gt;</code></td>
</tr>
<tr>
<td><code>&lt;C-unsafe-signed-char&gt;</code></td>
<td>signed char</td>
<td><code>&lt;integer&gt;</code></td>
</tr>
<tr>
<td><code>&lt;C-raw-unsigned-char&gt;</code></td>
<td>unsigned char</td>
<td><code>&lt;machine-word&gt;</code></td>
</tr>
<tr>
<td><code>&lt;C-raw-signed-char&gt;</code></td>
<td>signed char</td>
<td><code>&lt;machine-word&gt;</code></td>
</tr>
<tr>
<td><code>&lt;C-char&gt;</code></td>
<td>char</td>
<td><code>&lt;integer&gt;</code></td>
</tr>
<tr>
<td><code>&lt;C-unsafe-char&gt;</code></td>
<td>char</td>
<td><code>&lt;integer&gt;</code></td>
</tr>
<tr>
<td><code>&lt;C-raw-char&gt;</code></td>
<td>char</td>
<td><code>&lt;machine-word&gt;</code></td>
</tr>
<tr>
<td><code>&lt;C-size-t&gt;</code></td>
<td>size_t</td>
<td><code>&lt;integer&gt;</code></td>
</tr>
<tr>
<td><code>&lt;C-unsafe-size-t&gt;</code></td>
<td>size_t</td>
<td><code>&lt;integer&gt;</code></td>
</tr>
<tr>
<td><code>&lt;C-raw-size-t&gt;</code></td>
<td>size_t</td>
<td><code>&lt;machine-word&gt;</code></td>
</tr>
<tr>
<td><code>&lt;C-ssize-t&gt;</code></td>
<td>ssize_t</td>
<td><code>&lt;integer&gt;</code></td>
</tr>
<tr>
<td><code>&lt;C-unsafe-ssize-t&gt;</code></td>
<td>ssize_t</td>
<td><code>&lt;integer&gt;</code></td>
</tr>
<tr>
<td><code>&lt;C-raw-ssize-t&gt;</code></td>
<td>ssize_t</td>
<td><code>&lt;machine-word&gt;</code></td>
</tr>
</tbody>
</table>
3.6.4 Pointer designator classes and related functions

This section describes the pre-defined classes that designate C pointer types. Subclasses of the abstract classes documented here are instantiable, and C pointers are represented in Dylan by instances of these classes.

Note: Pointer designator classes are defined for all the designator classes in *The integer designator classes and their mappings*, but are not listed here. To form the name of the pointer designator class for a particular designator class, append a * to the part of the name enclosed in angle brackets. Thus for `<C-int>` the pointer designator class is `<C-int*>`.

<C-pointer> Open Primary Abstract Class

Superclasses `<C-value>`

Discussion The abstract superclass of all classes that designate a C pointer type. Instances of concrete subclasses of `<C-pointer>` encapsulate a raw C address. The make methods on subclasses of `<C-pointer>` accept the keyword argument `address`, which must be a Dylan `<integer>` or `<machine-word>` representation of the C address.

pointer-address Function

Recovers the address from an instance of `<C-pointer>` and returns it as a Dylan `<machine-word>`.

Signature `pointer-address C-pointer => address`

Parameters

• `c-pointer` – An instance of `<C-pointer>`.

Values

• `address` – An instance of `<machine-word>`.

Discussion Recovers the address from an instance of `<C-pointer>` and returns it as a Dylan `<machine-word>`.

pointer-cast Function

Converts a pointer from one pointer type to another.

Signature `pointer-cast pointer-designator-class C-pointer => new-C-pointer`

Parameters

• `pointer-designator-class` – A subclass of `<C-pointer>`.

• `c-pointer` – An instance of `<C-pointer>`.

Values

• `new-c-pointer` – An instance of `<C-pointer>`.

Discussion Converts a pointer from one pointer type to another. The new pointer will have the same address as the old pointer.

null-pointer Function

Returns a null pointer whose type is given by the pointer-designator-class.

Signature `null-pointer pointer-designator-class => null-pointer`

Parameters

• `pointer-designator-class` – A subclass of `<C-pointer>`.

• `c-pointer` – An instance of `<C-pointer>`.
null-pointer? Function
Returns true if a pointer is null

**Signature** null-pointer? C-pointer => boolean

**Parameters**
- c-pointer – An instance of <C-pointer>.

**Values**
- boolean – An instance of <boolean>.

**Discussion** Returns #t if a pointer is null and #f otherwise.

<C-void*> Open Concrete Class

**Superclasses** <C-pointer>

**Discussion** The class designating C’s void* pointer type. No pointer-value methods are defined on this class.

<C-statically-typed-pointer> Open Abstract Class

**Superclasses** <C-pointer>, <mutable-object-with-elements>

**Discussion** The abstract superclass of all classes designating a C pointer type for a non-void base.

define C-pointer-type Defining Macro
Defines a constant bound to a pointer class designating pointers to a designator class name.

**Macro Call**

```dylan
define C-pointer-type *pointer-class-name* => *designator-class-name*
```

**Parameters**
- pointer-class-name – A Dylan variable name.

**Values**
- designator-class – A Dylan name.

**Discussion** Defines a constant bound to a pointer class designating pointers to designator-class-name. Note that the pointer type may already exist. The class defined will be open, abstract and instantiable. Objects returned by make(*pointer-class-name*) will be instances of a sealed concrete subclass of pointer-class-name.

referenced-type Function
Returns the class designating the contents type of the designated C pointer type.

**Signature** referenced-type pointer-designator-class => designator-class

**Parameters**
- pointer-designator-class – A subclass of <C-pointer>.

**Values**
- designator-class – A subclass of <C-value>.
Discussion Returns the class designating the contents type of the C pointer type designated by pointer-designator-class. The same designator class is returned whenever referenced-type is called with the same argument.

c-type-cast Function
Converts a value to a value of a specified type, according to the semantics of a C type cast.

Signature c-type-cast type value => value

Parameters

• type – See Description.
• value – An instance of <object>.

Values

• value – An instance of <object>.

Discussion
Returns the value of the second argument, converted to the type specified by the first argument, in accordance with the semantics of a C type cast. This is convenient to use when translating C code to Dylan. It may also be helpful for converting a value to the form required by a C-function wrapper argument.

The first argument can be either a C type designator or one of the Dylan classes <boolean>, <character>, <machine-word>, or any subclass of <number>. For a C type designator, the value is converted to the Dylan class which it maps to. <C-[un]*signed-short>* and <C-[un]*signed-char>* truncate the value as well as ensuring that it is an <integer>.

Example For example, with a function declared in C as

```c
Foo(long x);
```

and called as

```c
Foo((long) p);
```

if the Dylan declaration is

```dylan
define C-function Foo
  parameter x :: <C-both-long>;
  c-name: "Foo";
end;
```

then the equivalent call will be:

```dylan
Foo(c-type-cast(<C-both-long>, p));
```

which will ensure that the C semantics are preserved without needing to analyze exactly what the type cast is doing.

The functions pointer-value and pointer-value-setter perform the primitive Dylan-to-C and C-to-Dylan conversions as documented with the designator class of the pointer’s contents type (see The integer designator classes and their mappings). The C-FFI signals an error if it cannot convert the object you attempt to store in the pointer to a compatible type.

These two functions are part of a protocol for extending the C type conversions. You can define new methods for pointer-value and pointer-value-setter for types defined by define C-subtype that are subtypes of <C-pointer>.
**pointer-value** Open Generic function
Dereferences a c-typed pointer using its encapsulated raw C address.

**Signature**
pointer-value C-typed-pointer #key index => object

**Parameters**
- c-typed-pointer – An instance of `<C-statically-typed-pointer>`.

**Values**
- object – An instance of `<object>`.

**Discussion**
Dereferences c-typed-pointer using its encapsulated raw C address, and returns a Dylan object representing the object at that address. If you supply index, the pointer is treated as a pointer to an array, and the function returns the appropriate element indexed by the correct unit size.

It is an error if C-typed-pointer does not point to a valid address or is a null pointer.

**See also**
- pointer-value-setter.

**pointer-value-setter** Open Generic function
Allows you to set pointer values.

**Signature**
pointer-value-setter new-value C-typed-pointer #key index => new-value

**Parameters**
- new-value – An instance of `<object>`.
- c-typed-pointer – An instance of `<C-statically-typed-pointer>`.
- index (#key) – An instance of `<integer>`.

**Values**
- new-value – An instance of `<object>`.

**Discussion**
Allows you to set pointer values. If you supply index, the pointer is treated as a pointer to an array, and the function returns the appropriate element indexed by the correct unit size.

It is an error if C-typed-pointer does not point to a valid address or is a null pointer.

**pointer-value-address** Open Generic function
Returns a pointer of the same type as a C-typed pointer that points to the object offset into the C-typed pointer.

**Signature**
pointer-value-address C-typed-pointer #key index => object

**Parameters**
- c-typed-pointer – An instance of `<C-statically-typed-pointer>`.
- index (#key) – An instance of `<integer>`.

**Values**
- object – An instance of `<object>`.

**Discussion**
Returns a pointer of the same type as C-typed-pointer that points to the index th object offset into C-typed-pointer. The following expression is guaranteed to be true:

**Example**
pointer-value(*C-typed-pointer*, index: i) = pointer-value (pointer-value-address(*C-typed-pointer*, index: i))

**element(<C-statically-typed-pointer>) Method**
Dereferences a c-typed pointer using its encapsulated raw C address.

**Signature**
```
 element C-typed-pointer index => object
```

**Parameters**

- **c-typed-pointer** – An instance of `<C-statically-typed-pointer>`.

**Values**

- **object** – An instance of `<object>`.

**Discussion**
Dereferences a c-typed pointer using its encapsulated raw C address. Synonymous with a call to `pointer-value` that includes the optional index. Thus it does the same thing as:

```plaintext
pointer-value(*C-statically-typed-pointer*, index: *index*)
```

**element-setter(<C-statically-typed-pointer>) Method**
Allows you to set pointer values.

**Signature**
```
 element-setter new C-typed-pointer index => object
```

**Parameters**

- **c-typed-pointer** – An instance of `<C-statically-typed-pointer>`.
- **index** – An instance of `<integer>`.

**Values**

- **object** – An instance of `<object>`.

**Discussion**
Synonymous with a call to `pointer-value-setter` that includes the optional index. Thus it does the same thing as:

```plaintext
pointer-value-setter(*new*, *C-statically-typed-pointer*, index: *index*)
```

**=(<C-pointer>) Method**
Returns #t if two pointers are equal.

**Signature**
```
= C-pointer-1 C-pointer-2 => boolean
```

**Parameters**

- **c-pointer-1** – An instance of `<C-pointer>`.
- **c-pointer-2** – An instance of `<C-pointer>`.

**Values**

- **boolean** – An instance of `<boolean>`.

**Discussion**
Returns #t if two pointers are equal. This is equivalent to:

```plaintext
pointer-value(*C-typed-pointer*, index: i) = pointer-value (pointer-value-address(*C-typed-pointer*, index: i))
```
Dylan Library Reference, Release 1.0

\[
\text{pointer-address}(\text{<C-pointer-1>}) = \text{pointer-address}(\text{<C-pointer-2>})
\]

Note that operations corresponding to C pointer arithmetic are not defined on \text{<C-pointer>}. If pointer arithmetic operations are required, use \text{pointer-value} with an \text{index:} argument.

**See also**

- \text{pointer-value}.

\text{<\text{<C-pointer>}> Method}

Returns \#t if the second argument is less than the first.

**Signature** \text{<C-pointer-1 C-pointer-2 => boolean}

**Parameters**

- \text{c-pointer-1} – An instance of \text{<C-pointer>}.  
- \text{c-pointer-2} – An instance of \text{<C-pointer>}.  

**Values**

- boolean – An instance of \text{<boolean>}.  

**Discussion**

Returns \#t if the second argument is less than the first. This allows pointer comparison operations to be performed on instances of \text{<C-pointer>}. Note that operations corresponding to C pointer arithmetic are not defined on \text{<C-pointer>}. If pointer arithmetic operations are required, use \text{pointer-value} with an \text{index:} argument.

**See also**

- \text{pointer-value}.

The following functions comprise the conceptual foundation on which the pointer accessing protocol is based. In the signatures of these functions, \text{byte-index} is in terms of address units (typically bytes) and \text{scaled-index} is scaled by the size of the units involved. In the setters, \text{new} is the new value to which the value in the pointed-at location will be set. These functions can be used to deference any general instance of \text{<C-pointer>}.

**C-char-at**

Function

**Signature**

\[
\text{C-char-at} \quad \text{<C-pointer> \ #key \ byte-index \ scaled-index => machine-word}
\]

**C-char-at-setter**

Function

**Signature**

\[
\text{C-char-at-setter} \quad \text{new} \quad \text{<C-pointer> \ #key \ byte-index \ scaled-index => machine-word}
\]
C-signed-char-at

Function
Signature

C-signed-char-at *C-pointer* #key *byte-index* *scaled-index* => *machine-word*

C-signed-char-at-setter

Function
Signature

C-signed-char-at-setter *new* *C-pointer* #key *byte-index* *scaled-index* => *machine-word*

C-unsigned-char-at

Function
Signature

C-unsigned-char-at *C-pointer* #key *byte-index* *scaled-index* => *machine-word*

C-unsigned-char-at-setter

Function
Signature

C-unsigned-char-at-setter *new* *C-pointer* #key *byte-index* *scaled-index* => *machine-word*

C-unsigned-short-at

Function
Signature

C-unsigned-short-at *C-pointer* #key *byte-index* *scaled-index* => *machine-word*

C-unsigned-short-at-setter

Function
Signature

C-unsigned-short-at-setter *new* *C-pointer* #key *byte-index* *scaled-index* => *machine-word*
C-signed-short-at

Function
Signature

\[
\text{C-signed-short-at} \rightarrow \text{C-pointer} \rightarrow \#key \rightarrow \text{byte-index} \rightarrow \text{scaled-index} \rightarrow \text{machine-word}
\]

C-signed-short-at-setter

Function
Signature

\[
\text{C-signed-short-at-setter} \rightarrow \text{new} \rightarrow \text{C-pointer} \rightarrow \#key \rightarrow \text{byte-index} \rightarrow \text{scaled-index} \rightarrow \text{machine-word}
\]

C-short-at

Function
Signature

\[
\text{C-short-at} \rightarrow \text{C-pointer} \rightarrow \#key \rightarrow \text{byte-index} \rightarrow \text{scaled-index} \rightarrow \text{machine-word}
\]

C-short-at-setter

Function
Signature

\[
\text{C-short-at-setter} \rightarrow \text{new} \rightarrow \text{C-pointer} \rightarrow \#key \rightarrow \text{byte-index} \rightarrow \text{scaled-index} \rightarrow \text{machine-word}
\]

C-unsigned-long-at

Function
Signature

\[
\text{C-unsigned-long-at} \rightarrow \text{C-pointer} \rightarrow \#key \rightarrow \text{byte-index} \rightarrow \text{scaled-index} \rightarrow \text{machine-word}
\]

C-unsigned-long-at-setter

Function
Signature

\[
\text{C-unsigned-long-at-setter} \rightarrow \text{new} \rightarrow \text{C-pointer} \rightarrow \#key \rightarrow \text{byte-index} \rightarrow \text{scaled-index} \rightarrow \text{machine-word}
\]
### C-signed-long-at

**Function**

**Signature**

\[
\text{C-signed-long-at } \ast \text{C-pointer} \ast \text{key} \ast \text{byte-index} \ast \text{scaled-index} \Rightarrow \ast \text{machine-word}
\]

### C-signed-long-at-setter

**Function**

**Signature**

\[
\text{C-signed-long-at-setter } \ast \text{new} \ast \text{C-pointer} \ast \text{key} \ast \text{byte-index} \ast \text{scaled-index} \Rightarrow \ast \text{machine-word}
\]

### C-long-at

**Function**

**Signature**

\[
\text{C-long-at } \ast \text{C-pointer} \ast \text{key} \ast \text{byte-index} \ast \text{scaled-index} \Rightarrow \ast \text{machine-word}
\]

### C-long-at-setter

**Function**

**Signature**

\[
\text{C-long-at-setter } \ast \text{new} \ast \text{C-pointer} \ast \text{key} \ast \text{byte-index} \ast \text{scaled-index} \Rightarrow \ast \text{machine-word}
\]

### C-unsigned-int-at

**Function**

**Signature**

\[
\text{C-unsigned-int-at } \ast \text{C-pointer} \ast \text{key} \ast \text{byte-index} \ast \text{scaled-index} \Rightarrow \ast \text{machine-word}
\]

### C-unsigned-int-at-setter

**Function**

**Signature**

\[
\text{C-unsigned-int-at-setter } \ast \text{new} \ast \text{C-pointer} \ast \text{key} \ast \text{byte-index} \ast \text{scaled-index} \Rightarrow \ast \text{machine-word}
\]

---

**3.6. Designator classes**
C-signed-int-at

Function
Signature
```
C-signed-int-at  *C-pointer*  #key  *byte-index*  *scaled-index*  =>  *machine-word*
```

C-signed-int-at-setter

Function
Signature
```
C-signed-int-at-setter  *new*  *C-pointer*  #key  *byte-index*  *scaled-index*  =>  *machine-word*
```

C-int-at

Function
Signature
```
C-int-at  *C-pointer*  #key  *byte-index*  *scaled-index*  =>  *machine-word*
```

C-int-at-setter

Function
Signature
```
C-int-at-setter  *new*  *C-pointer*  #key  *byte-index*  *scaled-index*  =>  *machine-word*
```

C-double-at

Function
Signature
```
C-double-at  *C-pointer*  #key  *byte-index*  *scaled-index*  =>  *float*
```

C-double-at-setter

Function
Signature
```
C-double-at-setter  *new-double-float*  *C-pointer*  #key  *byte-index*  *scaled-index*  =>  *float*
```
C-float-at

Function

Signature

C-float-at *C-pointer* #key *byte-index* *scaled-index* => *float*

C-float-at-setter

Function

Signature

C-float-at-setter *new-single-float* *C-pointer* #key *byte-index* *scaled-index* => *float*

C-pointer-at

Function

Signature

C-pointer-at *C-pointer* #key *byte-index* *scaled-index* => *C-pointer*

C-pointer-at-setter

Function

Signature

C-pointer-at-setter *new* *C-pointer* #key *byte-index* *scaled-index* => *C-pointer*

C-size-t-at

Function

Signature

C-size-t-at *C-pointer* #key *byte-index* *scaled-index* => *machine-word*

C-size-t-at-setter

Function

Signature

C-size-t-at-setter *new* *C-pointer* #key *byte-index* *scaled-index* => *machine-word*
C-ssize-t-at

Function
Signature

C-ssize-t-at *C-pointer* #key *byte-index* *scaled-index* =>
*machine-word*

C-ssize-t-at-setter

Function
Signature

C-ssize-t-at-setter *new* *C-pointer* #key *byte-index* *scaled-index* =>
*machine-word*

3.6.5 Structure types

<C-struct> Open Abstract Class

Discussion
The abstract superclass of all classes designating a C struct type. It is a subclass of <C-value>. It is a subclass of <C-value>. You can describe new struct types using the define C-struct macro.

Classes designating C structs are not instantiable. Where a slot, array element, function parameter or function result is typed as a struct value, pointers to that struct type are accepted and returned.

3.6.6 Union types

<C-union> Open Abstract Class

Discussion
The abstract superclass of all classes designating a C union type. It is a subclass of <C-value>. You can describe new union types with the macro define C-union. Classes designating C unions are not instantiable. Where a slot, array element, function parameter or function result is typed as a union value, pointers to that union type are accepted and returned.

3.6.7 Notes on C type macros

The C-FFI’s C interface description language does not model all of the ways of defining new types in C, but all C types should be expressible in it. As a simplification, we do not support anonymous base types in the C interface description language. If a structure or union field has an in-line type definition in C, that definition must be extracted and given a name in order for it to be used. For example, the following C struct

```c
struct something {
    char *name;
    long flags;
    union {
        long int_val;
    }
}
```

(continues on next page)
can be described with these definitions:

```dylan
define C-union <anonymous-union-1>
    slot int-val :: <C-long>;
    slot string-val :: <C-string>;
end C-union;

define C-struct <anonymous-struct-1>
    slot name :: <C-string>;
    slot flags :: <C-long>;
    slot val :: <anonymous-union-1>;
end C-struct;
```

The slots of these ex-inline types must be accessed through a chain of accesses, for example `o.val.string-val`.

### 3.7 Defining types

This section covers the definition macros that create Dylan designators for C types, structs and unions.

#### 3.7.1 Defining specialized versions of designator classes

**define C-subtype Defining Macro**

Defines a specialized designator class for a C type based on an existing designator class for that type.

**Macro Call**

```dylan
define [*modifiers* [*] C-subtype name (superclasses)
    /*slot-spec* ; ...) [ ]
    /*type-options* ] [ ];
end [C-subtype] [*name* ]
```

**Parameters**

- **modifiers** – The same as the modifiers allowed in `define class`.
- **name** – A Dylan variable name.
- **superclasses** – A list of Dylan names.
- **slot-spec** – Same syntax as a slot definition in `define class`.
- **type-options** – A property list.

**Discussion**

Defines a specialized designator class for a C type based on an existing designator class for that type. It does this by defining a subclass of the original designator class, and is a simple wrapper around `define class` from which it takes its syntax. The superclasses, slot-specs, and `modifiers` are passed on to `define class` unchanged. In effect, it expands to:
In terms of C, define C-subtype can be thought of as implementing a strongly typed version of typedef because a new designator class is generated that Dylan’s type system can distinguish from the designator class on which it was based. As well as inheriting from an existing designator class, other Dylan classes can be mixed in too.

The optional type-options must be a property list. The c-name: keyword is recognized, allowing the original C name of the type designated by the class to be documented. The pointer-type-name: keyword option can be used to name the designator for pointers to name.

Example Some example C declarations:

```c
typedef void *Handle;
typedef Handle WindowHandle;
typedef Handle StreamHandle;

extern WindowHandle CurrentWindow (void);
extern StreamHandle CurrentStream (void);
```

Example FFI definitions:

```dylan
define C-subtype <Handle> (<C-void*>) end;
define C-subtype <WindowHandle> (<Handle>) end;
define C-subtype <StreamHandle> (<Handle>) end;

define C-function CurrentWindow
  result value :: <WindowHandle>;
  c-name: "CurrentWindow";
end C-function;

define C-function CurrentStream
  result value :: <StreamHandle>;
  c-name: "CurrentStream";
end C-function;
```

Example transactions:

```dylan
? <C-void*> == <WindowHandle> | <WindowHandle> == <StreamHandle>;
#f

? define variable *cw* = CurrentWindow();
// Defined *cw*

? *cw*
<WindowHandle> #xff5400

? define variable *cs* = CurrentStream();
// Defined *cs*

? *cs*
<StreamHandle> #xff6400
```

(continues on next page)
The following example uses the ability to specify extra superclasses to place a type beneath an abstract class.

Example C declarations:

```c
struct _Matrix {
    int rank;
    int *dimensions;
    int *values;
};
typedef struct _Matrix *Matrix;
extern Matrix MatrixAdd (Matrix m, Matrix n);
```

Example FFI definitions:

```c
define C-struct <_Matrix-struct> 
    slot rank :: <C-int>;
    slot dimensions :: <C-int*>;
    slot values :: <C-int*>;
    pointer-type-name: <_Matrix-struct*>
end C-struct;

define C-subtype <Matrix> (<_Matrix-struct*>, <number>) end;

define C-function MatrixAdd 
    parameter m :: <Matrix>;
    parameter n :: <Matrix>;
    result value :: <Matrix>;
    c-name: "MatrixAdd"
end C-function;

define method + (m1 :: <Matrix>, m2 :: <Matrix>) =>
    (r :: <Matrix>)
    MatrixAdd(m1, m2)
end method;
```

### 3.7.2 Defining specialized designator classes

**define C-mapped-subtype Defining Macro**

Allows you to define a name to which to bind a pointer designator.

**Macro Call**

```c
define *modifiers* C-mapped-subtype *type-name* (*superclasses*)
    [map *high-level-type*
        [, import-function: *import-fun* ]
        [, export-function: *export-fun* ];]
    [import-map *high-level-type*,
        import-function: *import-fun* ;]
    [export-map *high-level-type*,
        export-function: *export-fun* ;]
```

(continues on next page)
Parameters

- **modifiers** – The same as the modifiers allowed in `define-class`.
- **type-name** – A Dylan variable name.
- **superclasses** – A list of Dylan names.
- **high-level-type** – An instance of a Dylan `<type>`.
- **import-fun** – An instance of `<function>`.
- **export-fun** – An instance of `<function>`.
- **type-options** – A property list.

Discussion

Allows you to define a name to which to bind a pointer designator.

The `modifiers` may be `sealed` or `open`. (The default is `sealed`.) Their effect on the class defined is the same as the similar modifiers on an ordinary class.

The possible combinations are, a map clause, an import-map clause, an export-map clause, or both an import-map and an export-map clause. Any other combinations are illegal.

The `import-map` clause specifies that a type conversion takes place when `type-name` is used as a designator for values imported from C into Dylan. The conversion is accomplished by calling the `import-function` on the imported value. This call is automatically inserted into function wrappers, structure member getters, pointer-value dereference functions and so on by the C-FFI. The `high-level-type` is used as the Dylan type specifier for the appropriate parameter or result in any wrapper function or `c-struct` accessor which uses the defined class. The `export-map` clause specifies a similar type conversion for exported values. The `high-level-type` must in either case name an instantiable Dylan type.

```
map <type-c>;
```

is equivalent to:

```
import-map <type-c>;
export-map <type-c>;
```

The import and export functions are monadic functions whose single argument is the appropriate low-level value for export functions and the appropriate Dylan type for import functions. Any mapped subtype which specifies an import-map must specify an `import-function`. Any mapped subtype which specifies an export-map must specify an `export-function`.

Map boolean example:

```
bool-header.h:

typedef int bool;
bool bool_function (bool b);
void bool_pointer_function (bool * b);
//eof
```
Module: my-module

```dylan
define C-mapped-subtype <bool> (<C-int>)
  map <boolean>,
    export-function:
      method (v :: <boolean>) => (result :: <integer>)
        as(<integer>, if(v) 1 else 0 end if)
      end,
    import-function:
      method (v :: <integer>) => (result :: <boolean>)
        ~zero?(v)
      end;
  end;
//end module
```

Mapped string example: an alternate version of C-string which automatically converts instances of `<byte-string>` to instances of `<C-example-string>` on export.

```dylan
string-header.h
typedef char * string;
string string-filter(string s);
void string-modifier(string * s);
//eof
```

```dylan
module: my-module
define C-mapped-subtype <C-example-string> (<C-char*>, <string>)
  export-map type-union(<byte-string>, <C-example-string>),
    export-function: c-string-exporter;
end;

define method c-string-exporter
  (s :: <byte-string>)
=> (result :: <C-char*>)
  as(<C-example-string>, s)
end;

define method c-string-exporter
  (s :: <C-example-string>)
=> (result :: <C-example-string>)
  s
end;
//end module
```

It is possible to define an ordinary subtype of a mapped supertype. The mapping characteristic of the subtype is inherited from the supertype. It is also possible to define a mapped subtype of a mapped supertype. When the subtype and supertype both specify an export function, the export functions of the subtype and the supertype are composed with the subtype’s export function applied to the result of the supertype’s export function. Import functions of a mapped subtype and supertype are similarly composed. Mapping characteristics are inherited from the supertype where the subtype does not define them. (You can think of this as composition with identity when either the supertype or subtype fails to specify an import or export function.) This shadowing is
only useful when import and export maps are defined separately. Here is an example of a mapped subtypes which adds an import map to the mapped version of `<C-example-string>` defined above.

```dylan
define C-mapped-subtype <other-string>  
   (<C-example-string>)  
import-map <byte-string>,  
import-function:  
   method (v :: <byte-string>)  
    => (result :: <C-example-string>)  
    as(<C-example-string>, v)  
end method;  
end;
```

The import signature is `<byte-string>`. The export signature is inherited from `<C-example-string>` `type-union(<byte-string>, <C-example-string>)`. For an example involving composition of mapped types consider the following (hypothetical) definitions of `<C-raw-int>`, `<C-mapped-int>` and `<bool>`. The `<C-raw-int>` class is a primitive which returns and accepts instances of `<machine-word>`. The `<C-mapped-int>` class is a mapped subtype which converts the instances of `<machine-word>` to instances of `<integer>`. The `<bool>` class is a mapped subtype of `<C-mapped-int>` which converts to and from `<boolean>`.

```dylan
define C-mapped-subtype <C-mapped-int> (<C-raw-int>)  
map <boolean>,  
export-function:  
   method (v :: <integer>) => (result :: <machine-word>)  
    as(<machine-word>, v)  
end,  
import-function:  
   method (v :: <machine-word>) => (result :: <integer>)  
    as<integer>, v)  
end;  
end;
```

```dylan
define C-mapped-subtype <bool> (<C-mapped-int>)  
map <boolean>,  
export-function:  
   method (v :: <boolean>) => (result :: <integer>)  
    if(v) 1 else 0 end if  
end,  
import-function:  
   method (v :: <integer>) => (result :: <boolean>)  
    ~zero?(v)  
end;  
end;
```

### 3.7.3 Describing structure types

**define C-struct Defining Macro**

Describes C’s aggregate structures.

**Macro Call**

```dylan
define C-struct *name*  
   [*slot-spec* ; ...] [;]
```
Parameters

- **name** – A Dylan variable name.
- **slot-spec** –
  - **type-options** – A property list.

Discussion

Describes C’s aggregate structures. The name is defined to be a designator class encapsulating the value of a structure, not a pointer to the structure. This is significant because many of the protocols associated with structures work only on pointers to structures — pointers to structures being the most common form and the form closest to Dylan’s object model. The new designator class is defined to be a subclass of `<C-struct>`.

Once defined, a structure-designating class is most likely to be used as the basis for a pointer type definition in terms of which most further transactions will take place. Structure-designating classes are abstract and cannot have direct instances. Accessor methods defined for the slots of the structure are specialized on the structure designator’s pointer-type. However, the class itself may be needed to specify an in-line structure in another structure, union, or array, or a value-passed structure argument or result in a C function.

A slot-spec has the following syntax:

```
[<slot-adjective>] slot *getter-name* :: *c-type* #key *setter*
  *address-getter* *c-name length* *width*
```

The *slot-adjective* can be **constant**, **array** or **bitfield**. The **array** slot adjective indicates that the slot is repeated and the **dimensions** option is used to indicate how many repetitions are defined, and how it is accessed. The **bitfield** slot adjective indicates that the slot is really a bitfield. If **bitfield** is given then the **width** option must also be given. The **c-type** given for a **bitfield** slot must be an integer designator. The **c-type** for a **bitfield** slot indicates how the value is interpreted in Dylan by the slot accessor. A slot may not be specified as both an **array** and a **bitfield**. If **constant** is specified, then no setter is generated. The **constant** adjective can be supplied for **array** and **bitfield** slots.

The **getter-name** keyword specifies the name of the Dylan function to which the getter method for the structure slot will be added. The specializer of the getter method’s single argument will be a designator indicating a pointer to the struct’s name.

The **c-type** specifies the field’s C type, and must be a designator class. Unlike Dylan slot specifications, the type declaration here is not optional.

The optional **setter** keyword specifies the generic function to which the setter method for the structure slot will be added. It defaults to **getter-name**'-setter'. No setter method is defined if the **setter** option is `<f>`. If the **constant** keyword is supplied, no **setter** option should be supplied.

The optional **address-getter** specifies the name of a function that can be used to return a pointer to the data in the member. It must return a `<C-pointer>` object that points to a C type. No **address-getter** is defined by default.

You can use the **dimensions** keyword only if you used the **array** slot adjective. This **dimensions** value can be either a list of integers or a single integer. The accessor for an array slot is defined to take an extra integer parameter for each dimension given.

You can use the **width** keyword option only if you used the **bitfield** adjective.
The optional c-name keyword allows you to document the original C name of the slot.

The type-options clause is a property list allowing you to specify properties of the type as a whole. It accepts the optional keyword c-name:, allowing you to document the original C name of the struct to be documented. The optional keyword pointer-type-name: is also accepted. When given, the name is bound to the struct pointer type on which the accessors are defined.

The type option pack: n indicates that the struct has the packing semantics of Microsoft’s #pragma pack(*n*).

Example  Example C declaration:

```c
struct Point {
    unsigned short x;
    unsigned short y;
};
Point *OnePoint(); /* Returns a pointer to a Point */
Point *PointArray(); /* Returns a Point array */
```

Example FFI definition:

```c
define C-struct <Point>
    slot x :: <C-unsigned-short>;
    slot y :: <C-unsigned-short>;
    pointer-type-name: <Point*>;
end C-struct;

define C-function one-point
    result point :: <Point*>;
    c-name: "OnePoint";
end C-function;

define C-function point-array
    result array :: <Point*>;
    c-name: "PointArray";
end C-function;
```

Example transactions:

```c
? define variable p = one-point();
// Defined p.

? values(p.x, p.y);
100
50

? define variable array = point-array();
// Defined array.

? array[5].object-class; // implicit conversion to
// the pointer type
{<Point> pointer #xff5e00}

? begin array[5].x := 10; array[5].y := 20 end;
20

? values(array[5].x, array[5].y)
```

(continues on next page)
3.7.4 Describing union types

**define C-union Defining Macro**

Describes C union types to the c-ffi.

**Macro Call**

```
define C-union *name* [*slot-spec* ; ...
] [;
[ *type-options* ] [;
end [C-union] [ *name* ]
```

**Parameters**

- **name** – A Dylan variable name.
- **slot-spec** –
- **type-options** – A property list.

**Discussion**

Describes C union types to the C-FFI. The syntax for the macro and its use are similar to `define c-struct` except that bitfield slots are not allowed. The designator created by the macro is a subclass of `<c-union>`.

Each of the slots in a union is laid out in memory on top of one another just as in C’s `union` construct.

**Example**

Example C declaration:

```c
union Num {
    int int_value;
    double double_value;
};
Num *OneNum(); /* Returns a pointer to a Num */
Num *NumArray(); /* Returns a Num array */
```

Example FFI definition:

```dylan
define C-union <Num>
    slot int-value :: <C-int>;
    slot double-value :: <C-double>;
    pointer-type-name : <Num*>;
end C-union;

define C-function one-num
    result num :: <Num*>;
    c-name : "OneNum";
end C-function;
define C-function num-array
```

(continues on next page)
result array :: <Num*>;  
c-name: "NumArray";  
end C-function;

Example transactions:

? define variable n = one-num();  
// Defined n.

? values(p.int-value, p.double-value);  
154541  
92832.e23 // or something

? define variable array = num-array();  
// Defined array.

? array[5].object-class; // implicit conversion to  
// the pointer type  
<{Num> pointer #xff5e00}

? array[5].int-value := 0;  
0

? array[5].double-value;  
11232e-12 // or a different something

### 3.8 Functions

This section describes the C FFI macros that allow C functions to be made available to Dylan and Dylan functions available to C.

#### 3.8.1 Function types

This section describes classes that designate C function types and how to construct them.

<C-function-pointer> Open Abstract Class

Discussion The superclass of all classes that designate a C function type. It is a subclass of <C-pointer>. The Dylan variable bound by define c-callable is of this type.

#### 3.8.2 Describing C functions to Dylan

define C-function Defining Macro

Describes a C function to the c-ffi.

Macro Call

```
define C-function *name*  
[*parameter-spec*; ...]  
[*result-spec*;]  
[*function-option*, ...;]  
end [C-function] [*name*]
```
Parameters

- **name** – A Dylan variable name.
- **parameter-spec** –
- **result-spec** –
- **function-option** – A property list.

Discussion

Describes a C function to the C-FFI. In order for a C function to be called correctly by Dylan, the same information about the function must be given as is needed by C callers, typically provided by `extern` declarations for the function in a C header file: the function’s name and the types of its parameters and results.

The result of processing a `define C-function` definition is a Dylan function which is bound to `name`. This function takes Dylan objects as arguments, converting them to their C representations according to the types declared for the parameters of the C function before calling the C function with them. If the C function returns results, these results are converted to Dylan representations according to the declared types of those results before being returned to the Dylan caller of the function. By default the function created is a raw method, not a generic function. A generic function method can be defined by using the `generic-function-method: option`.

Either the `c-name:` function option must be supplied, or the `indirect:` option must be supplied with a value other than `#f`, but not both.

A parameter-spec has the following syntax:

```
[*adjectives*] parameter name :: *c-type* #key *c-name*
```

If no parameters are specified, the C function is taken to have no arguments.

The adjectives can be either `output`, `input`, or both. The calling discipline is specified by the `input` and `output` adjectives.

By itself, `input` indicates that the argument is passed into the function by value. This option is the default and is used primarily to document the code. There is a parameter to the generated Dylan function corresponding to each `input` parameter of the C function.

The `output` adjective specifies that the argument value to the C function is used to identify a location into which an extra result of the C function will be stored. There is no parameter in the generated Dylan function corresponding to an `output` parameter of the C function. The C-FFI generates a location for the extra return value itself and passes it to the C function. When the C function returns, the value in the location is accessed and returned as an extra result from the Dylan function. The C-FFI allocates space for the output parameter’s referenced type, passes a pointer to the allocated space, and returns `pointer-value` of that pointer. A struct or union type may not be used as an output parameter.

Example of `output` parameter definition:

```
define C-function mix-it-up
  output parameter out1 :: <some-struct*>
  output parameter out2 :: <C-int*>
  result value :: <C-int>
  c-name: "mix_it_up"
end C-function mix-it-up;
```

Example transaction:
If both input and output are supplied, they specify that the argument value to the C function is used to identify a location from which a value is accessed and into which an extra result value is placed by the C function. There is a parameter to the generated Dylan function corresponding to each input output parameter of the C function that is specialized as the union of the export type of the referenced type of the type given for the parameter in define c-function, and #f.

When the C function returns, the value in the location is accessed and returned as an extra result from the Dylan function. If an input output parameter is passed as #f from Dylan then a NULL pointer is passed to the C function, and the extra value returned by the Dylan function will be #f.

Example of input output parameter definition:

```
define C-function mix-it-up
    input output parameter inout :: <C-int*>;
    result value :: <C-int>;
    c-name: "mix_it_up";
end C-function mix-it-up;
```

Example transaction:

```
? mix-it-up(7);
1
14
```

Note that neither output nor input output affects the declared type of an argument: it must have the same type it has in C and so, because it represents a location, must be a pointer type.

A result-spec has the following syntax:

```
result [name :: c-type]
error-result [name :: c-type]
```

If no result is specified, the Dylan function does not return a value for the C result, and the C function is expected to have a return type of void.

error-result is used when it is necessary to call the import-map function on the result and then discard it. This is often used when mapping a return value to a Dylan error.

Each function-option is a keyword–value pair. The generic-function-method: option may be either #t or #f, indicating whether to add a method to the generic function name or to bind a bare constant method directly to name. The default value for generic-function-method: is #f.

The option C-modifiers: can be used to specify platform dependent modifiers for the C function being called. For example, on Windows, use C-modifiers: "__stdcall" if the C function to be called is defined to be a __stdcall function.

The c-name: option is used to specify the name of the C function as it is defined in the object or shared library file. The c-name must be a constant string.

The indirect: #t option defines a function that accepts a C function pointer as its first argument and calls the function given with the signature described by the parameters and result given. In this case the Dylan function defined accepts one more argument than if c-name was given. The type specified for the first parameter of the Dylan function is <c-function-pointer>. One of c-name or indirect: #t must be supplied, but not both.
Example C declarations:

```c
/* Compute the length of a string */
int strlen(char *string);

/* Set the given locations to values, returning an error code */
int fill_locations(int *loc1, int *loc2);

/* Read at most as far as indicated in max_then_read, updating it to contain how much was actually read */
void read_stuff(int *max_then_read);
```

Example FFI definitions:

```c
define C-function strlen
  parameter string :: <C-char*>;
  result value :: <C-int>;
  c-name: "strlen";
end C-function;

define C-function fill-locations
  output parameter loc1 :: <C-int*>;
  output parameter loc2 :: <C-int*>;
  result return-code :: <C-int>;
  c-name: "fill_locations";
end C-function;

define C-function read-stuff
  input output parameter :: <C-int*>;
  c-name: "read_stuff";
end C-function;
```

Example transactions:

```c
? strlen($my-c-string);
44
? fill-locations();
0
101 // extra output value
102 // extra output value
? read-stuff(100);
50 // extra output value
```

In effect, a define C-function such as:

```c
define C-function foo
  parameter string :: <C-char*>;
  parameter count :: <C-int>;
  result value :: <C-int>;
  c-name: "foo";
end C-function;
```

expands into something like:

```c
define function foo (string, count)
  let c-string = %as-c-representation(<C-char*>, string);
```

(continues on next page)
let c-count = %as-c-representation(<C-int>, count);
let c-result = %call-c-function("foo", c-string,
    c-count);
%as-dylan-representation(<C-int>, c-result);
end;

with the declared type.

3.8.3 Describing Dylan functions for use by C

**define C-callable-wrapper Defining Macro**

Makes a Dylan function callable from C by describing a C contract for the function.

**Macro Call**

```
define C-callable-wrapper ["*dylan-rep-name*"
of "*dylan-function*
["*parameter-spec* ; ..."] [;]
["*result-spec* ] [;]
["*function-options* ]];
end [C-callable-wrapper]
```

**Parameters**

- **dylan-rep-name** – A Dylan variable name.
- **dylan-function** – An instance of `<function>`.
- **parameter-spec** –
- **result-spec** –
- **function-options** – A property list.

**Discussion**

Makes a Dylan function callable from C by describing a C contract for the function. In order to generate a correct C-callable function wrapper, the same information about the function must be given as would be needed by C callers, typically provided by `extern` declarations for the function in a C header file: the types of its parameters and results.

The result of processing a `define C-callable-wrapper` definition is a function with a C entry point with the contract described. This function takes C values as arguments, converting them to Dylan representations according to the types declared for the parameters of the C function before calling the Dylan function with them. If the C function was described as returning results, the results of the call to the Dylan function are converted to C representations according to the declared types of those results before being returned to the C caller of the function.

The `dylan-function` is a Dylan function that accepts the correct number of parameters, and is called by the C callable wrapper.

The function-options are a property list. This list may contain a string value for the c-name keyword. If a c-name is specified, that name is made visible to C as the name of the generated *C-callable wrapper* function. Given a compatible `extern` declaration, this allows C code to call Dylan code simply by invoking a named function. The `export:` option takes the values `#t` or `#f` and indicates whether the c-name for the generated *C-callable-wrapper* function is to be exported from the library’s *dll*. `#t` means it is exported, `#f` means it is not. The default is `#f`. 
The \textit{c-modifiers:} option is the same as in the \textit{c-function} macro, except that the modifiers apply to the C function wrapper which is generated. See \textit{define C-function}.

If \textit{dylan-rep-name} is specified, it is bound to an instance of a function-pointer designator class identifying the generated C-callable wrapper function. You can pass this pointer to C code for use as, for example, a callback.

A parameter-spec has the following syntax:

\begin{verbatim}
[\{\textbf{adjectives}\} \textbf{parameter name} :: \textbf{c-type} \textbf{key} \textbf{c-name}]
\end{verbatim}

If no parameters are specified, the C function is taken to have no arguments.

An adjective can be \textit{input}, \textit{output}, or both. The calling discipline is specified by the \textit{input} and \textit{output} adjectives.

If a parameter is \textit{output}, the corresponding parameter is not passed to the Dylan function, but the Dylan function is expected to return an extra value that is placed in the location pointed to by the parameter. When the pointer is NULL, the extra value from the Dylan function is ignored. The type designated for the parameter must be a pointer type.

If a parameter is both \textit{input} and \textit{output}, the parameter must be a pointer type, and the value accepted by the Dylan function is the result. The functions pointer-value and pointer-value-setter perform the primitive Dylan-to-C and C-to-Dylan conversions as documented with the designator class of the pointer’s contents type (see Table 1.1). The C-FFI signals an error if it cannot convert the object you attempt to store in the pointer to a compatible type on that pointer. The Dylan function is expected to return an extra value which is placed into the location specified by the pointer passed to the C function. If the pointer passed to the C function is NULL, then the value passed to the Dylan function will be \texttt{#f}, and the extra value returned will be ignored.

There is currently no way to define a C-callable function that accepts a variable number of arguments.

A result-spec has the following syntax:

\begin{verbatim}
\textbf{result} \textbf{name} :: \textbf{c-type}
\end{verbatim}

If no \textbf{result} is specified, the C function defined does not return a value. It is defined as what in C terminology is known as a \textit{void} function.

\textbf{Example}  
Example C declarations:

\begin{verbatim}
/* Compute the length of a string */
int strlen(char *string);

/* Set the given locations to values, returning an error code */
int fill_locations(int *loc1, int * loc2);

/* Read at most as far as indicated in max_then_read, updating it to contain how much was actually read */
void read_stuff(int *max_then_read);
\end{verbatim}

Example FFI definitions:

\begin{verbatim}
define method dylan-strlen (string) => (length) ... end;
define C-callable-wrapper of dylan-strlen 
  parameter string :: <C-char*>;
\end{verbatim}

(continues on next page)
result value :: <C-int>;
c-name: "strlen";
end C-function;

define method dylan-fill-locations () => (return-code :: <integer>,
  val1 :: <integer>,
  val2 :: <integer>)
  ...
end;

define C-callable-wrapper of dylan-fill-locations
  output parameter loc1 :: <C-int*>;
  output parameter loc2 :: <C-int*>;
  result return-code :: <C-int>;
  c-name: "fill_locations";
end C-function;

define method dylan-read-stuff (max :: <integer>) => (read :: <integer>) ...
end;

define C-callable-wrapper of dylan-read-stuff
  input output parameter max-then-read :: <C-int*>;
  c-name: "read_stuff";
end C-function;

Example C calls:

{ int length, *loc1, *loc2, max_then_read;
  length = strlen("ABC");
  fill_locations(loc1, loc2);
  max_then_read = 100
  read_stuff(&max_then_read);
}

In effect, a define C-callable-wrapper such as:

define C-callable-wrapper of foo
  parameter string :: <C-char*>;
  parameter count :: <C-int>;
  result value :: <C-int>;
  c-name: "foo";
end C-function;

expands into something like:

%c-callable-function "foo" (c-string, c-count)
let dylan-string
  = %as-dylan-representation(<C-char*>, c-string);
let dylan-count
  = %as-dylan-representation(<C-int>, c-count);
let dylan-result
  = foo(dylan-string, dylan-count);
where the % functions perform the primitive conversions between Dylan and C representations, checking that their arguments are compatible with the declared type.

Callback example:

```dylan
define C-function c-sort
   parameter strings :: <C-string*>
   parameter compare :: <C-function-pointer>
   result sorted-strings :: <C-string*>
   c-name: "sort"
end C-function;

define C-callable-wrapper callback-for-< of <
   parameter string1 :: <C-string>
   parameter string2 :: <C-string>
   result int :: <C-int>
end C-callable-wrapper;

? callback-for-<
{function pointer #xff6e00}

? c-sort(some-c-strings, callback-for-<);
{<C-string> array}
```

### 3.8.4 Objective C

A full-featured Objective C bridge is provided separately, however, parts of that bridge are implemented within this library.

**define objc-selector Defining Macro**

Describe Objective C selectors to the c-ffi.

#### Macro Call

```dylan
define objc-selector *name*
   [*parameter-spec*; ...]
   [*result-spec*;]
   [*function-option*, ...;]
end [C-function] [*name*]
```

#### Parameters

- **name** – A Dylan variable name.
- **parameter-spec** –
- **result-spec** –
- **function-option** – A property list.

#### Discussion

Defining an Objective C selector is much the same as define C-function, except:

- There must be at least one parameter specification. The first parameter specifies the target of the method, so it should be either an Objective C class or an object instance.
• Rather than specifying a c-name: for the function, a selector is specified instead.

• The c-modifiers keyword can be used to select alternate versions of objc_msgSend when calling into the Objective C run-time.

• An additional keyword, type-encoding: must be supplied with a valid type encoding for the selector. See the Objective C bridge documentation for more details.

Example

```dylan
define objc-selector sel/alloc
  parameter target :: <objc/class>
  result objc-instance :: <objc/instance-address>
  selector: "alloc"
end;
```

3.8.5 Variables

This section covers describing and accessing C variables.

**define C-variable Defining Macro**

Describes C variables to the c-ffi.

**Macro Call**

```dylan
define C-variable *getter-name* :: *c-type*
  #key *setter* *c-name* import: *boolean*
end [C-variable]
```

**Parameters**

• **getter-name** – A Dylan variable name.

• **c-type** – A Dylan name.

• **setter** – #f or a Dylan variable name.

• **c-name** – A string constant.

• **import** – #f or #t.

**Discussion**

Describes C variables to the C-FFI. It defines a getter and setter function for accessing the variable’s value. The c-name keyword argument is required and gives the C name of the variable to be accessed. The setter keyword allows you to specify the name of the setter function, or if a setter function is to be defined at all. If setter is #f, no setter function will be defined.

The import: option indicates if the C variable must be imported from another .dll or not. #t indicates it is in another .dll and must be imported, #f means that it is not to be imported. Whether the variable has to be imported from another .dll or not is determined by which Dylan project the C source files are part of. If they are in the same project as the C-variable definition then the value of “import:” should be #f as the definition and variable will be linked into the same .dll. If the definition is in a different project from the C source files then they will be in separate .dlls and import: needs to be #t. The default value is #f.

For integer, float, or pointer-typed C variables the representation is clear and unambiguous. For C struct or union typed variables the translation is not so simple. A C union or struct has no direct representation in Dylan. You may only have a reference to the C object in Dylan through a <c-pointer> object. For this reason, define c-variable is not permitted for variables with C aggregate types. Use define C-address for those variables.
Example

```dylan
define C-variable process-count :: <C-int>,
c-name: "process_count" end;

process-count();
57

process-count() := 0;
0

process-count();
0

define C-variable machine-name-1 :: <C-char*>,
c-name: "MachineName";
end;

machine-name-1();
#{<C-char*> #xaaabc00}
```

In C and other static languages what is known as a variable is a named allocation of memory. To access a global C variable from Dylan it is occasionally necessary to get a handle to the location where that variable is kept. The `define C-address` macro can be used for this purpose.

**define C-address Defining Macro**

Defines a Dylan constant binding that is a `<C-pointer>` to the location of a C global variable.

**Macro Call**

```dylan
define C-address *name* :: *pointer-designator-type* 
    #key *c-name* import: *boolean* 
end [C-address] [*name* ]
```

**Parameters**

- `name` – A Dylan variable name.
- `pointer-designator-type` –
- `c-name` – A string constant.
- `import` – `#f` or `#t`.

**Discussion**

Defines a Dylan constant binding, `name`, that is a `<C-pointer>` which points to the location of the C global variable `c-name`.

`Pointer-designator-type` must be the type of the constant to be defined, and a subtype of `<C-pointer>`.

The `import:` option indicates if the C address must be imported from another `.dll` or not. `#t` indicates it is in another `.dll` and must be imported, `#f` means that it is not to be imported. Whether the variable has to be imported from another `.dll` or not is determined by which Dylan project the C source files are part of. If they are in the same project as the `C-address` definition then the value of “import:” should be `#f` as the definition and variable will be linked into the same `.dll`. If the definition is in a different project from the C source files then they will be in separate `.dll`s and `import:` needs to be `#t`. The default value is `#f`. 

3.8. Functions 83
3.9 Allocating and deallocating C storage

C objects can be allocated by calling \texttt{make} on an associated wrapper class or by allocating them on the stack using the macro \texttt{with-stack-structure}.

The C component of a \texttt{make} -allocated object is not deallocated by default when the Dylan designator object is reclaimed by the garbage collector, so we provide a manual means of freeing this storage with the function \texttt{destroy}.

\texttt{make(subclass(\texttt{<C-pointer>}) \texttt{)}} Method

Allocates a C object on the heap.

\begin{verbatim}
Signature  make subclass(<c-pointer>) #key allocator element-count extra-bytes address => C-pointer

Parameters

  - subclass – A subclass of \texttt{<C-pointer>}.  
  - allocator (#key) – An instance of \texttt{<function>}.  
  - element-count (#key) – An instance of \texttt{<integer>}.  
  - extra-bytes (#key) – An instance of \texttt{<integer>}.  
  - address (#key) – An instance of \texttt{<integer>} or \texttt{<machine-word>}.  

Values

  - c-pointer – An instance of type \texttt{<c-pointer>} pointing to the object.

Discussion

Allocates a C object on the heap, using whatever standard C allocation function is in use on the target platform (typically \texttt{malloc}) to allocate the storage. This method is applicable to subclasses of \texttt{<C-pointer>} and returns an instance of its argument class.

If the address option is provided, no new storage is allocated, but instead, a new pointer with the given machine word address is returned.

The \texttt{allocator} argument should be a Dylan function that can serve as an allocator. It must accept a single integer argument — the number of bytes to be allocated — and return a Dylan \texttt{<machine-word>} that represents the address of the memory it allocated.

The amount of storage allocated by default is the result of:

\begin{verbatim}
size-of(*pointer-wrapper-class*.referenced-type)
\end{verbatim}

If a positive integer is passed as an extra-bytes option, that number of extra bytes is also allocated.

If a positive integer is passed as an element-count option, space for element-count copies of the referenced type is allocated, taking into account the extra-bytes option for each of them. The element-count argument can be used for allocating arrays of sizes that are not known statically. The keyword element-count is used for this option rather than size in order to avoid conflict with the size collection keyword. The logical size of a collection represented by a pointer wrapper and the number of array elements that implement it may differ; a null-terminated string is an example of such a case.

This \texttt{make} method calls \texttt{initialize} on the wrapper object it generates before returning it.
```dylan
#define variable *space-for-one-int* = make(<C-int*>);

*space-for-one-int*[0];
97386437634 // Could have been anything unless the
// default allocator guarantees to zero new memory.

*space-for-one-int*[0] := 0;
0

*space-for-one-int*[0];
0

#define variable *space-for-ten-ints* = make(<C-int*>, element-count: 10);

#define C-struct <Z-properties>
  slot type :: <C-int>;
  array slot properties :: <C-int>,
end C-struct <Z-properties>;

#define variable *props* = make(<Z-properties>,
  extra-bytes: 10 * size-of(<C-int>));
```

destroy Open Generic function
Frees the allocated heap memory at a specified address.

Signature  
destroy C-pointer #key de-allocator => ()

Parameters
• c-pointer – An instance of <C-pointer>.
• de-allocator (#key) – An instance of <function>.

Discussion
Frees the allocated heap memory at the address encapsulated in C-pointer.
The deallocator argument should be a Dylan function that can serve as a deallocation facility. It must accept an address as a <machine-word> and free the storage allocated at that address.

You should only use destroy on pointers allocated using make where no address was given. If allocator was passed to make, the matching deallocator should be passed to destroy.

There is a default method for destroy on <C-statically-typed-pointer>.

with-stack-structure Statement Macro
Allocates an object within the scope of the body of the code.

Macro Call

```dylan
with-stack-structure (*name* :: *wrapper-type*
  #key *element-count* *extra-bytes*)
*body*
end [with-stack-structure]
```

Parameters
• name – A Dylan variable name.
• wrapper-type – A Dylan name.
• \texttt{element-count} (#key) – An instance of \texttt{<integer>}.

• \texttt{extra-bytes} (#key) – An instance of \texttt{<integer>}.

**Discussion**

Allocates an object \texttt{name} within the scope of a \texttt{body}. The \texttt{element-count} and \texttt{extra-bytes} options behave as in \texttt{make}. The memory that was allocated is freed after \texttt{body} exits.

This macro gives the object \textit{dynamic extent}.

**Example**

```dylan
define C-struct <PointStruct>
    slot x-coord :: <C-unsigned-short>;
    slot y-coord :: <C-unsigned-short>;
    pointer-type-name: <PointStruct>
end C-struct;

define constant <Point> = <PointStruct*>;

define C-function PlotPoint
    parameter point :: <Point>;
    c-name: "PlotPoint";
end C-function;

define method plot (x, y)
    with-stack-structure (point :: <Point>)
        point.x-coord := 20;
        point.y-coord := 30;
        PlotPoint(point);
end;
end;
```

### 3.10 Utility designator classes

The following designator classes are defined for convenience purposes using \texttt{define c-mapped-subtype}.

**<C-boolean> Open Abstract Class**

**Discussion** A mapped subclass of \texttt{<C-int>} that provides an analogue to Dylan’s \texttt{<boolean>} class. The Dylan type for both import and export is \texttt{<boolean>}, and the C type is \texttt{int}. The C integer 0 is mapped to \texttt{#f} in Dylan, and all other values are mapped to \texttt{#t}.

**<C-string> Open Abstract Class**

**Discussion**

A mapped subclass of \texttt{<C-char*>} and \texttt{<string>}. On export the Dylan types \texttt{<C-string>}, \texttt{or <byte-string>} may be passed to C. On import all values are mapped to \texttt{<C-string>}. A \texttt{<byte-string>} may be passed to C directly and no copying takes place. The value in C will be a pointer to the data of the byte-string. The implementation of \texttt{<byte-string>} is such that, unless there are NULL characters embedded in the string, \texttt{strlen} in C and \texttt{size} in Dylan will return the same value.

A \texttt{<byte-string>} may only be safely passed to a C function if its value is never stored and used after the call returns.

**<C-character> Open Abstract Class**
Discussion  The Dylan type for import and export is `<character>`. It is a designator that allows instances of `<character>` to be passed to and from C.

**with-c-string Statement Macro**

Passes a C pointer to the contents of a `<byte-string>`.

**Macro Call**

```dylan
with-c-string (*variable* = *string-valued-expression*)
  *body*
end
```

**Parameters**

- `variable` – A Dylan variable name.
- `string-valued-expression` – An instance of `<string>`.

**Discussion**

Use this macro when you need to pass C a pointer to the contents of a `<byte-string>`, but for some reason it cannot be passed directly. Inside the `body`, `variable` is bound to a `<C-string>` object that refers to the contents of the string returned by `string-valued-expression`.

**Note:** The `<c-string>` object is only live during the period that `body` is executing. If the program holds onto the pointer after that, the data it refers to cannot be guaranteed to be correct, because the garbage collector can no longer keep track of it.

**clear-memory! Function**

Stores zeros in the specified bytes of memory.

**Signature**

`clear-memory! pointer, size => ()`

**Parameters**

- `pointer` – An instance of type `<C-pointer>` that points to the memory location at which to start writing zeros.
- `size` – An instance of type `<integer>`. The number of bytes to clear.

**Discussion**

Stores zeros into `size` bytes of memory beginning at `pointer`. The space is assumed to be a whole number of words and word-aligned.

**copy-bytes! Function**

Copies an arbitrary number of bytes at an arbitrary alignment.

**Signature**

`copy-bytes! destination-pointer, source-pointer, size => ()`

**Parameters**

- `destination-pointer` – An instance of type `<C-pointer>`.
- `source-pointer` – An instance of type `<C-pointer>`.
- `size` – An instance of `<integer>`.

**Discussion**

Copies an arbitrary number of bytes at arbitrary alignment instead of copying whole words.

**See also**

- `copy-into!`.
copy-into! Function
Copies the specified number of words.

Signature  
    copy-into! destination-pointer, source-pointer, size) => ()

Parameters
• destination-pointer – An instance of type <C-pointer>.
• source-pointer – An instance of type <C-pointer>.
• size – An instance of <integer>.

Discussion
Copies size bytes from source-pointer to destination-pointer.
Although the size is specified in bytes, it will be assumed to be a multiple of the word size. The function may also assume that both pointers are word-aligned and that the two storage areas do not overlap.

See also
• copy-bytes!.

equal-memory? Function
Returns #t if the size of the two designated memory spaces have the same contents.

Signature  
    equal-memory? ptr1, ptr2, size => <boolean>

Parameters
• ptr1 – An instance of type <C-pointer>.
• ptr2 – An instance of type <C-pointer>.
• size – An instance of <integer>.

Discussion Returns #t if the size bytes of memory starting at pointer ptr1 have the same contents as the memory starting at ptr2, else #f. The space is assumed to be a whole number of words and word-aligned.

<C-Dylan-object> Open Abstract Class

Superclasses <C-void>+

Discussion
A mapped subclass of <C-void>+. Objects of this type correspond to specific Dylan objects. The Dylan type for import and export is <C-Dylan-Object>. The C type is void*.

To pass a reference to an arbitrary Dylan object to C, the Dylan object first must be registered using register-C-Dylan-object. Then a <C-Dylan-object> handle to the object can be created using the function export-C-Dylan-object. The handle can then be passed directly to any C transition point designated as <C-Dylan-object>. Any object received by Dylan from a transition point designated as <C-Dylan-object> may be passed to import-C-Dylan-object to get the Dylan object for which it was a handle.

register-C-Dylan-object Function
Allows objects to be passed to a C function as instances of <C-Dylan-object>.

Signature  
    register-C-Dylan-object object

Parameters
• object – An instance of <object>.
Discussion

Allows objects to be passed to a C function as instances of `<C-Dylan-object>`.

The `register-C-Dylan-object` function arranges for the garbage collector to leave the storage used by `object` unclaimed, and assures that the handle passed to C is not accidentally corrupted (from C’s point of view) by the memory manager.

See also

- `unregister-C-Dylan-object`.

`unregister-C-Dylan-object` Function

Deallocates an object.

**Signature** unregister-C-Dylan-object `object`

**Parameters**

- `object` – An instance of `<object>`.

**Discussion** Deallocates an object. When the handle is no longer needed from C, you call `unregister-C-Dylan-object` to allow the object to be normally reclaimed by the memory manager. Calls to `register-C-Dylan-object` and `unregister-C-Dylan-object` on the same object nest or interleave without interference. That is, if `register-C-Dylan-object` is called exactly twice on an object then `unregister-C-Dylan-object` must be called exactly twice before the memory manager can reclaim the space for the object as it normally would.

`export-C-Dylan-object` Function

Fetches the `<C-Dylan-object>` handle for a Dylan object.

**Signature** export-C-Dylan-object `object` => `c-dylan-object`

**Parameters**

- `object` – An instance of `<C-Dylan-object>`.

**Discussion** Fetches the `<C-Dylan-object>` handle for a Dylan object.

`import-C-Dylan-object` Function

Fetches the Dylan object for a `<C-Dylan-object>` handle.

**Signature** import-c-dylan-object `c-dylan-object` => `object`

**Parameters**

- `object` – An instance of `<C-Dylan-object>`.

**Values**

- `object` – An instance of `<object>`.

**Discussion** Fetches the Dylan object for a `<C-Dylan-object>` handle.
4.1 The bit-set Module

<bit-set> Primary Class

Superclasses <set>

Init-Keywords

- all-members-from – If this is a non-negative integer then the set created will be infinite. All integers greater than or equal to the one supplied will be members of the set. The default is #f.
- member-vector –
- members – If supplied, this gives the initial elements of the set as a sequence of integers.
- pad –
- upper-bound-hint – An integer which indicates that all the elements of the set are expected to below this value. This is merely an aid to the implementation when allocating the set, and integers which are greater than or equal to this number can be added at any time. The default is zero.

Discussion Represents finite sets and some infinite sets over the non-negative integers in an efficient manner using a <bit-vector>. The infinite sets which can be represented are those which are the complement of a finite set.

copy-bit-set! Function

Signature copy-bit-set! set1 set2 => ()

Parameters

- set1 – An instance of <bit-set>.
- set2 – An instance of <bit-set>.

Discussion Destructively modifies set1 so that it contains exactly the same elements as set2. After the copy, set1 and set2 do not share any structure.

empty-bit-set! Function

Signature empty-bit-set! set => ()

Parameters

- set – An instance of <bit-set>.

Discussion Destructively modifies set by removing all its elements.
### infinite? Sealed Generic function

**Signature**

\[
\text{infinite? set \(\Rightarrow\) result}
\]

**Parameters**

- `set` – An instance of `<bit-set>`.

**Values**

- `result` – An instance of `<boolean>`.

**Discussion**

Returns `#t` if the set is infinite, otherwise `#f`.

### member? Sealed Generic function

**Signature**

\[
\text{member? set element \(\Rightarrow\) result}
\]

**Parameters**

- `set` – An instance of `<bit-set>`.
- `element` – An instance of `<integer>`.

**Values**

- `result` – An instance of `<boolean>`.

**Discussion**

Returns `#t` if `element` is a member of the set, otherwise `#f`. `element` must be a non-negative integer.

### set-add Sealed Generic function

**Signature**

\[
\text{set-add set element \(\Rightarrow\) new-set}
\]

**Parameters**

- `set1` – An instance of `<bit-set>`.
- `set2` – An instance of `<bit-set>`.

**Values**

- `new-set` – An instance of `<bit-set>`.

**Discussion**

Returns a new bit set which includes all the elements in `set` and `element` which must be a non-negative integer.

### set-add! Sealed Generic function

**Signature**

\[
\text{set-add! set element \(\Rightarrow\) new-set}
\]

**Parameters**

- `set` – An instance of `<bit-set>`.
- `element` – An instance of `<integer>`.

**Values**

- `new-set` – An instance of `<bit-set>`.

**Discussion**

Modifies `set` to include `element`. The returned set, `new-set == set. element` must be a non-negative integer.

### set-complement Sealed Generic function

**Signature**

\[
\text{set-complement set \(\Rightarrow\) new-set}
\]

**Parameters**
• **set** – An instance of `<bit-set>`.

Values

• **new-set** – An instance of `<bit-set>`.

Discussion Returns a bit-set which represents the complement of the argument set.

**set-complement!** Sealed Generic function

Signature `set-complement! set => new-set`

Parameters

• **set** – An instance of `<bit-set>`.

Values

• **new-set** – An instance of `<bit-set>`.

Discussion Alters `set` so that it contains the complement of the original set. `new-set` == `set`.

**set-difference** Sealed Generic function

Signature `set-difference set1 set2 => new-set`

Parameters

• **set1** – An instance of `<bit-set>`.
• **set2** – An instance of `<bit-set>`.

Values

• **new-set** – An instance of `<bit-set>`.

Discussion Returns a new bit-set whose elements are determined by removing elements from `set1` which are also members of `set2`. Neither `set1` or `set2` will be altered.

**set-difference!** Sealed Generic function

Signature `set-difference! set1 set2 => new-set`

Parameters

• **set1** – An instance of `<bit-set>`.
• **set2** – An instance of `<bit-set>`.

Values

• **new-set** – An instance of `<bit-set>`.

Discussion Alters `set1` by removing those elements which are also members of `set2`. `new-set` == `set1`.

**set-intersection** Sealed Generic function

Signature `set-intersection set1 set2 => new-set`

Parameters

• **set1** – An instance of `<bit-set>`.
• **set2** – An instance of `<bit-set>`.

Values

• **new-set** – An instance of `<bit-set>`.

Discussion Returns a new bit-set which represents the intersection of the argument sets. `new-set` == `set1`.
Discussion  Returns a new bit-set containing only elements which appear in both set1 and set2. Neither set1 or set2 will be altered.

**set-intersection! Sealed Generic function**  
**Signature**  set-intersection! set1 set2 => new-set  
**Parameters**  
- set1 – An instance of <bit-set>.  
- set2 – An instance of <bit-set>.  
**Values**  
- new-set – An instance of <bit-set>.  
**Discussion**  Alters set1 so that it only contains those elements which are also members of set2. new-set == set1.

**set-remove Sealed Generic function**  
**Signature**  set-remove set element => new-set  
**Parameters**  
- set – An instance of <bit-set>.  
- element – An instance of <integer>.  
**Values**  
- new-set – An instance of <bit-set>.  
**Discussion**  Returns a new bit-set which includes all the elements in set except for element which must be a non-negative integer.

**set-remove! Sealed Generic function**  
**Signature**  set-remove! set element => new-set  
**Parameters**  
- set – An instance of <bit-set>.  
- element – An instance of <integer>.  
**Values**  
- new-set – An instance of <bit-set>.  
**Discussion**  Modifies set so that it no longer contains element. The returned set, new-set == set. element must be a non-negative integer.

**set-union Sealed Generic function**  
**Signature**  set-union set1 set2 => new-set  
**Parameters**  
- set1 – An instance of <bit-set>.  
- set2 – An instance of <bit-set>.  
**Values**  
- new-set – An instance of <bit-set>.  
**Discussion**  Returns a new bit-set containing every element of set1 and set2. Neither set1 or set2 will be altered.
set-union! Sealed Generic function

Signature  set-union! set1 set2 => new-set

Parameters

• set1 – An instance of <bit-set>.
• set2 – An instance of <bit-set>.

Values

• new-set – An instance of <bit-set>.

Discussion  Alters set1 so that it also contains the elements in set2. new-set == set1.

size Sealed Generic function

Signature  size set => false-or-integer

Parameters

• set – An instance of <bit-set>.

Values

• size – Either #f or an instance of <integer>.

Discussion  Returns the cardinality of the set or #f if the set is infinite. This operation may be relatively slow.

universal-bit-set! Function

Signature  universal-bit-set! set => ()

Parameters

• set – An instance of <bit-set>.

Discussion  Destructively modifies set to include all non-negative integers as members.

4.2 The bit-vector Module

<bit-vector> Open Primary Abstract Class

Superclasses  <vector>

Init-Keywords

• size – Specifies the number of bits in the bit-vector. The default is 0.

Discussion  A compact representation of a vector of bits. The elements of the vector have the type <bit> and may be the values 0 and 1. The elements are indexed from 0 up to (size - 1).

<bit> Type

Equivalent  limited(<integer>, min: 0, max: 1)

Discussion  A subtype of <integer>, this is the type of elements of <bit-vector>. Objects of this type may have the value 0 or 1.

bit-count Function

Signature  bit-count vector #key bit-value => count

Parameters
vector – An instance of `<bit-vector>`.

bit-value (#key) – An instance of `<bit>`.

Values

• count – An instance of `<integer>`.

Discussion Returns the number of bits in vector which are equal to bit-value. This may be a relatively slow operation.

bit-vector-and Function

Signature bit-vector-and vector1 vector2 #key pad1 pad2 => result pad

Parameters

• vector1 – An instance of `<bit-vector>`.

• vector2 – An instance of `<bit-vector>`.

• pad1 (#key) – An instance of `<bit>`.

• pad2 (#key) – An instance of `<bit>`.

Values

• result – An instance of `<bit-vector>`.

• pad – An instance of `<bit>`.

Discussion Returns a new vector which is the bitwise and of the two argument vectors. Each vector has an associated pad value. If the vectors are of different lengths, the shorter is considered extended with its pad value. The size of the result vector may be extended or shortened provided the bits added or dropped are the same as the result-pad value. The size of the result will be no smaller than the minimum of the argument sizes, and no greater than the maximum of the argument sizes. The result-pad value is calculated by taking the logical and of the two input pad values.

bit-vector-and! Function

Signature bit-vector-and! vector1 vector2 #key pad1 pad2 => result pad

Parameters

• vector1 – An instance of `<bit-vector>`.

• vector2 – An instance of `<bit-vector>`.

• pad1 (#key) – An instance of `<bit>`.

• pad2 (#key) – An instance of `<bit>`.

Values

• result – An instance of `<bit-vector>`.

• pad – An instance of `<bit>`.

Discussion Returns a vector which is the bitwise and of the two argument vectors. vector1 may or may not be modified by this operation. Each vector has an associated pad value. If the vectors are of different lengths, the shorter is considered extended with its pad value. The size of the result vector may be extended or shortened provided the bits added or dropped are the same as the result-pad value. The size of the result will be no smaller than the minimum of the argument sizes, and no greater than the maximum of the argument sizes. The result-pad value is calculated by taking the logical and of the two input pad values.
bit-vector-andc2 Function

Signature  bit-vector-andc2 vector1 vector2 #key pad1 pad2 => result pad

Parameters

• vector1 – An instance of <bit-vector>.
• vector2 – An instance of <bit-vector>.
• pad1 (#key) – An instance of <bit>.
• pad2 (#key) – An instance of <bit>.

Values

• result – An instance of <bit-vector>.
• pad – An instance of <bit>.

Discussion  Returns a new vector which is the result of taking the bitwise and of vector1 and the bitwise not of vector2. Each vector has an associated pad value. If the vectors are of different lengths, the shorter is considered extended with its pad value. The size of the result vector may be extended or shortened provided the bits added or dropped are the same as the result-pad value. The size of the result will be no smaller than the minimum of the argument sizes, and no greater than the maximum of the argument sizes. The result-pad value is calculated by taking the logical and of pad1 with the complement of pad2.

bit-vector-andc2! Function

Signature  bit-vector-andc2! vector1 vector2 #key pad1 pad2 => result pad

Parameters

• vector1 – An instance of <bit-vector>.
• vector2 – An instance of <bit-vector>.
• pad1 (#key) – An instance of <bit>.
• pad2 (#key) – An instance of <bit>.

Values

• result – An instance of <bit-vector>.
• pad – An instance of <bit>.

Discussion  Returns a vector which is the result of taking the bitwise and of vector1 and the bitwise not of vector2. vector1 may or may not be modified by this operation. Each vector has an associated pad value. If the vectors are of different lengths, the shorter is considered extended with its pad value. The size of the result vector may be extended or shortened provided the bits added or dropped are the same as the result-pad value. The size of the result will be no smaller than the minimum of the argument sizes, and no greater than the maximum of the argument sizes. The result-pad value is calculated by taking the logical and of pad1 with the complement of pad2.

bit-vector-not Function

Signature  bit-vector-not vector #key pad => result result-pad

Parameters

• vector – An instance of <bit-vector>.
• pad (#key) – An instance of <bit>.
Values

• **result** – An instance of `<bit-vector>`.
• **result-pad** – An instance of `<bit>`.

Discussion Returns a new vector which is the bitwise not of its argument.

**bit-vector-not! Function**

Signature `bit-vector-not! vector #key pad => result result-pad`

Parameters

• `vector` – An instance of `<bit-vector>`.
• `pad (#key)` – An instance of `<bit>`.

Values

• **result** – An instance of `<bit-vector>`.
• **result-pad** – An instance of `<bit>`.

Discussion Modifies `vector` so that it becomes the bitwise not of its original contents. `result == vector`.

**bit-vector-or Function**

Signature `bit-vector-or vector1 vector2 #key pad1 pad2 => result pad`

Parameters

• `vector1` – An instance of `<bit-vector>`.
• `vector2` – An instance of `<bit-vector>`.
• `pad1 (#key)` – An instance of `<bit>`.
• `pad2 (#key)` – An instance of `<bit>`.

Values

• **result** – An instance of `<bit-vector>`.
• **pad** – An instance of `<bit>`.

Discussion Returns a new vector which is the bitwise or of the two argument vectors. Each vector has an associated pad value. If the vectors are of different lengths, the shorter is considered extended with its pad value. The size of the result vector may be extended or shortened provided the bits added or dropped are the same as the result-pad value. The size of the result will be no smaller than the minimum of the argument sizes, and no greater than the maximum of the argument sizes. The result-pad value is calculated by taking the logical or of the two input pad values.

**bit-vector-or! Function**

Signature `bit-vector-or! vector1 vector2 #key pad1 pad2 => result pad`

Parameters

• `vector1` – An instance of `<bit-vector>`.
• `vector2` – An instance of `<bit-vector>`.
• `pad1 (#key)` – An instance of `<bit>`.
• `pad2 (#key)` – An instance of `<bit>`.
Values

- **result** – An instance of `<bit-vector>`.
- **pad** – An instance of `<bit>`.

**Discussion** Returns a vector which is the bitwise or of the two argument vectors. `vector1` may or may not be modified by this operation. Each vector has an associated pad value. If the vectors are of different lengths, the shorter is considered extended with its pad value. The size of the result vector may be extended or shortened provided the bits added or dropped are the same as the result-pad value. The size of the result will be no smaller than the minimum of the argument sizes and no greater than the maximum of the argument sizes. The result-pad value is calculated by taking the logical or of the two input pad values.

**bit-vector-xor Function**

**Signature** `bit-vector-xor vector1 vector2 #key pad1 pad2 => result pad`

**Parameters**

- **vector1** – An instance of `<bit-vector>`.
- **vector2** – An instance of `<bit-vector>`.
- **pad1 (#key)** – An instance of `<bit>`.
- **pad2 (#key)** – An instance of `<bit>`.

**Values**

- **result** – An instance of `<bit-vector>`.
- **pad** – An instance of `<bit>`.

**Discussion** Returns a new vector which is the bitwise exclusive or of the two argument vectors. Each vector has an associated pad value. If the vectors are of different lengths, the shorter is considered extended with its pad value. The size of the result will be no smaller than the minimum of the argument sizes, and no greater than the maximum of the argument sizes. The size of the result vector may be extended or shortened provided the bits added or dropped are the same as the result-pad value. The result-pad value is calculated by taking the logical xor of the two input pad values.

**bit-vector-xor! Function**

**Signature** `bit-vector-xor! vector1 vector2 #key pad1 pad2 => result pad`

**Parameters**

- **vector1** – An instance of `<bit-vector>`.
- **vector2** – An instance of `<bit-vector>`.
- **pad1 (#key)** – An instance of `<bit>`.
- **pad2 (#key)** – An instance of `<bit>`.

**Values**

- **result** – An instance of `<bit-vector>`.
- **pad** – An instance of `<bit>`.

**Discussion** Returns a vector which is the bitwise exclusive or of the two argument vectors. `vector1` may or may not be modified by this operation. Each vector has an associated pad value. If the vectors are of different lengths, the shorter is considered extended with its pad value. The size of the result vector may be extended or shortened provided the bits added or
dropped are the same as the result-pad value. The size of the result will be no smaller than the minimum of the argument sizes, and no greater than the maximum of the argument sizes. The result-pad value is calculated by taking the logical xor of the two input pad values.

4.3 The collectors Module

collect Macro
collect-first Macro
collect-first-into Macro
collect-into Macro
collect-last Macro
collect-last-into Macro
collected Macro
collecting Macro
collector-protocol Open Generic function

Signature collector-protocol class => new-collector add-first add-last add-sequence-first add-sequence-last collection

Parameters

• class – An instance of <object>.

Values

• new-collector – An instance of <object>.
• add-first – An instance of <function>.
• add-last – An instance of <function>.
• add-sequence-first – An instance of <function>.
• add-sequence-last – An instance of <function>.
• collection – An instance of <function>.

4.4 The plists Module

4.4.1 Overview

Property lists associate values with keys, but without the overhead of a <table>. These are generally useful for a small number of keys or where memory usage is a concern.

This implementation allows plists to be stored in either a <list> or a <vector>. The keys are commonly called indicators and are typically a <symbol>. The values can be any <object>.

In memory, the plist is arranged with keys and values alternating in a single sequence:

```
# [key1:, 1,  key2:, 2]
```

Conveniently, this is the same layout as a sequence of arguments when using #rest in conjunction with keyword arguments and #all-keys as can be seen in the example using with-keywords-removed.
Reading

• get-property

Modifying

• put-property!
• remove-keywords
• remove-property!
• with-keywords-removed

Iterating

• keyword-sequence
• value-sequence

4.4.2 Reference

get-property Generic function
Return the value for an indicator, with a default should it not exist.

Signature get-property plist indicator #key default => property

Parameters

• plist – An instance of <sequence>.
• indicator – An instance of <object>.
• default (#key) – An instance of <object>.

Values

• property – An instance of <object>.

keyword-sequence Generic function
Returns a sequence containing the indicators in the plist.

Signature keyword-sequence plist => keywords

Parameters

• plist – An instance of <sequence>.

Values

• keywords – An instance of <sequence>.

See also

• value-sequence

put-property! Statement Macro
Modify the plist, adding indicator with the given value.

Macro Call
put-property!(*plist*, *indicator*, *value*)

Parameters

- **plist** – An instance of `<sequence>`.
- **indicator** – An instance of `<object>`.
- **value** – An instance of `<object>`.

Example

```
put-property!(buffer-contents-properties(buffer),
             #'optimization-colors', #f)
```

See also

- `remove-property!`

remove-keywords **Generic function**

Returns a copy of the `plist` with `keywords` removed.

**Signature**  
`remove-keywords` `plist` `keywords` `=>` `plist`

Parameters

- **plist** – An instance of `<sequence>`.
- **keywords** – An instance of `<sequence>`.

Values

- **plist** – An instance of `<sequence>`.

See also

- `remove-property!`
- `with-keywords-removed`

remove-property! **Statement Macro**

Modify the `plist`, removing `indicator`, returning the old value, if any.

**Macro Call**

```
remove-property!(*plist*, *indicator*)
```

Parameters

- **plist** – An instance of `<sequence>`.
- **indicator** – An instance of `<object>`.

Values

- **value** – An instance of `<object>`.

Example

```
remove-property!(buffer-properties(buffer), #'project);
```

See also

- `put-property!`
- `remove-keywords`
• **with-keywords-removed**

**value-sequence** **Generic function**

Returns a sequence containing the values in the `plist`.

**Signature**  
`value-sequence plist => values`

**Parameters**

- `plist` – An instance of `<sequence>`.

**Values**

- `values` – An instance of `<sequence>`.

**See also**

- `keyword-sequence`

**with-keywords-removed Statement Macro**

**Macro Call**

```dylan
with-keywords-removed(*var* = *plist*, *keywords*)
*body*
end
```

**Parameters**

- `var` – A Dylan name `bnf`.
- `plist` – An instance of `<sequence>`.
- `keywords` – An instance of `<sequence>`.
- `body` – A Dylan body `bnf`.

**Discussion**

Executes the body, with the `keywords` removed from `plist` and the modified `plist` available as `var`.

**Example**

```dylan
define sealed method make
  (class == <interval-stream>, #rest initargs,
   #key buffer, interval, direction, #all-keys)
=> (stream :: <interval-stream>)
  ignore(direction);
  let (start-bp, end-bp) = values(interval-start-bp(buffer | interval),
    interval-end-bp(buffer | interval));
  let buffer
    = buffer
      | select (interval by instance?)
        <buffer> => interval;
        otherwise => bp-buffer(start-bp);
      end;
  with-keywords-removed (initargs = initargs, ![interval:])
  apply(next-method, class,
    start-bp: start-bp, end-bp: end-bp,
    buffer: buffer, initargs)
  end
end method make;
```

**See also**

4.4. The plists Module
• `remove-keywords`
• `remove-property!`
• `with-keywords-removed`

## 4.5 The set Module

<object-set> Class  
Superclasses <set>  
Init-Keywords  
• `size`

<set> Open Abstract Class  
Superclasses <mutable-explicit-key-collection>

## 4.6 The table-extensions Module

The Collections library’s `table-extensions` module extends the Dylan language’s standard table features.

**Note:** Open Dylan provides a slightly different table implementation from that described by the DRM. See [Language differences](#) for details of these differences.

### 4.6.1 Basics

The `table-extensions` module exports the classes `<string-table>` and `<case-insensitive-string-table>; the type `<hash-state>; the constructor macro `table`; the generic function `remove-all-keys!` and two methods thereon; and the functions `collection-hash`, `sequence-hash`, `string-hash`, `values-hash`, `case-insensitive-string-hash`, and `case-insensitive-equal`.

The `<string-table>` class is a class of tables that use strings for keys. `<case-insensitive-string-table>` is similar, but the keys are considered to be case insensitive.

The `<hash-state>` type implements *hash states*. A hash state is defined by the DRM, page 123, as “an implementation-dependent type that is associated with a hash id and can be used by the implementation to determine whether the hash id has been invalidated.” See pages 122–123 of the DRM for more details.

The various hash functions and the `case-insensitive-equal` equivalence predicate are convenient building blocks for creating new table classes and hash functions.

### 4.6.2 Hash functions

Different hash functions are not required to return the same hash code for equal or even identical objects. For instance,

```
collection-hash(#(), object-hash, object-hash);
```

is not guaranteed to return the same values as
Furthermore, `collection-hash` with `ordered: #t` is not guaranteed to return the same hash code as `collection-hash` with `ordered: #f`. Such a requirement would render the `ordered:` keyword useless.

### 4.6.3 Weak tables

Open Dylan allows all general instances of the built-in class `<table>` to be weak. See `weak tables` of this volume for information about weakness.

You can create weak tables with the `<table>` class’s `weak:` init-keyword. The legal values for this keyword are:

- `#"key"` Creates a table with weak keys. When there are no longer any strong references to a key, the table entry of which it is part becomes eligible for garbage collection.
- `#"value"` Creates a table with weak values. When there are no longer any strong references to a value, the table entry of which it is a part becomes eligible for garbage collection.
- `#f` Creates a table with strong keys and values. This is the default value.

#### The table-extensions Module

This section contains a reference description for each item exported from the module `table-extensions`.

**<string-table> Sealed Class**

A table class that uses strings for keys.

**Superclasses** `<table>`

**Discussion**

The `<string-table>` class is the class of tables that use instances of `<string>` for their keys. It is an error to use a key that is not an instance of `<string>`.

Keys are compared with the equivalence predicate `\=.`

The elements of the table are instances of `<object>`.

Modifying the key once it has been used to add an element to a `<string-table>` results in undefined behavior.

**<case-insensitive-string-table> Sealed Class**

A table class that uses case-insensitive strings for keys.

**Superclasses** `<table>`

**Discussion**

The `<string-table>` class is the class of tables that use instances of `<string>` for their keys. It is an error to use a key that is not an instance of `<string>`.

Keys are compared with the equivalence predicate `case-insensitive-equal`.

The elements of the table are instances of `<object>`.

Modifying the key once it has been used to add an element to a `<case-insensitive-string-table>` results in undefined behavior.

**<hash-state> Class**

A hash state.

**Superclasses** `<object>`
Discussion

Anything that the Dylan Reference Manual describes as a hash state is an instance of this type.

Examples of hash states include the second argument and second return value of object-hash.

collection-hash Function

Hashes the elements of a collection.

**Signature**

\[
\text{collection-hash \ key-hash-function \ elt-hash-function \ collection \ initial-state \ #key \ ordered} \\
\Rightarrow \text{hash-id} \ \text{hash-state}
\]

**Parameters**

- **key-hash-function** – An instance of `<function>`.
- **elt-hash-function** – An instance of `<function>`.
- **collection** – An instance of `<collection>`.
- **initial-state** – An instance of `<hash-state>`.
- **ordered(#key)** – An instance of `<boolean>`. Default value: `#f`.

**Values**

- **hash-id** – An instance of `<integer>`.
- **result-state** – An instance of `<hash-state>`.

**Discussion**

Hashes every element of `collection` using `key-hash-function` on the keys and `elt-hash-function` on the elements, and merges the resulting hash codes in order.

The `ordered` keyword is passed on to `merge-hash-ids`.

The functions `key-hash-function` and `elt-hash-function` must be suitable for use as hash functions. See page 123 of the DRM.

sequence-hash Function

Hashes the elements of a sequence.

**Signature**

\[
\text{sequence-hash \ elt-hash-function \ sequence \ initial-state \ #key \ ordered} \\
\Rightarrow \text{hash-id} \ \text{result-state}
\]

**Parameters**

- **elt-hash-function** – An instance of `<function>`.
- **sequence** – An instance of `<sequence>`.
- **initial-state** – An instance of `<hash-state>`.
- **ordered(#key)** – An instance of `<boolean>`. Default value: `#f`.

**Values**

- **hash-id** – An instance of `<integer>`.
- **result-state** – An instance of `<hash-state>`.

**Discussion**

Hashes every element of `sequence` using `elt-hash-function`, and merges the resulting hash codes in order.
The function \textit{elt-hash-function} must be suitable for use as a hash function. See page 123 of the DRM.

The \textit{ordered} keyword is passed on to \textit{merge-hash-ids}.

\textbf{values-hash Function}

Hashes the values passed to it.

\textbf{Signature} \texttt{values-hash elt-hash-function initial-state \#rest arguments \Rightarrow hash-id result-state}

\textbf{Parameters}

- \texttt{elt-hash-function} – An instance of \texttt{<function>}.  
- \texttt{hash-state} – An instance of \texttt{<hash-state>}.  
- \texttt{initial-state} – An instance of \texttt{<hash-state>}.  
- \texttt{arguments (\#rest)} – Instances of \texttt{<object>}.  

\textbf{Values}

- \texttt{hash-id} – An instance of \texttt{<integer>}.  
- \texttt{result-state} – An instance of \texttt{<hash-state>}.  

\textbf{Discussion}

Hashes every object in \texttt{arguments} using \textit{elt-hash-function}, and merges the resulting hash codes in order.

The function \textit{elt-hash-function} must be suitable for use as a hash function. See page 123 of the DRM.

The \textit{ordered} keyword is passed on to \textit{merge-hash-ids}.

\textbf{string-hash Function}

Hashes a string.

\textbf{Signature} \texttt{string-hash string initial-state \Rightarrow hash-id result-state}

\textbf{Parameters}

- \texttt{string} – An instance of \texttt{<string>}.  
- \texttt{initial-state} – An instance of \texttt{<hash-state>}.  

\textbf{Values}

- \texttt{hash-id} – An instance of \texttt{<integer>}.  
- \texttt{result-state} – An instance of \texttt{<hash-state>}.  

\textbf{Discussion}

Produces a hash code for a string, using the equivalence predicate \texttt{\=}.  

\textbf{case-insensitive-string-hash Function}

Hashes a string, without considering case information.

\textbf{Signature} \texttt{case-insensitive-string-hash string initial-state \Rightarrow hash-id result-state}

\textbf{Parameters}

- \texttt{string} – An instance of \texttt{<string>}.  
- \texttt{initial-state} – An instance of \texttt{<hash-state>}.  

\textbf{Values}

- \texttt{hash-id} – An instance of \texttt{<integer>}.  

4.6. The table-extensions Module
• **result-state** – An instance of `<hash-state>`.

**Discussion** Produces a hash code for a string using the equivalence predicate `case-insensitive-equal`, which does not consider the case of the characters in the strings it compares.

**See also**

• `case-insensitive-equal`

---

### `case-insensitive-equal` Function

Compares two strings for equality, ignoring case differences between them.

**Signature** `case-insensitive-equal string1 string2 => boolean`

**Parameters**

• `string1` – An instance of `<string>`.
• `string2` – An instance of `<string>`.

**Values**

• `boolean` – An instance of `<boolean>`.

**Discussion**

Compares `string1` and `string2` for equality, ignoring any case differences between them. Returns true if they are equal and false otherwise.

The function has the same behavior as Dylan’s standard method on `=` for sequences, except that when comparing alphabetical characters, it ignores any case differences.

This function is used as an equivalence predicate by `case-insensitive-string-hash`.

This function uses `as-uppercase` or `as-lowercase` to convert the characters in its string arguments.

**Example** The `case-insensitive-equal` function returns true if passed the following strings:

```
"The Cat SAT ON the Mat"
"The cat sat on the Mat"
```

Conversely, the standard method on `=` returns false when passed those strings.

**See also**

• `case-insensitive-string-hash`

---

### `remove-all-keys!` Open Generic function

Removes all keys from a collection and leaves it empty.

**Signature** `remove-all-keys! collection => collection`

**Parameters**

• `collection` – An instance of `<mutable-explicit-key-collection>`.

**Values**

• `collection` – An instance of `<mutable-explicit-key-collection>`.

**Discussion**

Modifies `collection` by removing all its keys and elements, and leaves it empty.
remove-all-keys!(<mutable-explicit-key-collection>) Method
Removes all keys from a collection and leaves it empty.

Signature
remove-all-keys! collection => collection

Parameters
• collection – An instance of <mutable-explicit-key-collection>.

Values
• collection – An instance of <mutable-explicit-key-collection>.

Discussion
Modifies collection by removing all its keys and elements, and leaves it empty. This method implements the generic function by making repeated calls to remove-key!.

remove-all-keys!(<table>) Method
Removes all keys from a table and leaves it empty.

Signature
remove-all-keys! table => table

Parameters
• table – An instance of <table>.
• table – An instance of <table>.

Discussion
Modifies table by removing all its keys and elements, and leaves it empty.

Note: To empty collections that are not instances of <mutable-explicit-key-collection>, use size-setter.

table Function Macro
Creates a table and populates it with keys and values.

Macro Call
table( { class, } key => value, ...)

Parameters
• class – An instance of <class>. Optional.
• key – An expression.
• value – An expression.

Values
• **table** – A new instance of `class`.

**Discussion** Creates a table of type `class` and populates it with key/value pairs. If `class` is omitted, creates a table of type `<table>`.

**Example**

```dylan
let my-table = table("red"=>"stop", "green"=>"go");
let my-table = table(<string-table>, "red"=>"stop", "green"=>"go");
```

The **collections** library also re-exports the **byte-vector module** from the **common-dylan library**.
CHAPTER
FIVE

THE COLORING-STREAM LIBRARY

The coloring-stream module provides a means for writing text to the console or terminal with colors and varying intensity.

On Unix, this is typically done via ANSI codes. Windows support is yet to be implemented.

5.1 Usage

It is implemented as a <wrapper-stream>, so it can wrap any given stream. While it can be created via make, it is easiest to just call colorize-stream on a stream to get back a <coloring-stream> wrapper that can be used to output color.

If the underlying stream doesn’t support color, then the returned <coloring-stream> will silently drop all text attributes.

Once you have a <coloring-stream>, you can set text attributes by writing instances of <text-attributes>. Text attributes are created with text-attributes. Text can have a foreground color, a background color and an intensity:

```plaintext
define error-attributes = text-attributes(foreground: $color-red);
define blinding-attributes = text-attributes(foreground: $color-green, background: $color-red, intensity: $bright-intensity);
```

To reset text back to the defaults, use the predefined $reset-attributes constant.

Text attributes can be stored and used repeatedly. This lets you set up some stylized text attributes and re-use them.

The text attributes can be written to the stream in two different ways:

- Via format
- Via print or print-object

Once you have a <coloring-stream>, then you can write colorized text to it with format:

```plaintext
define error-attributes = text-attributes(foreground: $color-red);
define cs = colorize-stream(*standard-output*);
format(cs, "%Error:%= %s", error-attributes, $reset-attributes, error-message);
```

This is implemented by providing an implementation of print specialized on <text-attributes> and <coloring-stream>.

This can be used directly as well:
let error-attributes = text-attributes(foreground: $color-red);
let cs = colorize-stream(*standard-output*);
print(error-attributes, cs);
print("Error:", cs);
print($reset-attributes, cs);
print(' ', cs);
print(error-message, cs);

5.2 The coloring-stream Module

<coloring-stream> Abstract Class

Superclasses <wrapper-stream>

Discussion <coloring-stream> is the abstract wrapper stream that is used to determine whether or not a stream supports color output to avoid having to check at every write.

colorize-stream Generic function

Signature colorize-stream (stream #key force-ansi?) => (coloring-stream)

Parameters

• stream – An instance of <stream>.
• force-ansi? (#key) – An instance of <boolean>.

Values

• coloring-stream – An instance of <coloring-stream>.

Discussion

Wrap a stream with an appropriate instance of <coloring-stream>. It uses stream-supports-color? to determine whether or not the underlying stream supports color output.

When force-ansi? is #t, then the usual checks are skipped and a coloring stream that generates ANSI output is created. This is useful when outputting to a string prior to writing the text to *standard-output* or when writing a network server where the user may have an ANSI-capable client.

When called on a <coloring-stream>, if force-ansi? is set and the stream is not an ANSI coloring stream, then the stream will be unwrapped and a new ANSI coloring stream wrapper will be created. Otherwise, calling colorize-stream on a <coloring-stream> will return the same stream.

Example

let stream = colorize-stream(*standard-output*);

Or, using force-ansi?:

let text
  = with-output-to-string (s :: <byte-string-stream>)
  let force-ansi? = stream-supports-color?(*standard-output*);
  let s = colorize-stream(s, force-ansi?: force-ansi?);
  ...

(continues on next page)
stream-supports-color? Open Generic function

Signature stream-supports-color? (stream) => (well?)

Parameters
  • stream – An instance of <stream>.

Values
  • well? – An instance of <boolean>.

Discussion
Return whether or not the underlying stream supports color output.
This checks that:
  • The stream is a <file-stream>.
  • stream-console? is true. (On Unix, this means that isatty is true for the stream.)
  • The TERM environment variable is not "dumb".
  • The EMACS environment variable is not "t".

stream-supports-ansi-color? Open Generic function

Signature stream-supports-ansi-color? (stream) => (well?)

Parameters
  • stream – An instance of <stream>.

Values
  • well? – An instance of <boolean>.

Discussion
Return whether or not the underlying stream might support ANSI color output, assuming that
the underlying stream supports color output at all.
On Unix, this will always return #t.
On Windows, this attempts to detect situations where ANSI output would be permissible, such
as running within an alternate console window like ConEMU.

Note: This does NOT check to see if the stream actually supports coloring. It is meant to be
used in conjunction with stream-supports-color?.

5.2.1 Text Attributes
<text-attributes> Class

Superclasses <object>

Init-Keywords
  • background – An instance of false-or(<text-color>).
• **foreground** – An instance of false-or(<text-color>).
• **intensity** – An instance of false-or(<text-intensity>).

**Discussion**

Instances of `<text-attributes>` are used for representing the desired text appearance. They can be passed to `format` when writing to a `<coloring-stream>`. `background` and `foreground`, if given, should be one of the color constants like `$color-red`, `$color-green`, etc.

`intensity`, if given, should be one of `$bright-intensity`, `$dim-intensity` or `$normal-intensity`.

Values that are omitted are set to the default values for the terminal.

**text-attributes Function**

**Signature**  
`text-attributes (#key foreground background intensity) => (attributes)`

**Parameters**

• **foreground** (#key) – An instance of false-or(<text-color>).
• **background** (#key) – An instance of false-or(<text-color>).
• **intensity** (#key) – An instance of false-or(<text-intensity>).

**Values**

• **attributes** – An instance of `<text-attributes>.

**Discussion**

`text-attributes` provides an easy wrapper for creating instances of `<text-attributes>`. `background` and `foreground`, if given, should be one of the color constants like `$color-red`, `$color-green`, etc.

`intensity`, if given, should be one of `$bright-intensity`, `$dim-intensity` or `$normal-intensity`.

Values that are omitted are set to the default values for the terminal.

**Example**

```
let error-attributes = text-attributes(foreground: $color-red);
```

**$reset-attributes Constant**

**Type** `<text-attributes>`

**Discussion**

A predefined constant to reset the text back to the default attributes.

This is equivalent to `text-attributes(intensity: $normal-intensity).`

**Text Intensity**

**$bright-intensity Constant**

**$dim-intensity Constant**
Discussion

Note: Not all terminals support dimmed text.

$normal-intensity Constant

Text Colors

$color-black Constant
$color-blue Constant
$color-cyan Constant
$color-default Constant
$color-green Constant
$color-magenta Constant
$color-red Constant
$color-white Constant
$color-yellow Constant
The command-line-parser library provides a facility to parse the command line. It exports two modules:

- `command-line-parser` - Main API module
- `option-parser-protocol` - For extending the API

### 6.1 Quick Start

Let’s say you want to parse a command line that looks like this:

```
frob --name=zoo --debug -r a -r b -r c --choice=foo one two three
```

The “frob” command accepts a `--name` argument that takes a value, a boolean `--debug` (or `--nodebug`) a `-r` argument that may be repeated, and then at least one positional argument (here “one”, “two”, and “three”). Here’s how to create a parser for the frob command:

```plaintext
let parser = make(<command-line-parser>, min-positional-options: 1);
add-option(parser,
  make(<parameter-option>,
    names: #("name"),
    help: "A name"));
add-option(parser,
  make(<flag-option>,
    names: #("debug"),
    negative-names: #("nodebug"),
    default: #f,
    help: "Enable or disable debugging"));
add-option(parser,
  make(<repeated-parameter-option>,
    names: #("r"),
```

(continues on next page)
There are additional option classes not shown here. See the reference section for more info.

Now parse the command line:

```dylan
block ()
  parse-command-line(parser, application-arguments(),
  description: "The most excellent Frobber.");
exception (ex :: <help-requested>)
  exit-application(0);
exception (ex :: <usage-error>)
  exit-application(2);
end;
```

And to access the option values:

```dylan
let debug? :: <boolean> = get-option-value(parser, "debug");
let name :: false-or(<string>) = get-option-value(parser, "name");
let dash-r :: <deque> = get-option-value(parser, "r");
let choice :: <string> = get-option-value(parser, "choice");
let args :: <sequence> = positional-options(parser);
```

# 6.2 Reference

## 6.2.1 The command-line-parser Module

<command-line-parser> Open Class

Encapsulates a set of command-line options.

**Superclasses** <object>

**Init-Keywords**

- **provide-help-option** – A boolean specifying whether the parser should automatically add the default help option. By default, help may be requested via `--help` or `-h`. If `#f`, no help option will be added to the parser, and you must explicitly handle any request for help yourself.

- **help-option** – A `<flag-option>` that will be added to the parser as the option that signals a request for help. The main purpose of this init keyword is to make it possible to use something other than `--help` and `-h` to request help. This keyword has no effect if `provide-help-option?` is `#f`.

- **min-positional-options** – The minimum number of positional (unnamed) options. An `<integer>`, defaulting to zero. If fewer positional options than this are supplied, `<usage-error>` is signaled.
max-positional-options – The maximum number of positional (unnamed) options. An <integer>, defaulting to $maximum-integer. If more positional options than this are supplied, <usage-error> is signaled.

<command-line-parser-error> Open Class
Superclass of all errors signaled by this library.

Superclasses <format-string-condition>,<error>

<usage-error> Open Class
Signaled when a command-line cannot be parsed.

Superclasses <command-line-parser-error>

Discussion This is commonly handled by calling exit-application(2) since the error has already been displayed on *standard-error*.

<help-requested> Sealed Class
Signaled when help was explicitly requested via the help option, usually --help.

Superclasses <usage-error>

Discussion This is commonly handled by calling exit-application(0) since the command-line synopsis has already been displayed on *standard-output*.

add-option Function
Add an option to a command-line parser.

Signature add-option (parser option) => ()

Parameters
• parser – An instance of <command-line-parser>.
• option – An instance of <option>.

Discussion If any of the option names specified are already used by other options then <command-line-parser-error> is signaled.

parse-command-line Function
Parses the command line in argv and side-effects parser accordingly.

Signature parse-command-line (parser argv) => ()

Parameters
• parser – An instance of <command-line-parser>.
• argv – An instance of <sequence>. Normally the value returned by application-arguments is passed here.
• usage(#key) – As for print-synopsis.
• description(#key) – As for print-synopsis.

Discussion
By default, the --help flag is handled automatically by displaying the usage string, the description, and calling print-synopsis(parser, *standard-output*). Then <help-requested> is signaled and the caller should handle it, perhaps by calling exit-application(0).

If argv isn’t a valid set of options as described by the parser then <usage-error> is signaled and the caller should handle it, perhaps by calling exit-application(2).

See Quick Start for an example.
**print-synopsis** Open Generic function

Display a synopsis of the command line described by `parser` on `stream`.

**Signature**

\[
\text{print-synopsis (parser stream) => ()}
\]

**Parameters**

- **parser** – An instance of `<command-line-parser>`.
- **stream** – An instance of `<stream>`.
- **usage** (#key) – An instance of `<string>` or `#f`. A brief synopsis of the overall command-line syntax. The default is `#f`, in which case “Usage: `<application-name> [options]\n`” will be displayed, where `<application-name>` is the result of calling `locator-base(application-name())`.
- **description** (#key) – An instance of `<string>` or `#f`. This is displayed after `usage` and before the detailed list of options. This is intended to be a sentence or short paragraph.

**positional-options** Generic function

Returns the sequence of command line arguments that remain after all optional arguments have been consumed.

**Signature**

\[
\text{positional-options (parser) => (args :: <sequence>)}
\]

**Parameters**

- **object** – An instance of `<object>`.

**Values**

- **#rest results** – An instance of `<object>`.

**option-present?** Function

Returns `#t` if this option was supplied on the command line.

**Signature**

\[
\text{option-present? (parser name) => (present?)}
\]

**Parameters**

- **parser** – An instance of `<command-line-parser>`.
- **name** – An instance of `<string>`.

**Values**

- **present?** – An instance of `<boolean>`.

**Discussion** If called before `parse-command-line` has been called on the associated parser, this will always return `#f`.

**get-option-value** Function

Retrieves an option from an `<command-line-parser>` by its long name.

**Signature**

\[
\text{get-option-value (parser long-name) => (value)}
\]

**Parameters**

- **parser** – An instance of `<command-line-parser>`.
- **long-name** – An instance of `<string>`.

**Values**

- **value** – An instance of `<object>`. 

---

Chapter 6. The command-line-parser Library
Option Classes

<option> Open Primary Abstract Class
Superclass of all other option types.

Superclasses <object>

Init-Keywords

• names – Names for this option; a sequence of strings. For convenience a single string may also be specified. Strings of length 1 are considered to be short options, i.e., they are prefixed by a single dash on the command line.

• type – The kind of value represented by this option. That is, the string passed on the command line will be coerced to this type via the parse-option-parameter generic function. Clients may implement that function for their own types to extend the parser.

Predefined types include <integer>, subclass(<float>), subclass(<sequence>).

• help – A string documenting the option. Displayed in --help output. Some automatic substitutions are performed:

  1. "%default" => the string representation of the default value for the option.
  2. "%app" => the basename of the executable program.
  3. "%%" => "%"

• variable – A string to stand in for the option value in --help output. For example, if the option name is --database this might be "URL", which would display as:

   --database URL A database URL.

• default – A default value for the option that will be used if the option isn’t specified by the user.

<flag-option> Sealed Class
Defines a simple flag option, i.e., one that specifies a boolean value.

Superclasses <option>

Init-Keywords

• negative-names – Same as names, but specifies the negative forms.

Discussion

They default to #f and exist in both positive and negative forms: “--foo” and “--no-foo”. In the case of conflicting options, the rightmost takes precedence to allow for abuse of the shell’s “alias” command.

For example, a single instance of this class could be used to specify all of the following command-line options:

- q, -v, --quiet, --verbose

<parameter-option> Sealed Class
Defines an option that requires a value be specified.

Superclasses <option>
Discussion

If the option appears more than once, the rightmost value takes precedence. If the option never appears, these will default to #f.

Examples:

-cred, -c=red, -c = red, --color red, --color=red

<optional-parameter-option> Sealed Class

Similar to <parameter-option>, but the parameter is optional.

Superclasses <option>

Discussion

The parameter must directly follow the option with no intervening whitespace, or follow an “=” token. The value is #f if the option never appears, #t if the option appears but the parameter does not, and the value of the parameter otherwise.

Examples:

-z, -z3, -z=3, -z = 3, --zip, --zip=3, --zip = 3

Invalid examples:

-z 3, --zip 3, --zip3

<repeated-parameter-option> Sealed Class

Similar to <parameter-option>, but may appear more than once.

Superclasses <option>

Discussion

The final value is a deque of parameter values in the order they appeared on the command line. It defaults to the empty deque.

Examples:

-wall, -w=all, -w = all, --warnings all, --warnings=all

<choice-option> Sealed Class

Similar to <parameter-option>, but provides a restricted set of values to choose from.

Superclasses <parameter-option>

Init-Keywords

- choices – A sequence of objects (usually strings). If the value supplied on the command line isn’t one of these objects then <usage-error> is signaled. If you supply a sequence of non-string choices you will also need to supply the test: init keyword since all command-line arguments are strings and won’t compare equal with the default test, =.

- test – A function to test whether the value supplied on the command line is the same as one of the choices. The default is =. Another commonly used value is string-equal-ic?, to ignore case in the comparison.

Discussion

Example:
<keyed-option> Sealed Class

Each occurrence of this type of option defines a key => value mapping.

Superclasses <option>

Discussion

These are a bit obscure. The best example is gcc’s `-D` option. The final value is a <string-table> containing each specified key, with one of the following values:

- `#t`: The user specified “-Dkey”
- a string: The user specified “-Dkey=value”

You can read this with \texttt{element(table, key, default: #f)} to get a handy lookup table.

Examples:

\texttt{-Dkey, -Dkey=value, -D key = value, --define key = value}

\textbf{option-parser-definer Macro}

6.2.2 The option-parser-protocol Module

This module exports an API that can be used to extend the existing command line parser without modifying the source in this library. It shouldn’t be common to need this. See the source code for details.
The common-dylan library contains a number of features that were either omitted from the Dylan language described in the DRM, or that Open Dylan’s developers have found useful in a broad range of situations.

### 7.1 The byte-vector Module

**<byte-vector> Type**

- **Equivalent** limited(<vector>, of: <byte>)
- See also
  - <byte>

**<byte> Type**

- **Equivalent** limited(<integer>, min: 0, max: 255)

**byte-storage-address** Open Generic function

Returns the address of the raw byte storage of an object.

- **Signature** byte-storage-address (the-buffer) => (result-offset)
- **Parameters**
  - the-buffer – An instance of <object>.
- **Values**
  - result-offset – An instance of <machine-word>.
- See also
  - byte-storage-address(<buffer>)
  - byte-storage-address(<byte-string>)
  - byte-storage-address(<byte-vector>)
  - byte-storage-offset-address

**byte-storage-address(<byte-string>)** Sealed Method

Returns the address of the raw byte storage of a <byte-string>.

- See also
  - byte-storage-address

**byte-storage-address(<byte-vector>)** Sealed Method

Returns the address of the raw byte storage of a <byte-vector>.
See also

- `byte-storage-address`

**byte-storage-offset-address** Open Generic function

Returns the address of the raw byte storage of an object, with an offset.

**Signature** `byte-storage-offset-address (the-buffer data-offset) => (result-offset)`

**Parameters**

- `the-buffer` – An instance of `<object>`.
- `data-offset` – An instance of `<integer>`.

**Values**

- `result-offset` – An instance of `<machine-word>`.

See also

- `byte-storage-offset-address (<buffer>)`
- `byte-storage-offset-address (<byte-string>)`
- `byte-storage-offset-address (<byte-vector>)`
- `byte-storage-address`

**byte-storage-offset-address (<byte-string>)** Sealed Method

Returns the address of the raw byte storage of a `<byte-string>`, with an offset.

See also

- `byte-storage-offset-address`

**byte-storage-offset-address (<byte-vector>)** Sealed Method

Returns the address of the raw byte storage of a `<byte-vector>`, with an offset.

See also

- `byte-storage-offset-address`

**byte-vector-fill** Generic function

**Signature** `byte-vector-fill (target value) => (#rest results)`

**Parameters**

- `target` – An instance of `<object>`.
- `value` – An instance of `<object>`.

**Values**

- `#rest results` – An instance of `<object>`.

**byte-vector-fill (<byte-vector>, <integer>)** Sealed Method

**byte-vector-fill (<byte-vector>, <byte-character>)** Sealed Method

**byte-vector-ref** Function

**Signature** `byte-vector-ref (byte-vector index) => (#rest results)`

**Parameters**

- `byte-vector` – An instance of `<byte-vector>`.
- `index` – An instance of `<integer>`.

Chapter 7. The common-dylan Library
Values

• #rest results – An instance of <object>.

byte-vector-ref-setter Function

Signature byte-vector-ref-setter (value byte-vector index) => (#rest results)

Parameters

• value – An instance of <object>.
• byte-vector – An instance of <byte-vector>.
• index – An instance of <integer>.

Values

• #rest results – An instance of <object>.

copy-bytes Open Generic function

Signature copy-bytes (dst dst-start src src-start n) => ()

Parameters

• dst – An instance of <object>.
• dst-start – An instance of <object>.
• src – An instance of <object>.
• src-start – An instance of <object>.
• n – An instance of <object>.

copy-bytes (<sequence>, <integer>, <sequence>, <integer>, <integer>) Open Method
copy-bytes (<vector>, <integer>, <vector>, <integer>, <integer>) Open Method
copy-bytes (<string>, <integer>, <string>, <integer>, <integer>) Open Method
copy-bytes (<vector>, <integer>, <integer>, <integer>, <integer>) Open Method
copy-bytes (<byte-vector>, <integer>, <byte-vector>, <integer>, <integer>) Sealed Method
copy-bytes (<byte-string>, <integer>, <byte-vector>, <integer>, <integer>) Sealed Method
copy-bytes (<byte-vector>, <integer>, <byte-string>, <integer>, <integer>) Sealed Method
copy-bytes (<byte-string>, <integer>, <byte-string>, <integer>, <integer>) Sealed Method
copy-bytes (<byte-vector>, <integer>, <simple-object-vector>, <integer>, <integer>) Sealed Method
copy-bytes (<simple-object-vector>, <integer>, <byte-vector>, <integer>, <integer>) Sealed Method

hexstring (byte-vector) Sealed Method

Returns a string of lowercase hexadecimal digits representing the data.

Signature hexstring (data) => (result)

Parameters

• data – An instance of <byte-vector>.

Values

• result – An instance of <byte-string>.
See also

- from-hexstring(<byte-string>)

**from-hexstring(<byte-string>)** Sealed Method

Returns a <byte-vector> containing *data* interpreted as a hexadecimal representation of a series bytes.

**Signature**

from-hexstring (string) => (result)

**Parameters**

- **string** – An instance of <byte-string>.

**Values**

- **result** – An instance of <vector>.

See also

- hexstring(<byte-vector>)

### 7.2 The common-extensions Module

The *common-extensions* module contains a variety of useful basic extensions to the Dylan language and is exported from the *common-dylan* library. As a convenience, the *common-dylan* module re-exports everything from the *common-extensions* and *dylan* modules.

The extensions are:

- **Collection model extensions**: <stretchy-sequence>, <string-table>, difference, fill-table!, find-element, position, remove-all-keys!, define table, split, and join.

- **Condition system extensions**: <format-string-condition>, <simple-condition>, and condition-to-string.

- **Program constructs**: iterate and when.

- **Application development conveniences**:
  - debug-message
  - ignore, ignorable
  - $unsupplied, unsupplied?, unsupplied?, supplied?
  - $unfound unfound?, found?, unfound
  - one-of

- **Performance analysis**: timing, profiling.

- **Type conversion functions**: integer-to-string, string-to-integer, and float-to-string.

- **Byte storage access functions**: byte-storage-address, byte-storage-offset-address, which are re-exported from the byte-vector module.

**assert Statement Macro**

Signals an error if the expression passed to it evaluates to false.

**Macro Call**

```
assert *expression* *format-string* [*format-arg* ]* => *false*
```
assert *expression* => *false*

Parameters

- **expression** – A Dylan expression
- **format-string** – A Dylan expression
- **format-arg** – A Dylan expression

Values

- **false** – #f.

Discussion

Signals an error if *expression* evaluates to #f.

An assertion or “assert” is a simple tool for testing that conditions hold in program code.

The *format-string* is a format string as defined on page 112 of the DRM. If *format-string* is supplied, the error is formatted accordingly, along with any instances of *format-arg*.

If *expression* is not #f, assert does not evaluate *format-string* or any instances of *format-arg*.

See also

- **debug-assert**

<byte-character> Sealed Class

The class of 8-bit characters that instances of <byte-string> can contain.

Superclasses <character>

Discussion The class of 8-bit characters that instances of <byte-string> can contain.

concatenate! Open Generic function

A destructive version of the Dylan language’s concatenate; that is, one that might modify its first argument.

Signature  concatenate! sequence #rest more-sequences => result-sequence

Parameters

- **sequence** – An instance of <sequence>.
- **more-sequences** (#rest) – Instances of <sequence>.

Values

- **result-sequence** – An instance of <sequence>.

Discussion

A destructive version of the Dylan language’s concatenate; that is, one that might modify its first argument.

It returns the concatenation of one or more sequences, in a sequence that may or may not be freshly allocated. If result-sequence is freshly allocated, then, as for concatenate, it is of the type returned by type-for-copy of sequence.

Example

```
> define variable *x* = "great-";
"great-"
> define variable *y* = "abs";
"abs"
```

(continues on next page)
condition-to-string Open Generic function

Returns a string representation of a condition object.

**Signature**

```
condition-to-string  condition  =>  string
```

**Parameters**

- `condition` – An instance of `<condition>`.

**Values**

- `string` – An instance of `<string>`.

**Discussion**

Returns a string representation of a general instance of `<condition>`. There is a method on `<format-string-condition>` and method on `<type-error>`.

decompile Statement Macro

Signals an error if the expression passed to it evaluates to false — but only when the code is compiled in interactive development mode.

**Macro Call**

```
debug-assert  *expression*  *format-string*  [  *format-arg*  ]*  =>  *false*
```

**Parameters**

- `expression` – A Dylan expression `bnf`.
- `format-string` – A Dylan expression `bnf`.
- `format-arg` – A Dylan expression `bnf`.

**Values**

- `false` – `#f`.

**Discussion**

Signals an error if `expression` evaluates to false — but only when the code is compiled in debugging mode.

An assertion or “assert” is a simple and popular development tool for testing conditions in program code.

This macro is identical to `assert`, except that the assert is defined to take place only while debugging.

The Open Dylan compiler removes debug-assertions when it compiles code in “production” mode as opposed to “debugging” mode.

The `format-string` is a format string as defined on page 112 of the DRM.

**See also**

- `assert`
**debug-message Function**
Formats a string and outputs it to the debugger.

**Signature**
`debug-message format-string #rest format-args => ()`

**Parameters**
- `format-string` – An instance of `<string>`.
- `format-args (#rest)` – Instances of `<object>`.

**Discussion**
Formats a string and outputs it to the debugger.

The `format-string` is a format string as defined on page 112 of the DRM.

**default-handler (<warning>) Method**
Prints the message of a warning instance to the Open Dylan debugger window’s messages pane.

**Signature**
`default-handler warning => false`

**Parameters**
- `warning` – An instance of `<warning>`.

**Values**
- `false` – `#f`.

**Discussion**
Prints the message of a warning instance to the Open Dylan debugger window’s messages pane. It uses `debug-message`, to do so.

This method is a required, predefined method in the Dylan language, described on page 361 of the DRM as printing the warning’s message in an implementation-defined way. We document this method here because our implementation of it uses the function `debug-message`, which is defined in the `common-dylan` library. Thus to use this `default-handler` method on `<warning>`, your library needs to use the `common-dylan` library or a library that uses it, rather than simply using the Dylan library.

**Example**
In the following code, the signalled messages appear in the Open Dylan debugger window.

```dylan
define class <my-warning> (<warning>)
end class;

define method say-hello()
    format-out("hello there!\n");
    signal("help!");
    signal(make(<my-warning>));
    format-out("goodbye\n");
end method say-hello;

say-hello();
```

The following messages appear in the debugger messages pane:

- Application Dylan message: Warning: help!
- Application Dylan message: Warning: {<my-warning>}

Where `{<my-warning>}` means an instance of `<my-warning>`.

**See also**

7.2. The common-extensions Module
• debug-message.
• default-handler, page 361 of the DRM.

default-last-handler Function
Formats and outputs a Dylan condition using condition-to-string and passes control on to the next handler.

Signature  default-last-handler serious-condition next-handler => ()

Parameters
• serious-condition – A object of class <serious-condition>.
• next-handler – A function.

Discussion
A handler utility function defined on objects of class <serious-condition> that can be by bound dynamically around a computation via let handler or installed globally via define last-handler.

This function formats and outputs the Dylan condition serious-condition using condition-to-string from this library, and passes control on to the next handler.

This function is automatically installed as the last handler if your library uses the common-dylan library.

Example  The following form defines a dynamic handler around some body:

```dylan
let handler <serious-condition> = default-last-handler;
```

while the following form installs a globally visible last-handler:

```dylan
define last-handler <serious-condition> = default-last-handler;
```

See also
• define last-handler

define last-handler Defining Macro
Defines a “last-handler” to be used after any dynamic handlers and before calling default-handler.

Macro Call

```dylan
define last-handler (*condition*, #key *test*, *init-args*)
   = *handler* ;
define last-handler condition = handler;
define last-handler;
```

Parameters
• condition – A Dylan expression bnf. The class of condition for which the handler should be invoked.
• test – A Dylan expression bnf. A function of one argument called on the condition to test applicability of the handler.
• **init-args** – A Dylan expression *bnf*. A sequence of initialization arguments used to make an instance of the handler’s condition class.

• **handler** – A Dylan expression *bnf*. A function of two arguments, *condition* and *next-handler*, that is called on a condition which matches the handler’s condition class and test function.

**Discussion**

A last-handler is a global form of the dynamic handler introduced via `let handler`, and is defined using an identical syntax. The last handler is treated as a globally visible dynamic handler. During signalling if a last-handler has been installed then it is the last handler tested for applicability before `default-handler` is invoked. If a last-handler has been installed then it is also the last handler iterated over in a call to `do-handlers`.

The first two defining forms are equivalent to the two alternate forms of `let handler`. If more than one of these first defining forms is executed then the last one executed determines the installed handler. The current last-handler can be uninstalled by using the degenerate third case of the defining form, that has no condition description or handler function.

The intention is that libraries will install last handlers to provide basic runtime error handling, taking recovery actions such as quitting the application, trying to abort the current application operation, or entering a connected debugger.

**Example** The following form defines a last-handler function called `default-last-handler` that is invoked on conditions of class `<serious-condition>`:

```dylan
define last-handler <serious-condition>  
  = default-last-handler;
```

**See also**

• `win32-last-handler` in the *C FFI and Win32* library reference, under library *win32-user* and module *win32-default-handler*.

---

### define table Defining Macro

Defines a constant binding in the current module and initializes it to a new table object.

**Macro Call**

```dylan
define table *name* [ :: *type* ] = { [ *key* => *element* ]* }
```

**Parameters**

- **name** – A Dylan name *bnf*.
- **type** – A Dylan operand *bnf*. Default value: `<table>`.
- **key** – A Dylan expression *bnf*.
- **element** – A Dylan expression *bnf*.

**Discussion**

Defines a constant binding *name* in the current module, and initializes it to a new table object, filled in with the keys and elements specified.

If the argument *type* is supplied, the new table created is an instance of that type. Therefore *type* must be `<table>` or a subclass thereof. If *type* is not supplied, the new table created is an instance of a concrete subclass of `<table>`.

**Example**
```dylan
define table $colors :: <object-table>
  = { #'red' => $red,  
    #'green' => $green,  
    #'blue' => $blue };  
```

difference Open Generic function

Returns a sequence containing the elements of one sequence that are not members of a second.

Signature  difference sequence-1 sequence-2 #key test => result-sequence

Parameters

- **sequence-1** – An instance of `<sequence>`.
- **sequence-2** – An instance of `<sequence>`.
- **test** – An instance of `<function>`. Default value: `\==`.

Values

- **result-sequence** – An instance of `<sequence>`.

Discussion Returns a sequence containing the elements of `sequence-1` that are not members of `sequence-2`. You can supply a membership test function as `test`.

Example

```dylan
> difference(#(1,2,3), #(2,3,4));  
#(1)  
```

false-or Function

Returns a union type comprised of `singleton(#f)` and one or more types.

Signature  false-or type #rest more-types => result-type

Parameters

- **type** – An instance of `<type>`.
- **more-types** (#rest) – Instances of `<type>`.

Values

- **result-type** – An instance of `<type>`.

Discussion Returns a union type comprised of `singleton(#f), type, any other types passed as more-types`.

This function is useful for specifying slot types and function return values.

The expression

```dylan
false-or(*t-1*, *t-2*, ..)  
```

is type-equivalent to

```dylan
type-union(singleton(#f), *t-1*, *t-2*, ..)  
```

fill-table! Function

Fills a table with the keys and elements supplied.

Signature  fill-table! table keys-and-elements => table
Parameters

- **table** – An instance of `<table>`.
- **keys-and-elements** – An instance of `<sequence>`.

Values

- **table** – An instance of `<table>`.

Discussion

Modifies table so that it contains the keys and elements supplied in the sequence `keys-and-elements`.

This function interprets `keys-and-elements` as key-element pairs, that is, it treats the first element as a table key, the second as the table element corresponding to that key, and so on. The keys and elements should be suitable for `table`.

Because `keys-and-elements` is treated as a sequence of paired key-element values, it should contain an even number of elements; if it contains an odd number of elements, `fill-table!` ignores the last element (which would have been treated as a key).

**find-element Open Generic function**

Returns an element from a collection such that the element satisfies a predicate.

**Signature**

`find-element collection function #key skip failure => element`

Parameters

- **collection** – An instance of `<collection>`.
- **predicate** – An instance of `<function>`.
- **skip (#key)** – An instance of `<integer>`. Default value: 0.
- **failure (#key)** – An instance of `<object>`. Default value: #f.

Values

- **element** – An instance of `<object>`.

Discussion

Returns a collection element that satisfies `predicate`.

This function is identical to Dylan’s `find-key`, but it returns the element that satisfies `predicate` rather than the key that corresponds to the element.

**float-to-string Function**

Formats a floating-point number to a string.

**Signature**

`float-to-string float => string`

Parameters

- **float** – An instance of `<float>`.

Values

- **string** – An instance of `<string>`.

Discussion

Formats a floating-point number to a string. It uses scientific notation where necessary.

**<format-string-condition> Instantiable Sealed Class**

The class of conditions that take a format string.

**Superclasses**

- `<condition>`
Discussion

The class of conditions that take a format string, as defined by the DRM.

It is the superclass of Dylan’s `<simple-condition>`.

See also

- The `Format module` in the `IO library`.

**ignore Function**

A compiler directive that tells the compiler it must not issue a warning if its argument is bound but not referenced.

**Signature**

ignore variable => ()

**Parameters**

- `variable` – A Dylan variable-name `bnf`.

**Discussion**

When the compiler encounters a variable that is bound but not referenced, it normally issues a warning. The `ignore` function is a compiler directive that tells the compiler it must not issue this warning if `variable` is bound but not referenced. The `ignore` function has no run-time cost.

The `ignore` function is useful for ignoring arguments passed to, or values returned by, a function, method, or macro. The function has the same extent as a `let`; that is, it applies to the smallest enclosing implicit body.

Use `ignore` if you never intend to reference `variable` within the extent of the `ignore`. The compiler will issue a warning to tell you if your program violates the `ignore`. If you are not concerned about the `ignore` being violated, and do not wish to be warned if violation occurs, use `ignorable` instead.

**Example**

This function ignores some of its arguments:

```dylan
define method foo (x ::<integer>, #rest args)
    ignore(args);
    ...
end
```

Here, we use `ignore` to ignore one of the values returned by `fn`:

```dylan
let (x,y,z) = fn();
ignore(y);
```

See also

- `ignorable`

**ignorable Function**

A compiler directive that tells the compiler it need not issue a warning if its argument is bound but not referenced.

**Signature**

ignorable variable => ()

**Parameters**

- `variable` – A Dylan variable-name `bnf`.

**Discussion**

When the compiler encounters a variable that is bound but not referenced, it normally issues a warning. The `ignorable` function is a compiler directive that tells the compiler it need not issue this warning if `variable` is bound but not referenced. The `ignorable` function has no run-time cost.
The `ignorable` function is useful for ignoring arguments passed to, or values returned by, a function, method, or macro. The function has the same extent as a `let`; that is, it applies to the smallest enclosing implicit body.

The `ignorable` function is similar to `ignore`. However, unlike `ignore`, it does not issue a warning if you subsequently reference variable within the extent of the `ignorable` declaration. You might prefer `ignorable` to `ignore` if you are not concerned about such violations and do not wish to be warned about them.

**Example**  This function ignores some of its arguments:

```dylan
define method foo (x ::<integer>, #rest args)
  ignorable(args);
  ...
end
```

Here, we use `ignorable` to ignore one of the values returned by `fn`:

```dylan
let (x,y,z) = fn();
ignorable(y);
```

**See also**
- `ignore`

**integer-to-string Function**

Returns a string representation of an integer.

**Signature**  `integer-to-string` `integer` #key `base` `size` `fill` `=> string`

**Parameters**
- `integer` – An instance of `<integer>`.
- `base` – An instance of `<integer>` (default 10).
- `size` – An instance of `<integer>` (default 0).
- `fill` – An instance of `<character>` (default 0).

**Values**
- `string` – An instance of `<byte-string>`.

**Discussion**  Returns a string representation of `integer` in the given `base`, which must be between 2 and 36. The size of the string is right-aligned to `size`, and it is filled with the `fill` character. If the string is already larger than `size` then it is not truncated.

**iterate Statement Macro**

Iterates over a body.

**Macro Call**

```dylan
iterate *name* ({{*argument* = *init-value*}*}) [ *body* ] end [ iterate ]
```

**Parameters**
- `name` – A Dylan variable-name `bnf`.
- `argument` – A Dylan variable-name `bnf`.
- `init-value` – A Dylan expression `bnf`.

---

7.2. The common-extensions Module
• **body** – A Dylan body *bnf*.

**Values**

• **value** – Zero or more instances of `<object>`.

**Discussion**

Defines a function that can be used to iterate over a body. It is similar to `for`, but allows you to control when iteration will occur.

It creates a function called *name* which will perform a single step of the iteration at a time; *body* can call *name* whenever it wants to iterate another step. The form evaluates by calling the new function with the initial values specified.

**Example**

```
iterate recurse (x = 5)
  if (x < 2) x
  else x * recurse(x - 1)
end
```

**one-of Function**

Returns a union type comprised of singletons formed from its arguments.

**Signature**  
`one-of object #rest more-objects => type`

**Parameters**

• **object** – An instance of `<object>`.

• **more-objects (#rest)** – Instances of `<object>`.

**Values**

• **type** – An instance of `<type>`.

**Discussion**

Returns a union type comprised of `singleton(object)` and the singletons of any other objects passed with *more-object*.

```
one-of(x, y, z)
```

Is a type expression that is equivalent to

```
type-union(singleton(x), singleton(y), singleton(z))
```

**position**

Open Generic function

Returns the key at which a particular value occurs in a sequence.

**Signature**  
`position sequence target #key test start end skip count => position`

**Parameters**

• **sequence** – An instance of `<sequence>`.

• **target** – An instance of `<object>`.

• **test (#key)** – An instance of `<function>`. Default value: `\==`.

• **start (#key)** – An instance of `<integer>`. Default value: `0`.

• **end (#key)** – An instance of `<object>`. Default value: `#f`.

• **skip (#key)** – An instance of `<integer>`. Default value: `0`. 
• \texttt{count(#key)} – An instance of \texttt{<object>}. Default value: \#f.

Values

• \texttt{position} – An instance of \texttt{false-or(<integer>)}.

Discussion

Returns the position at which \texttt{target} occurs in \texttt{sequence}.

If \texttt{test} is supplied, \texttt{position} uses it as an equivalence predicate for comparing \texttt{sequence}’s elements to \texttt{target}. It should take two objects and return a boolean. The default predicate used is \texttt{\textbackslash ==}.

The \texttt{skip} argument is interpreted as it is by Dylan’s \texttt{find-key} function: \texttt{position} ignores the first \texttt{skip} elements that match \texttt{target}, and if \texttt{skip} or fewer elements satisfy \texttt{test}, it returns \#f.

The \texttt{start} and \texttt{end} arguments indicate, if supplied, which subrange of the \texttt{sequence} is used for the search.

\texttt{remove-all-keys!} Open Generic function

Removes all keys in a mutable collection, leaving it empty.

Signature \texttt{remove-all-keys! mutable-collection => ()}

Parameters

• \texttt{mutable-collection} – An instance of \texttt{<mutable-collection>}.

Discussion Modifies \texttt{mutable-collection} by removing all its keys and leaving it empty. There is a predefined method on \texttt{<table>}.

\texttt{<simple-condition>} Instantiable Sealed Class

The class of simple conditions.

Superclasses \texttt{<format-string-condition>}

Discussion The class of simple conditions. It is the superclass of \texttt{<simple-error>}, \texttt{<simple-warning>}, and \texttt{<simple-restart>}.

Operations

• \texttt{condition-format-string}

• \texttt{condition-format-args}

\texttt{<stretchy-sequence>} Open Abstract Class

The class of stretchy sequences.

Superclasses \texttt{<sequence>, <stretchy-collection>}

Discussion The class of stretchy sequences.

\texttt{<string-table>} Instantiable Sealed Class

The class of tables that use strings for keys.

Superclasses \texttt{<table>}

Discussion

The class of tables that use instances of \texttt{<string>} for their keys. It is an error to use a key that is not an instance of \texttt{<string>}

Keys are compared with the equivalence predicate \texttt{\textbackslash =}.

The elements of the table are instances of \texttt{<object>}.  

7.2. The common-extensions Module
It is an error to modify a key once it has been used to add an element to a <string-table>. The effects of modification are not defined.

Note: This class is also exported from the table-extensions module of the table-extensions library.

String-to-Integer Function

Returns the integer represented by its string argument, or by a substring of that argument, in a number base between 2 and 36.

**Signature**

```
string-to-integer string #key base start end default => integer next-key
```

**Parameters**

- **string** – An instance of <byte-string>.
- **base (#key)** – An instance of <integer>. Default value: 10.
- **start (#key)** – An instance of <integer>. Default value: 0.
- **end (#key)** – An instance of <integer>. Default value: sizeof(*string*).
- **default (#key)** – An instance of <integer>. Default value: $unsupplied.

**Values**

- **integer** – An instance of <integer>.
- **next-key** – An instance of <integer>.

**Discussion**

Returns the integer represented by the characters of string in the number base base, where base is between 2 and 36. You can constrain the search to a substring of string by giving values for start and end.

This function returns the next key beyond the last character it examines.

If there is no integer contained in the specified region of the string, this function returns default, if specified. If you do not give a value for default, this function signals an error.

This function is similar to C’s `strtod` function.

Subclass Function

Returns a type representing a class and its subclasses.

**Signature**

```
subclass class => subclass-type
```

**Parameters**

- **class** – An instance of <class>.

**Values**

- **subclass-type** – An instance of <type>.

**Discussion**

Returns a type that describes all the objects representing subclasses of the given class. We term such a type a subclass type.

The subclass function is allowed to return an existing type if that type is type equivalent to the subclass type requested.
Without `subclass`, methods on generic functions (such as Dylan’s standard `make` and `as`) that take types as arguments are impossible to reuse without resorting to ad hoc techniques. In the language defined by the DRM, the only mechanism available for specializing such methods is to use singleton types. A singleton type specializer used in this way, by definition, gives a method applicable to exactly one type. In particular, such methods are not applicable to subtypes of the type in question. In order to define reusable methods on generic functions like this, we need a type which allows us to express applicability to a type and all its subtypes.

For an object $O$ and class $Y$, the following `instance?` relationship applies:

**INSTANCE-1:** `instance?(*O*, subclass(*Y*))` True if and only if $O$ is a class and $O$ is a subclass of $Y$.

For classes $X$ and $Y$ the following `subtype?` relationships hold (note that a rule applies only when no preceding rule matches):

**SUBTYPE-1:** `subtype?(subclass(*X*), subclass(*Y*))` True if and only if $X$ is a class and $X$ is a subclass of $Y$.

**SUBTYPE-2:** `subtype?(singleton(*X*), subclass(*Y*))` True if and only if $X$ is a class and $X$ is a subclass of $Y$.

**SUBTYPE-3:** `subtype?(subclass(*X*), singleton(*Y*))` Always false.

**SUBTYPE-4:** `subtype?(subclass(*X*), *Y*)` where $Y$ is not a subclass type. True if $Y$ is `<class>` or any proper supertype of `<class>` (including `<object>`, any implementation-defined supertypes, and unions involving any of these). There may be other implementation-defined combinations of types $X$ and $Y$ for which this is also true.

**SUBTYPE-5:** `subtype?(*X*, subclass(*Y*))` where $X$ is not a subclass type. True if $Y$ is `<object>` or any proper supertype of `<object>` and $X$ is a subclass of `<class>`.

Note that by subclass relationships `SUBTYPE-4` and `SUBTYPE-5`, we get this correspondence: `<class>` and `subclass(<object>)` are type equivalent.

Where the `subtype?` test has not been sufficient to determine an ordering for a method’s argument position, the following further method-ordering rules apply to cases involving subclass types (note that a rule applies only when no preceding rule matches):

- **SPECIFICITY+1**. `subclass(*X*)` precedes `subclass(*Y*)` when the argument is a class $C$ and $X$ precedes $Y$ in the class precedence list of $C$.

- **SPECIFICITY+2**. `subclass(*X*)` always precedes $Y$, $Y$ not a subclass type. That is, applicable subclass types precede any other applicable class-describing specializer.

The constraints implied by sealing come by direct application of sealing rules 1–3 (see page 136 of the DRM) and the following disjointness criteria for subclass types (note that a rule applies only when no preceding rule matches):

- **DISJOINTNESS+1**. A subclass type `subclass(*X*)` and a type $Y$ are disjoint if $Y$ is disjoint from `<class>`, or if $Y$ is a subclass of `<class>` without instance classes that are also subclasses of $X$.

- **DISJOINTNESS+2**. Two subclass types `subclass(*X*)` and `subclass(*Y*)` are disjoint if the classes $X$ and $Y$ are disjoint.

- **DISJOINTNESS+3**. A subclass type `subclass(*X*)` and a singleton type `singleton(*O*)` are disjoint unless $O$ is a class and $O$ is a subclass of $X$.

The guiding principle behind the semantics is that, as far as possible, methods on classes called with an instance should behave isomorphically to corresponding methods on corresponding subclass types called with the class of that instance. So, for example, given the heterarchy:
and methods:

```dylan
method foo (<A>)
method foo (<B>)
method foo (<C>)
method foo (<D>)

method foo-using-type (subclass(<A>))
method foo-using-type (subclass(<B>))
method foo-using-type (subclass(<C>))
method foo-using-type (subclass(<D>))
```

that for a direct instance \textit{D1} of \texttt{<D>}:

```dylan
foo-using-type(<D>)
```

should behave analogously to:

```dylan
foo(D1)
```

with respect to method selection.

\textbf{Example}

```dylan
define class <A> (<object>) end;
define class <B> (<A>) end;
define class <C> (<A>) end;
define class <D> (<B>, <C>) end;

define method make (class :: subclass(<A>), #key)
  print("Making an <A>");
  next-method();
end method;
define method make (class :: subclass(<B>), #key)
  print("Making a <B>");
  next-method();
end method;
define method make (class :: subclass(<C>), #key)
  print("Making a <C>");
  next-method();
end method;
define method make (class :: subclass(<D>), #key)
  print("Making a <D>");
  next-method();
end method;
```
$\texttt{unfound Constant}$

A unique value that can be used to indicate that a search operation failed.

**Type** &lt;list&gt;

**Value** A unique value.

**Discussion** A unique value that can be used to indicate that a search operation failed.

**Example**

```dylan
if (unfound?(element(section-index-table, section-name, default: $unfound)))
    section-index-table[section-name] := section-index-table.size + 1;
    write-record(stream, "SECTIONNAME", section-name);
end if;
```

See also

- found?
- unfound?
- unfound

$\texttt{unfound Function}$

Returns the unique “unfound” value, $\texttt{$unfound}$.  

**Signature** `unfound () => unfound-marker`

**Values**

- **unfound-marker** – The value $\texttt{$unfound}$.  

**Discussion**

Returns the unique “unfound” value, $\texttt{$unfound}$.  

**Example** See $\texttt{$unfound}$.  

See also

- found?
- unfound?
- $\texttt{$unfound}$

$\texttt{found? Function}$

Returns true if `object` is not equal to $\texttt{$unfound}$, and false otherwise.

**Signature** `found? object => boolean`

**Parameters**

- **object** – An instance of &lt;object&gt;.

**Values**
boolean – An instance of <boolean>.

Discussion

Returns true if object is not equal to $unfound, and false otherwise. It uses \= as the equivalence predicate.

Example See $unfound.

See also

• $unfound
• unfound?
• unfound

unfound? Function

Returns true if its argument is equal to the unique “unfound” value, $unfound, and false if it is not.

Signature unfound? object => unfound?

Parameters

• object – An instance of <object>.

Values

• unfound? – An instance of <boolean>.

Discussion Returns true if object is equal to the unique “unfound” value, $unfound, and false if it is not. It uses \= as the equivalence predicate.

Example See $unfound.

See also

• found?
• $unfound
• unfound

$unsupplied Constant

A unique value that can be used to indicate that a keyword was not supplied.

Type <list>

Value A unique value.

Discussion A unique value that can be used to indicate that a keyword was not supplied.

Example

```dylan
define method find-next-or-previous-string
  (frame :: <editor-state-mixin>,
   #key reverse? = $unsupplied)
  => ()
  let editor :: <basic-editor> = frame-editor(frame);
  let reverse? = if (supplied?(reverse?))
    reverse?
  else
    editor-reverse-search?(editor)
  end;
```
See also

- $supplied?
- unsupplied
- unsupplied?

**unsupplied Function**

Returns the unique “unsupplied” value, $unsupplied.

**Signature**

unsupplied () => unsupplied-marker

**Values**

- unsupplied-marker – The value $unsupplied.

**Discussion**

Returns the unique “unsupplied” value, $unsupplied.

**Example**

See $unsupplied.

See also $supplied?, unsupplied?

**supplied? Function**

Returns true if its argument is not equal to the unique “unsupplied” value, $unsupplied, and false if it is.

**Signature**

supplied? object => supplied?

**Parameters**

- object – An instance of <object>.

**Values**

- supplied? – An instance of <boolean>.

**Discussion**

Returns true if object is not equal to the unique “unsupplied” value, $unsupplied, and false if it is. It uses \= as the equivalence predicate.

**Example**

See $unsupplied.

See also $unsupplied, unsupplied?

**unsupplied? Function**

Returns true if its argument is equal to the unique “unsupplied” value, $unsupplied, and false if it is not.

**Signature**

unsupplied? value => boolean

**Parameters**

- value – An instance of <object>.

**Values**
• boolean – An instance of `<boolean>`.

Discussion Returns true if its argument is equal to the unique “unsupplied” value, `$unsupplied`, and false if it is not. It uses `\=` as the equivalence predicate.

Example See `$unsupplied`.

See also
• supplied?
• `$unsupplied`
• unsupplied

**when Statement Macro**
Executes an implicit body if a test expression is true, and does nothing if the test is false.

Macro Call

```
when (*test*) [ *consequent* ] end [ when ]
```

Parameters

• test – A Dylan expression bnf.
• consequent – A Dylan body bnf.

Values

• value – Zero or more instances of `<object>`.

Discussion

Executes consequent if test is true, and does nothing if test is false.

This macro behaves identically to Dylan’s standard if statement macro, except that there is no alternative flow of execution when the test is false.

Example

```
when (x < 0)
  ~ x;
end;
```

**split Function**
Split a sequence (e.g., a string) into subsequences delineated by a given separator.

Signature split sequence separator #key start end count remove-if-empty? => parts

Parameters

• sequence – An instance of `<sequence>`.
• separator – An instance of `<object>`.
• start (#key) – An instance of `<integer>`. Default value: 0.
• end (#key) – An instance of `<integer>`. Default value: sequence.size.
• count (#key) – An instance of `<integer>`. Default value: no limit.
• remove-if-empty? (#key) – An instance of `<boolean>`. Default value: #f.

Values

• parts – An instance of `<sequence>`.
Discussion

Splits `sequence` into subsequences, splitting at each occurrence of `separator`. The `sequence` is searched from left to right, starting at `start` and ending at `end - 1`.

The resulting parts sequence is limited in size to `count` elements.

If `remove-if-empty?` is true, the result will not contain any subsequences that are empty.

There are methods specialized on various types of `separator`. The most basic `separator` type is `<function>`, with which all of the others may be implemented.

\[
\text{split}(\text{seq} :: <\text{sequence}>, \text{separator} :: <\text{function}>, ...) \]

This is in some sense the most basic method, since others can be implemented in terms of it. The ‘separator’ function must accept three arguments:

1. the sequence in which to search for a separator,
2. the start index in that sequence at which to begin searching, and
3. the index at which to stop searching (exclusive).

The ‘separator’ function must return `#f` to indicate that no separator was found, or two values:

1. the start index of the separator in the sequence and
2. the index of the first element after the end of the separator.

It is an error for the returned start and end indices to be equal since this is equivalent to splitting on an empty separator, which is undefined. It is undefined what happens if the return values are outside the `[start, end)` range passed to the separator function.

The initial start and end indices passed to the separator function are the same as the ‘start’ and ‘end’ arguments passed to this method. The ‘separator’ function should stay within the given bounds whenever possible. (In particular it may not always be possible when the separator is a regex.)

\[
\text{split}(\text{seq} :: <\text{sequence}>, \text{separator} :: <\text{object}> , \text{#key} \text{ test} = \==, ...) \]

Splits ‘seq’ around occurrences of ‘separator’ using ‘test’ to check for equality. This method handles the relatively common case where ‘seq’ is a string and ‘separator’ is a character.

\[
\text{split}(\text{seq} :: <\text{sequence}>, \text{separator} :: <\text{sequence}> , \text{#key} \text{ test} = \==, ...) \]

Splits ‘seq’ around occurrences of the ‘separator’ subsequence. This handles the relatively common case where ‘seq’ and ‘separator’ are both strings.

Note that if you want to use ‘split’ to find a sequence which is a single element of another sequence it won’t work because this method is more specific than the previous one. That is considered to be an uncommon case and can be handled by using the method on `<function>`.

Example

\[
\text{split}("a.b.c", ".") => \#("a", "b", "c")
\]

See also

* `join`

**join**

Generic function

Join several sequences (e.g. strings) together, including a separator between each pair of adjacent sequences.

**Signature**

\[
\text{join} \text{ sequences separator} \text{#key key conjunction} => \text{joined}
\]

**Parameters**
• **sequences** – An instance of `<sequence>`.
• **separator** – An instance of `<sequence>`.
• **key (#key)** – Transformation to apply to each item. Default value: `identity`.
• **conjunction (#key)** – Last separator. Default value: `#f`

**Values**

• **joined** – An instance of `<sequence>`.

**Discussion**

Join `sequences` together, including `separator` between each sequence.

If the first argument is empty, an empty sequence of type `type-for-copy(separator)` is returned. If `sequences` is of size one, the first element is returned. Otherwise, the resulting `joined` sequence will be of the same type as `sequences`.

Every element in `sequences` is transformed by `key`, which is a function that must accept one argument.

If `conjunction` is not false, it is used instead of `separator` to join the last pair of elements in `sequences`.

**Example**

```dylan
join(range(from: 1, to: 3), ", ",
    key: integer-to-string, conjunction: " and ")
=> "1, 2 and 3"
```

**See also**

• `join`
• `split`

**join(<sequence>, <sequence>) Method**

Join several sequences together, including a separator between each pair of adjacent sequences.

**Signature**

```
join sequences separator #key key conjunction => joined
```

**Parameters**

• **items** – An instance of `<sequence>`.
• **separator** – An instance of `<sequence>`.
• **key (#key)** – Transformation to apply to each item. An instance of `<function>`.
• **conjunction (#key)** – Last separator. An instance of `false-or(<sequence>)`.

**Values**

• **joined** – An instance of `<sequence>`.

**See also**

• `join`
• `split`
7.3 The machine-words Module

This document describes a number of extensions to the Dylan language which are available from the Dylan library.

Throughout this document, arguments are instances of the class specified by the argument name, unless otherwise noted. Thus, the arguments `machine-word` and `integer` are instances of `<machine-word>` and `<integer>`, respectively.

The class `<machine-word>` is a sealed subclass of `<object>`, defined in the Dylan library. The class `<machine-word>` represents a limited range of integral values. The representation used has the natural size suggested by the implementation architecture. (When running a 32 bit OS, a `<machine-word>` is 32 bits wide. When running a 64 bit OS, then `<machine-word>` is 64 bits wide.) The class `<machine-word>` is disjoint from all other classes specified by the Dylan language.

The `\==` function compares instances of `<machine-word>` by value.

7.3.1 Useful functions from the Dylan module

This section describes additional methods defined in the Dylan module that pertain to `<machine-word>`. Note that this section only describes extensions to the Dylan library; for complete descriptions, you should also refer to the Dylan Reference Manual.

Note that the Common Dylan library also has these extensions because it uses the Dylan library.

odd? Function

Signature  odd? m => r

Parameters

• m – An instance of `<machine-word>`

Values

• r – An instance of `<boolean>`

even? Function

Signature  even? m => r

Parameters

• m – An instance of `<machine-word>`

Values

• r – An instance of `<boolean>`

zero? Function

Signature  zero? m => r

Parameters

• m – An instance of `<machine-word>`

Values

• r – An instance of `<boolean>`

Note: Cannot be used as the name of a result. It is not a valid Dylan name.
positive? Function
Signature positive? m => r
Parameters
• m – An instance of <machine-word>
Values
• r – An instance of <boolean>
negative? Function
Signature negative? m => r
Parameters
• m – An instance of <machine-word>
Values
• r – An instance of <boolean>
These functions return a result based on interpreting m as a signed integer value.

= Function
Signature = m1 m2 => r
Signature = i1 m2 => r
Signature = m1 i2 => r
Parameters
• m1 – An instance of <machine-word>
• m2 – An instance of <machine-word>
• i1 – An instance of <abstract-integer>
• i2 – An instance of <abstract-integer>
Values
• r – An instance of <boolean>
Discussion The comparison is performed with the <machine-word> arguments interpreted as signed integer values.

< Function
Signature < m1 m2 => r
Signature < i1 m2 => r
Signature < m1 i2 => r
Parameters
• m1 – An instance of <machine-word>
• m2 – An instance of <machine-word>
• i1 – An instance of <abstract-integer>
• i2 – An instance of <abstract-integer>
Values
• r – An instance of <boolean>

Discussion The comparison is performed with the <machine-word> arguments interpreted as signed integer values.

as Function

Signature as t == <integer> m => r

Parameters

• m – An instance of <machine-word>

Values

• r – An instance of <integer>

Discussion The result is an <integer> with the same value as m when interpreted as a signed integer value. An error is signaled if the value of m cannot be represented as an instance of <integer>.

as Function

Signature as t == <abstract-integer> m => r

Parameters

• m – An instance of <machine-word>

Values

• r – An instance of <abstract-integer>

Discussion The result is an <abstract-integer> with the same value as m when interpreted as a signed integer value.

(The uses for an instance of <abstract-integer> that is not also an instance of <integer> are rather limited without the Generic-Arithmetic library.)

as Function

Signature as t == <machine-word> i => r

Parameters

• i – An instance of <abstract-integer>

Values

• r – An instance of <machine-word>

Discussion If the value of i is outside the machine word range, then the result consists of the low $machine-word-size bits of the twos-complement representation of i. If any of the discarded bits differ from the sign of i, then an error is signaled.

limited Function

Signature limited t == <machine-word> #key signed? min max => r

Parameters

• signed? (#key) – An instance of <boolean>. Defaults to #t
• min (#key) – An instance of <machine-word>
• max (#key) – An instance of <machine-word>

7.3. The machine-words Module
Values

- \( r \) – An instance of \(<\text{type}>\)

Discussion

If the \texttt{signed?} argument is true (the default) then the \texttt{min} and \texttt{max} arguments are interpreted as signed values. When \texttt{signed?} is false, the \texttt{min} and \texttt{max} arguments are interpreted as unsigned values. The default value for each of \texttt{min} and \texttt{max} depends on the value of \texttt{signed?}. The defaults are taken from the corresponding minimum and maximum machine word values (see \texttt{$\text{maximum}\text{-}\text{signed}\text{-}\text{machine}\text{-}\text{word}$} and related constants below).

For convenience, the values of \texttt{min} and/or \texttt{max} may also be instances of \(<\text{abstract}\text{-}\text{integer}>\), in which case they are coerced to instances of \(<\text{machine}\text{-}\text{word}>\) as if by using \texttt{as}.

### 7.3.2 The machine-words Module

This section contains a reference entry for each item exported from the Machine-Words module, which is exported by the Common Dylan library.

<\text{machine}\text{-}\text{word}> Sealed Class

- **Summary** The class of objects that can represent a limited range of integral values.
- **Superclasses** \(<\text{object}>\)
- **Discussion** The class \(<\text{machine}\text{-}\text{word}>\) represents a limited range of integral values. The representation used has the natural size suggested by the implementation architecture. The class \(<\text{machine}\text{-}\text{word}>\) is disjoint from all other classes specified by the Dylan language.
- **Operations** The \(<\text{machine}\text{-}\text{word}>\) class provides the operations described below and in \textit{Useful functions from the Dylan module}.

**Variables**

The following variables are exported from the Machine-Words module.

- **\$machine\text{-}word\text{-}size Constant**
  - **Type** \(<\text{integer}>\)
  - **Discussion** The number of bits in the representation of a \(<\text{machine}\text{-}\text{word}>\).

- **\$maximum\text{-}signed\text{-}machine\text{-}word Constant**
  - **Type** \(<\text{machine}\text{-}\text{word}>\)
  - **Discussion** The largest machine word, when interpreted as a signed integer value.

- **\$minimum\text{-}signed\text{-}machine\text{-}word Constant**
  - **Type** \(<\text{machine}\text{-}\text{word}>\)
  - **Discussion** The smallest machine word, when interpreted as a signed integer value.

- **\$maximum\text{-}unsigned\text{-}machine\text{-}word Constant**
  - **Type** \(<\text{machine}\text{-}\text{word}>\)
  - **Discussion** The largest machine word, when interpreted as an unsigned integer value.

- **\$minimum\text{-}unsigned\text{-}machine\text{-}word Constant**
Type  <machine-word>

Discussion The smallest machine word, when interpreted as an unsigned integer value.

**as-unsigned Function**

**Signature**  
\[ \text{as-unsigned } t \ m \Rightarrow \text{result} \]

**Parameters**
- \(t\) – A type
- \(m\) – An instance of <machine-word>

**Values**
- \(\text{result}\) – An instance of \(t\)

**Discussion** The value of \(m\) is interpreted as an unsigned value and converted to an instance of \(<abstract-integer>\), then the result of that conversion is converted to type \(t\) using as.

**Basic and signed single word operations**

For all of the following functions, all arguments that are specified as being specialized to <machine-word> accept an instance of <abstract-integer>, which is then coerced to a <machine-word> before performing the operation.

**%logior Function**

**Signature**  
\[ \%\text{logior} \ #\text{rest} \text{ machine-words} \Rightarrow r \]

**Parameters**
- \(\text{machine-words}(\#\text{rest})\) – An instance of <machine-word>

**Values**
- \(r\) – An instance of <machine-word>

**%logxor Function**

**Signature**  
\[ \%\text{logxor} \ #\text{rest} \text{ machine-words} \Rightarrow r \]

**Parameters**
- \(\text{machine-words}(\#\text{rest})\) – An instance of <machine-word>

**Values**
- \(r\) – An instance of <machine-word>

**%logand Function**

**Signature**  
\[ \%\text{logand} \ #\text{rest} \text{ machine-words} \Rightarrow r \]

**Parameters**
- \(\text{machine-words}(\#\text{rest})\) – An instance of <machine-word>

**Values**
- \(r\) – An instance of <machine-word>

**%lognot Function**

**Signature**  
\[ \%\text{lognot} \ m \Rightarrow r \]

**Parameters**
These four functions have the same semantics as \texttt{logior, logxor, logand, and lognot} in the Dylan library, but they operate on \texttt{<machine-word>} s instead of \texttt{<integer>} s.

\%\texttt{logbit?} Function

\textbf{Signature} \ \%\texttt{logbit?} index m \Rightarrow \texttt{set?}

\textbf{Parameters}
- \texttt{index} – An instance of \texttt{<integer>}
- \texttt{m} – An instance of \texttt{<machine-word>}

\textbf{Values}
- \texttt{set?} – An instance of \texttt{<boolean>}

\textbf{Discussion} Returns true iff the indexed bit (zero based, counting from the least significant bit) of \texttt{m} is set. An error is signaled unless \(0 \leq \text{index} < \$\text{machine-word-size}\).

\%\texttt{count-low-zeros} Function

\textbf{Signature} \ %\texttt{count-low-zeros} m \Rightarrow c

\textbf{Parameters}
- \texttt{m} – An instance of \texttt{<machine-word>}

\textbf{Values}
- \texttt{c} – An instance of \texttt{<integer>}

\textbf{Discussion} Returns the number of consecutive zero bits in \texttt{m} counting from the least significant bit.

\textbf{Note:} This is the position of the least significant non-zero bit in \texttt{m}. So if \(i\) is the result, then \%\texttt{logbit?}(i, m) is true, and for all values of \(j\) such that \(0 \leq j < i\), \%\texttt{logbit?}(j, m) is false.

\%\texttt{count-high-zeros} Function

\textbf{Signature} \ %\texttt{count-high-zeros} m \Rightarrow c

\textbf{Parameters}
- \texttt{m} – An instance of \texttt{<machine-word>}
- \texttt{c} – An instance of \texttt{<integer>}

\textbf{Discussion} Returns the number of consecutive zero bits in \texttt{m} counting from the most significant bit.

\textbf{Note:} The position of the most significant non-zero bit in \texttt{m} can be computed by subtracting this result from \$\text{machine-word-size} - 1. So if \(i\) is the result and \(p = ($\text{machine-word-size} - i - 1)\), then \%\texttt{logbit?}(p, m) is true, and for all values of \(j\) such that \(p < j < $\text{machine-word-size}+, \%\texttt{logbit?}(j, m)\) is false.

\%\texttt{count-ones} Function

\textbf{Signature} \ %\texttt{count-ones} m \Rightarrow c
Parameters

- \( m \) – An instance of \(<machine-word>\).
- \( c \) – An instance of \(<integer>\).

Discussion Returns the number of bits in \( m \) which have been set to 1.

%+ Function

Signature  
%+ m1 m2 => sum overflow?

Parameters

- \( m1 \) – An instance of \(<machine-word>\)
- \( m2 \) – An instance of \(<machine-word>\)

Values

- \( \text{sum} \) – An instance of \(<machine-word>\)
- \( \text{overflow}? \) – An instance of \(<boolean>\)

Discussion Signed addition.

%- Function

Signature  
%- m1 m2 => difference overflow?

Parameters

- \( m1 \) – An instance of \(<machine-word>\)
- \( m2 \) – An instance of \(<machine-word>\)

Values

- \( \text{difference} \) – An instance of \(<machine-word>\)
- \( \text{overflow}? \) – An instance of \(<boolean>\)

Discussion Signed subtraction.

%* Function

Signature  
%* m1 m2 => low high overflow?

Parameters

- \( m1 \) – An instance of \(<machine-word>\)
- \( m2 \) – An instance of \(<machine-word>\)

Values

- \( \text{low} \) – An instance of \(<machine-word>\)
- \( \text{high} \) – An instance of \(<machine-word>\)
- \( \text{overflow}? \) – An instance of \(<boolean>\)

Discussion Signed multiplication. The value of \( \text{overflow}? \) is false iff the \( \text{high} \) word result is a sign extension of the \( \text{low} \) word result.

%floor/ Function

Signature  
%floor/ dividend divisor => quotient remainder

Parameters
• **dividend** – An instance of `<machine-word>`
• **divisor** – An instance of `<machine-word>`

Values
• **quotient** – An instance of `<machine-word>`
• **remainder** – An instance of `<machine-word>`

%ceiling/ Function

Signature  %ceiling/ dividend divisor => quotient remainder

Parameters
• **dividend** – An instance of `<machine-word>`
• **divisor** – An instance of `<machine-word>`

Values
• **quotient** – An instance of `<machine-word>`
• **remainder** – An instance of `<machine-word>`

%round/ Function

Signature  %round/ dividend divisor => quotient remainder

Parameters
• **dividend** – An instance of `<machine-word>`
• **divisor** – An instance of `<machine-word>`

Values
• **quotient** – An instance of `<machine-word>`
• **remainder** – An instance of `<machine-word>`

%truncate/ Function

Signature  %truncate/ dividend divisor => quotient remainder

Parameters
• **dividend** – An instance of `<machine-word>`
• **divisor** – An instance of `<machine-word>`

Values
• **quotient** – An instance of `<machine-word>`
• **remainder** – An instance of `<machine-word>`

%divide Function

Signature  %divide/ dividend divisor => quotient remainder

Parameters
• **dividend** – An instance of `<machine-word>`
• **divisor** – An instance of `<machine-word>`

Values
• **quotient** – An instance of `<machine-word>`
The functions \%divide, \%floor/, \%ceiling/, \%round/, and \%truncate/ all perform signed division of the dividend by the divisor, returning a quotient and remainder such that:

\[(\text{quotient} \times \text{divisor} + \text{remainder} = \text{dividend})\]

When the division is inexact (in other words, when the remainder is not zero), the kind of rounding depends on the operation:

- \%floor/ The quotient is rounded toward negative infinity.
- \%ceiling/ The quotient is rounded toward positive infinity.
- \%round/ The quotient is rounded toward the nearest integer. If the mathematical quotient is exactly halfway between two integers, then the resulting quotient is rounded to the nearest even integer.
- \%truncate/ The quotient is rounded toward zero.
- \%divide If both operands are non-negative, then the quotient is rounded toward zero. If either operand is negative, then the direction of rounding is unspecified, as is the sign of the remainder.

For all of these functions, an error is signaled if the value of the divisor is zero or if the correct value for the quotient exceeds the machine word range.

\%negative Function

Signature \%negative m => r overflow?

Parameters
- m – An instance of <machine-word>

Values
- r – An instance of <machine-word>
- overflow? – An instance of <boolean>

\%abs Function

Signature \%abs m => r overflow?

Parameters
- m – An instance of <machine-word>

Values
- r – An instance of <machine-word>
- overflow? – An instance of <boolean>

\%shift-left Function

Signature \%shift-left m count => low high overflow?

Parameters
- m – An instance of <machine-word>
- count – An instance of <integer>

Values
- low – An instance of <machine-word>
- high – An instance of <machine-word>

7.3. The machine-words Module
• overflow? – An instance of <boolean>

Discussion Arithmetic left shift of \( m \) by \( \text{count} \). An error is signaled unless \( 0 \leq \text{count} < \$\text{machine-word-size} \). The value of overflow? is false iff the high word result is a sign extension of the low word result.

%shift-right Function

Signature \%shift-right \( m \) \( \text{count} \) \( \Rightarrow \) \( r \)

Parameters
• \( m \) – An instance of <machine-word>
• \( \text{count} \) – An instance of <integer>

Values
• \( r \) – An instance of <machine-word>

Discussion Arithmetic right shift of \( m \) by \( \text{count} \). An error is signaled unless \( 0 \leq \text{count} < \$\text{machine-word-size} \).

Overflow signalling operations

For all of the following functions, all arguments that are specified as being specialized to <machine-word> accept an instance of <abstract-integer>, which is then coerced to a <machine-word> before performing the operation.

so%+ Function

Signature so%+ \( m1 \) \( m2 \) \( \Rightarrow \) \( \text{sum} \)

Parameters
• \( m1 \) – An instance of <machine-word>
• \( m2 \) – An instance of <machine-word>

Values
• \( \text{sum} \) – An instance of <machine-word>

Discussion Signed addition. An error is signaled on overflow.

so%- Function

Signature so%- \( m1 \) \( m2 \) \( \Rightarrow \) \( \text{difference} \)

Parameters
• \( m1 \) – An instance of <machine-word>
• \( m2 \) – An instance of <machine-word>

Values
• \( \text{difference} \) – An instance of <machine-word>

Discussion Signed subtraction. An error is signaled on overflow.

so%* Function

Signature so%* \( m1 \) \( m2 \) \( \Rightarrow \) \( \text{product} \)

Parameters
• \( m1 \) – An instance of <machine-word>
• m2 – An instance of `<machine-word>`

Values
• product – An instance of `<machine-word>`

Discussion Signed multiplication. An error is signaled on overflow.

so%negative Function

Signature so%negative m => r

Parameters
• m – An instance of `<machine-word>`

Values
• r – An instance of `<machine-word>`

Discussion Negation. An error is signaled on overflow.

so%abs Function

Signature so%abs m => r

Parameters
• m – An instance of `<machine-word>`

Values
• r – An instance of `<machine-word>`

Discussion Absolute value. An error is signaled on overflow.

so%shift-left Function

Signature so%shift-left m count => r

Parameters
• m – An instance of `<machine-word>`
• count – An instance of `<integer>`

Values
• r – An instance of `<machine-word>`

Discussion Arithmetic left shift of m by count. An error is signaled unless 0 ≤ count < $machine-word-size$. An error is signaled on overflow.

Signed double word operations

For all of the following functions, all arguments that are specified as being specialized to `<machine-word>` accept an instance of `<abstract-integer>`, which is then coerced to a `<machine-word>` before performing the operation.

d%floor/ Function

Signature d%floor/ dividend-low dividend-high divisor => quotient remainder

Parameters
• dividend-low – An instance of `<machine-word>`
• dividend-high – An instance of `<machine-word>`
• divisor – An instance of <machine-word>

Values
• quotient – An instance of <machine-word>
• remainder – An instance of <machine-word>

\%ceiling Function

Signature \%ceiling/ dividend-low dividend-high divisor => quotient remainder

Parameters
• dividend-low – An instance of <machine-word>
• dividend-high – An instance of <machine-word>
• divisor – An instance of <machine-word>

Values
• quotient – An instance of <machine-word>
• remainder – An instance of <machine-word>

\%round Function

Signature \%round/ dividend-low dividend-high divisor => quotient remainder

Parameters
• dividend-low – An instance of <machine-word>
• dividend-high – An instance of <machine-word>
• divisor – An instance of <machine-word>

Values
• quotient – An instance of <machine-word>
• remainder – An instance of <machine-word>

\%truncate Function

Signature \%truncate/ dividend-low dividend-high divisor => quotient remainder

Parameters
• dividend-low – An instance of <machine-word>
• dividend-high – An instance of <machine-word>
• divisor – An instance of <machine-word>

Values
• quotient – An instance of <machine-word>
• remainder – An instance of <machine-word>

\%divide Function

Signature \%divide dividend-low dividend-high divisor => quotient remainder

Parameters
• dividend-low – An instance of <machine-word>
• dividend-high – An instance of <machine-word>
The functions `d%divide`, `d%floor`, `d%ceiling`, `d%round`, and `d%truncate` all perform signed division of the double word dividend by the divisor, returning a quotient and remainder such that

\[(\text{quotient } \times \text{ divisor } + \text{ remainder } = \text{ dividend})\]

When the division is inexact (in other words, when the remainder is not zero), the kind of rounding depends on the operation:

- `d%floor/` The quotient is rounded toward negative infinity.
- `d%ceiling/` The quotient is rounded toward positive infinity.
- `d%round/` The quotient is rounded toward the nearest integer. If the mathematical quotient is exactly halfway between two integers then the resulting quotient is rounded to the nearest even integer.
- `d%truncate/` The quotient is rounded toward zero.
- `d%divide` If both operands are non-negative, then the quotient is rounded toward zero. If either operand is negative, then the direction of rounding is unspecified, as is the sign of the remainder.

For all of these functions, an error is signaled if the value of the divisor is zero or if the correct value for the quotient exceeds the machine word range.

### Unsigned single word operations

For all of the following functions, all arguments that are specified as being specialized to `<machine-word>` accept an instance of `<abstract-integer>`, which is then coerced to a `<machine-word>` before performing the operation.

#### u%+ Function

**Signature** `u%+ m1 m2 => sum carry`

**Parameters**
- `m1` – An instance of `<machine-word>`
- `m2` – An instance of `<machine-word>`

**Values**
- `sum` – An instance of `<machine-word>`
- `carry` – An instance of `<machine-word>`

**Discussion** Unsigned addition. The value represented by `carry` is either 0 or 1.

#### u%- Function

**Signature** `u%- m1 m2 => sum borrow`

**Parameters**
- `m1` – An instance of `<machine-word>`
- `m2` – An instance of `<machine-word>`

**Values**

### 7.3. The machine-words Module
• **sum** – An instance of `<machine-word>`
• **borrow** – An instance of `<machine-word>`

**Discussion**  Unsigned subtraction. The value represented by **borrow** is either 0 or 1.

### u%* Function

**Signature**  `u%* m1 m2 => low high`

**Parameters**
• **m1** – An instance of `<machine-word>`
• **m2** – An instance of `<machine-word>`

**Values**
• **low** – An instance of `<machine-word>`
• **high** – An instance of `<machine-word>`

**Discussion**  Unsigned multiplication.

### u%divide Function

**Signature**  `u%divide dividend divisor => quotient remainder`

**Parameters**
• **dividend** – An instance of `<machine-word>`
• **divisor** – An instance of `<machine-word>`

**Values**
• **quotient** – An instance of `<machine-word>`
• **remainder** – An instance of `<machine-word>`

**Discussion**  Performs unsigned division of the dividend by the divisor, returning a quotient and remainder such that

\[
(quotient \times divisor + remainder = dividend)
\]

An error is signaled if the value of the **divisor** is zero.

### u%rotate-left Function

**Signature**  `u%rotate-left m count => r`

**Parameters**
• **m** – An instance of `<machine-word>`
• **count** – An instance of `<integer>`

**Values**
• **r** – An instance of `<machine-word>`

**Discussion**  Logical left rotation of **m** by **count**. An error is signaled unless 0 \(\leq\) **count** < $machine-word-size$.

### u%rotate-right Function

**Signature**  `u%rotate-right m count => r`
Dylan Library Reference, Release 1.0

Parameters

• \( m \) – An instance of \(<\text{machine-word}\>

• \( \text{count} \) – An instance of \(<\text{integer}\>

Values

• \( r \) – An instance of \(<\text{machine-word}\>

Discussion Logical right rotation of \( m \) by \( \text{count} \). An error is signaled unless \( 0 \leq \text{count} < \$\text{machine-word-size} \).

\( u\%\text{shift-left} \) Function

Signature \( u\%\text{shift-left} \ m \ \text{count} \Rightarrow r \)

Parameters

• \( m \) – An instance of \(<\text{machine-word}\>

• \( \text{count} \) – An instance of \(<\text{integer}\>

Values

• \( r \) – An instance of \(<\text{machine-word}\>

Discussion Logical left shift of \( m \) by \( \text{count} \). An error is signaled unless \( 0 \leq \text{count} < \$\text{machine-word-size} \).

\( u\%\text{shift-right} \) Function

Signature \( u\%\text{shift-right} \ m \ \text{count} \Rightarrow r \)

Parameters

• \( m \) – An instance of \(<\text{machine-word}\>

• \( \text{count} \) – An instance of \(<\text{integer}\>

Values

• \( r \) – An instance of \(<\text{machine-word}\>

Discussion Logical right shift of \( m \) by \( \text{count} \). An error is signaled unless \( 0 \leq \text{count} < \$\text{machine-word-size} \).

\( u\%< \) Function

Signature \( u\%< \ m1 \ m2 \Rightarrow \text{smaller?} \)

Parameters

• \( m1 \) – An instance of \(<\text{machine-word}\>

• \( m2 \) – An instance of \(<\text{machine-word}\>

Values

• \( \text{smaller?} \) – An instance of \(<\text{boolean}\>

Discussion Unsigned comparison.

Unsigned double word operations

For all of the following functions, all arguments that are specified as being specialized to \(<\text{machine-word}\>\) accept an instance of \(<\text{abstract-integer}\>\), which is then coerced to a \(<\text{machine-word}\>) before performing the operation.

7.3. The machine-words Module
ud%divide Function

**Signature**  
ud%divide dividend-low dividend-high divisor => quotient remainder

**Parameters**
- **dividend-low** – An instance of `<machine-word>`
- **dividend-high** – An instance of `<machine-word>`
- **divisor** – An instance of `<machine-word>`

**Values**
- **quotient** – An instance of `<machine-word>`
- **remainder** – An instance of `<machine-word>`

**Discussion**
Performs unsigned division of the double word dividend by the divisor, returning a quotient and remainder such that

\[(\text{quotient} \times \text{divisor} + \text{remainder} = \text{dividend})\]

An error is signaled if the value of the divisor is zero or if the correct value for the quotient exceeds the machine word range.

ud%shift-left Function

**Signature**  
ud%shift-left low high count => low high

**Parameters**
- **low** – An instance of `<machine-word>`
- **high** – An instance of `<machine-word>`
- **count** – An instance of `<integer>`

**Values**
- **low** – An instance of `<machine-word>`
- **high** – An instance of `<machine-word>`

**Discussion**
Logical left shift by count of the double word value represented by low and high. An error is signaled unless \(0 \leq \text{count} < \text{machine-word-size}\).

ud%shift-right Function

**Signature**  
ud%shift-right low high count => low high

**Parameters**
- **low** – An instance of `<machine-word>`
- **high** – An instance of `<machine-word>`
- **count** – An instance of `<integer>`

**Values**
- **low** – An instance of `<machine-word>`
- **high** – An instance of `<machine-word>`

**Discussion**
Logical right shift by count of the double word value represented by low and high. An error is signaled unless \(0 \leq \text{count} < \text{machine-word-size}\).
7.4 The simple-format Module

The simple-format module provides a subset of the functionality of the format, format-out, print, and standard-io modules, for programs that just need to do simple output to the console without pulling in the full IO library.

**format-to-string Function**

Returns a formatted string constructed from its arguments.

*Signature*

```dylan
format-to-string format-string #rest format-arguments => string
```

*Parameters*

- `format-string` – An instance of `<byte-string>`.
- `format-arguments` (#rest) – Instances of `<object>`.

*Values*

- `result-string` – An instance of `<byte-string>`.

*Conditions* This function signals an error of type `<simple-error>` if any of the format directives in `format-string` are invalid.

*Discussion*

Returns a formatted string constructed from its arguments, which include a `format-string` of formatting directives and a series of `format-arguments` to be formatted according to those directives.

The `format-string` must be a Dylan format string as described on pages 112-114 of the DRM.

**format-out Function**

Formats its arguments to the standard output.

*Signature*

```dylan
format-out format-string #rest format-arguments => ()
```

*Parameters*

- `format-string` – An instance of `<byte-string>`.
- `format-arguments` – Instances of `<object>`.

*Discussion*

Formats its arguments to the standard output.

This function writes directly to the console using the underlying OS facilities rather than using the `+standard-output+` stream defined by the standard-io module in the IO library. For example, on Unix systems this writes to file descriptor 1 via the `write` C function.

7.5 The simple-profiling Module

This module provides an easy to use interface for measuring application performance.

7.5.1 Simple Profiling

**timing Statement Macro**

Returns the time, in seconds and microseconds, spent executing the body of code it is wrapped around.

**Macro Call**

```dylan
timing body
```
**Parameters**

- **body** – A Dylan body `bnf`

**Values**

- **seconds** – An instance of `<integer>`.
- **microseconds** – An instance of `<integer>`.

**Discussion**

Returns the time, in seconds and microseconds, spent executing the body of code it is wrapped around.

The first value returned is the number of whole seconds spent in `body`. The second value returned is the number of microseconds spent in `body` in addition to `seconds`.

**Example**

```dylan
timing () [ *body* ] end [ timing ]
```

**profiling Macro**

Gives access to the CPU time, in seconds and microseconds, as well as some memory allocation statistics, spent executing the body of code it is wrapped around.

**Macro Call**

```dylan
profiling ([*profiling-type*, ...])
  *body*
  results
  *results*
end
```

**Parameters**

- **profiling-type** – Any of `cpu-time-seconds`, `cpu-time-microseconds`, `allocation` and `allocation-stats`.
- **body** – A Dylan body `bnf`
- **results** – A Dylan body `bnf`

**Discussion**

This macro takes a set of `profiling-type` parameters, performs the `body` of code and then executes the `results`. Within the results block, there will be bindings with the names of the profiling types which have the corresponding value.

**Note:** Using `allocation-stats` is more involved and not as flexible as one might hope. This needs further documentation and perhaps an improved implementation.
Example

```dylan
profiling (cpu-time-seconds, cpu-time-microseconds, allocation)
  execute-command(command)
  results
  message(context, "Command took %d.%s seconds, and allocated %d bytes →",
     cpu-time-seconds,
     integer-to-string(floor/(cpu-time-microseconds, 1000),
     →size: 3),
     allocation)
end
```

7.5.2 Internals

These functions don’t typically need to be called directly but may be useful in some scenarios.

<profiling-state> Type

Equivalent <object-table>

<cpu-profiling-type> Type

Equivalent one-of(#"cpu-time-seconds", #"cpu-time-microseconds")

profiling-type-result Open Generic function

Signature profiling-type-result (state keyword #key #all-keys) => (value)

Parameters

- **state** – An instance of <profiling-state>.
- **keyword** – An instance of <symbol>.

Values

- **value** – An instance of <object>.

profiling-type-result (<profiling-state>, <cpu-profiling-type>) Method

profiling-type-result (<profiling-state>, singleton(#"allocation")) Method

profiling-type-result (<profiling-state>, singleton(#"allocation-stats")) Method

start-profiling Function

Signature start-profiling (profiling-types) => (state)

Parameters

- **profiling-types** – A sequence of any of #"cpu-time-seconds", #"cpu-time-microseconds", #"allocation" and #"allocation-stats".

Values

- **state** – An instance of <profiling-state>.

This is useful for when direct control over profiling is needed rather than using the profiling macro.

start-profiling-type Open Generic function
Signature  start-profiling-type (state keyword) => ()

Parameters

• **state** – An instance of `<profiling-state>`.

• **keyword** – An instance of `<symbol>`.

**start-profiling-type**(<profiling-state>, <cpu-profiling-type>) Method

**start-profiling-type**(<profiling-state>, singleton(#"allocation")) Method

**start-profiling-type**(<profiling-state>, singleton(#"allocation-stats")) Method

**stop-profiling** Function

Signature  stop-profiling (state profiling-types) => ()

Parameters

• **state** – An instance of `<profiling-state>`.

• **profiling-types** – A sequence of `<symbol>`. These symbols should be the same as those passed to **start-profiling**.

**stop-profiling-type** Open Generic function

Signature  stop-profiling-type (state keyword) => ()

Parameters

• **state** – An instance of `<profiling-state>`.

• **keyword** – An instance of `<symbol>`.

**stop-profiling-type**(<profiling-state>, <cpu-profiling-type>) Method

**stop-profiling-type**(<profiling-state>, singleton(#"allocation")) Method

**stop-profiling-type**(<profiling-state>, singleton(#"allocation-stats")) Method

### 7.6 The simple-random Module

Common Dylan provides a simple facility for generating sequences of pseudo-random integers via the *simple-random* module.

Instances of the sealed class `<random>` generate pseudo-random integers. Given an instance of `<random>`, the function `random` will return a pseudo-random integer.

**<random>** Instantiable Sealed Class

The class of random number generators.

**Superclasses** `<object>`

**Init-Keywords**

• **seed** – An instance of `<integer>`. Default value: computed to be random.

**Discussion**

The class of random number generators.

The seed value from which to start the sequence of integers. Default value: computed to be random.
random Function
Returns a pseudorandomly generated number greater than or equal to zero and less than a specified value.

Signature  random upperbound #key random => random-integer

Parameters
• upperbound – An instance of <integer>.
• random(#key) – An instance of <random>.

Values
• random-integer – An instance of <integer>.

Discussion  Returns a pseudorandomly generated number greater than or equal to zero and less than upperbound.

7.7 The simple-timers Module

Common Dylan provides a simple facility for timers to track intervals of time via the simple-timers module.

Timers offer microsecond resolution on all supported platforms. Timers attempt to be monotonic where that capability is supported by the operating system.

<profiling-timer> Class
The timer class.

Superclasses  <object>

Discussion  The timer class. Timers start out stopped and must be started with timer-start.

timer-start Generic function
Starts a timer.

Signature  timer-start timer => ()

Parameters
• timer – An instance of <profiling-timer>.

See also
• timer-stop

timer-stop Generic function
Stops a timer and returns the elapsed time.

Signature  timer-stop timer => (seconds, microseconds)

Parameters
• timer – An instance of <profiling-timer>.

Values
• seconds – An instance of <integer>.
• microseconds – An instance of <integer>.

See also
• timer-start

timer-accumulated-time Generic function
Returns the time since the timer was started.
timer-accumulated-time Generic function
Signature timer-accumulated-time timer => (seconds, microseconds)
Parameters
• timer – An instance of <profiling-timer>.
Values
• seconds – An instance of <integer>.
• microseconds – An instance of <integer>.

timer-running? Generic function
Returns true if the timer is running.
Signature timer-running? timer => (running?)
Parameters
• timer – An instance of <profiling-timer>.
Values
• running? – An instance of <boolean>.

7.8 The transcendentals Module

The transcendentals module, exported from the common-dylan library, provides a set of generic functions for ANSI C-like behavior over real numbers. The restrictions and error cases described in this document are intended to be the same as they are in ANSI C.

The single module, transcendentals, exports these generic functions and methods.

Because implementation of these functions might be by a standard library for transcendentals accessed by a foreign function interface, the exact precision and algorithms (and hence, the exact results) for all of these functions is explicitly unspecified. Note, however, that a program expects the following, even in libraries that are implemented by calling foreign libraries:

• Domain and range errors should be signalled as Dylan errors.

• Floating point precision contagion must obey Dylan rules (that is, functions called on single precision values return single precision results, and functions on double precision values return double precision results)

7.8.1 Reference

This section contains a reference entry for each item exported from the common-dylan library’s transcendentals module.

^ Generic function
Summary Returns its first argument, raised to the power indicated by its second argument.
Signature ^ b x => y
Parameters
• b – An instance of type <real>.
• x – An instance of type <real>.
Values
• y – An instance of type <real>.
Discussion

Returns $b$ raised to the power $x$. If $b$ is 0 and $x$ is not positive, an error is signalled. If $b$ is negative and $x$ is not an integer, an error is signalled.

If $b$ and $x$ are both integers, the result is an integer. If $x$ is negative, an error is signalled.

The floating point precision is given by the precision of $b$.

See also

- $exp$

acos Generic function

Summary Returns the arc cosine of its argument.

Signature $acos x => y$

Parameters

- $x$ – an instance of type $<real>$. The angle, in radians. If $x$ is not in the range $[-1,+1]$, an error is signalled.

Values

- $y$ – An instance of type $<float>$.

Discussion Returns the arc cosine of its argument. The floating point precision of the result is given by the precision of $x$.

See also

- $asin$
- $atan$

acosh Generic function

Summary Returns the hyperbolic arc cosine of its argument.

Signature $acosh x => y$

Parameters

- $x$ – An instance of type $<real>$. The angle, in radians.

Values

- $y$ – An instance of type $<float>$.

Discussion Returns the hyperbolic arc cosine of its argument. The floating point precision of the result is given by the precision of $x$.

See also

- $asinh$
- $atanh$

asin Generic function

Summary Returns the arc sine of its argument.

Signature $asin x => y$

Parameters

- $x$ – An instance of type $<real>$. The angle, in radians. If $x$ is not in the range $[-1,+1]$, an error is signalled.
Values

- **y** – An instance of type `<float>`.

Discussion Returns the arc sine of its argument. The floating point precision of the result is given by the precision of `x`.

See also

- `acos`
- `atan`

**asinh** Generic function

Summary Returns the hyperbolic arc sine of its argument.

Signature `asinh x => y`

Parameters

- **x** – An instance of type `<real>`.

Values

- **y** – An instance of type `<float>`.

Discussion Returns the hyperbolic arc sine of its argument. The floating point precision of the result is given by the precision of `x`.

See also

- `acosh`
- `atanh`

**atan** Generic function

Summary Returns the arc tangent of its argument.

Signature `atan x => y`

Parameters

- **x** – An instance of type `<real>`.

Values

- **y** – An instance of type `<float>`.

Discussion Returns the arc tangent of its argument. The floating point precision of the result is given by the precision of `x`.

See also

- `acos`
- `asin`

**atan2** Generic function

Summary Returns the arc tangent of one angle divided by another.

Signature `atan2 x y => z`

Parameters

- **x** – An instance of type `<real>`.

The first angle, in radians.
• \( y \) – An instance of type \(<\text{real}>\). The second angle, in radians.

Discussion

Returns the arc tangent of \( x \) divided by \( y \). \( x \) may be zero if \( y \) is not zero. The signs of \( x \) and \( y \) are used to derive what quadrant the angle falls in.

The floating point precision of the result is given by the precision of \( x/y \).

**atanh** Generic function

Summary Returns the hyperbolic arc tangent of its argument.

Signature \( \text{atanh} \ x \Rightarrow y \)

Parameters

• \( x \) – An instance of type \(<\text{real}>\). The angle, in radians.

Values

• \( y \) – An instance of type \(<\text{float}>\).

Discussion Returns the hyperbolic arc tangent of its argument. The floating point precision of the result is given by the precision of \( x \).

See also

• \( \text{acosh} \)
• \( \text{asinh} \)

**cos** Generic function

Summary Returns the cosine of its argument.

Signature \( \cos \ x \Rightarrow y \)

Parameters

• \( x \) – An instance of type \(<\text{real}>\). The angle, in radians.

Values

• \( y \) – An instance of type \(<\text{float}>\).

Discussion Returns the cosine of its argument. The floating point precision of the result is given by the precision of \( x \).

See also

• \( \text{sin} \)
• \( \text{sincos} \)
• \( \text{tan} \)

**cosh** Generic function

Summary Returns the hyperbolic cosine of its argument.

Signature \( \cosh \ x \Rightarrow y \)

Parameters

• \( x \) – An instance of type \(<\text{real}>\). The angle, in radians.
Values

• \( y \) – An instance of type \(<\text{float}\>\).

Discussion Returns the hyperbolic cosine of its argument. The floating point precision of the result is given by the precision of \( x \).

See also

• \( \sinh \)
• \( \tanh \)

\$double-e Constant

summary The value of \( e \), the base of natural logarithms, as a double precision floating point number.

type \(<\text{double-float}\>\)

superclasses \(<\text{float}\>\)

description The value of \( e \), the base of natural logarithms, as a double precision floating point number.

See also

• \$single-e

\$double-pi Constant

Summary The value of \( \pi \) as a double precision floating point number.

Type \(<\text{double-float}\>\)

Superclasses \(<\text{float}\>\)

Discussion The value of \( \pi \) as a double precision floating point number.

See also

• \$single-pi

exp Generic function

Summary Returns \( e \), the base of natural logarithms, raised to the power indicated by its argument.

Signature exp \( x \Rightarrow y \)

Parameters

• \( x \) – An instance of type \(<\text{real}\>\).

Values

• \( y \) – An instance of type \(<\text{float}\>\).

Discussion Returns \( e \), the base of natural logarithms, raised to the power \( x \). The floating point precision is given by the precision of \( x \).

See also

• \(^\)
• \( \text{ilog2} \)
• \( \text{log} \)
• \( \text{logn} \)
**hypot Generic function**

**Summary**  Returns the Euclidian distance.

**Signature**  `hypot x, y => z`

**Parameters**

- `x` – An instance of type `<number>`.
- `y` – An instance of type `<number>`.

**Values**

- `z` – An instance of type `<number>`.

**Discussion**  Returns the Euclidian distance without unnecessary overflow or underflow.

The floating point precision is given by the precision of `x`.

**isqrt Generic function**

**Summary**  Returns the integer square root of its argument.

**Signature**  `isqrt x => y`

**Parameters**

- `x` – An instance of type `<integer>`.

**Values**

- `y` – An instance of type `<integer>`.

**Discussion**  Returns the integer square root of `x`, that is the greatest integer less than or equal to the exact positive square root of `x`. If `x < 0`, an error is signalled.

**See also**

- `sqrt`

**log Generic function**

**Summary**  Returns the natural logarithm of its argument.

**Signature**  `log x => y`

**Parameters**

- `x` – An instance of type `<real>`.

**Values**

- `y` – An instance of type `<float>`.

**Discussion**  Returns the natural logarithm of `x` to the base e. If `x <= 0 <= 1`, an error is signalled.

The floating point precision of the result is given by the precision of `x`.

**See also**

- `exp`
- `ilog2`
- `logn`

**logn Function**

**Summary**  Returns the logarithm of its argument to the given base.
Signature  logn x b => y

Parameters

• x – An instance of <number>
• b – The base. An instance of <number>.

Values

• y – An instance of <number>.

Discussion  Returns the logarithm of x to the base b. If x <= 0 <= 1, an error is signalled. The floating point precision of the result is given by the precision of x.

See also

• exp
• log
• ilog2

ilog2 Function

Summary  Returns the base two logarithm of its argument, truncated to an integer.

Signature  ilog2 x => y

Parameters

• x – An instance of <integer>.

Values

• y – An instance of <integer>.

Discussion  Returns the integer base two logarithm of x, that is the greatest integer less than or equal to the exact base two logarithm of x.

See also

• exp
• logn
• log

sin Generic function

Summary  Returns the sine of its argument.

Signature  sin x => y

Parameters

• x – An instance of type <real>. The angle, in radians.

Values

• y – An instance of type <float>.

Discussion  Returns the sine of its argument. The floating point precision of the result is given by the precision of x.

See also

• cos
• sincos
• tan

tan

sincos Generic function

Summary Returns both the sine and the cosine of its argument.

Signature sincos x => (s, c)

Parameters

• x – An instance of type <real>. The angle, in radians.

Values

• s – An instance of type <float>. The result of sin(x).
• c – An instance of type <float>. The result of cos(x).

Discussion Returns both the sine and the cosine of its argument. The floating point precision of the
results is given by the precision of x.

See also cos sin

$single-e Constant

Summary The value of e, the base of natural logarithms, as a single precision floating point number.

Type <single-float>

Superclasses <float>

Discussion The value of e, the base of natural logarithms, as a single precision floating point number.

See also

• $double-e

$single-pi Constant

Summary The value of π as a single precision floating point number.

Type <single-float>

Superclasses <float>

Discussion The value of π as a single precision floating point number.

See also

• $double-pi

sinh Generic function

Summary Returns the hyperbolic sine of its argument.

Signature sinh x => y

Parameters

• x – An instance of type <real>. The angle, in radians.

Values

• y – An instance of type <float>.

Discussion Returns the hyperbolic sine of its argument. The floating point precision of the result is
given by the precision of x.

See also

• cosh
sqrt Generic function

Summary Returns the square root of its argument.
Signature sqrt x => y

Parameters

• x – An instance of type <real>.

Values

• y – An instance of type <float>.

Discussion Returns the square root of x. If x is less than zero an error is signalled. The floating point precision of the result is given by the precision of x.

See also

• isqrt
tan Generic function

Summary Returns the tangent of its argument.
Signature tan x => y

Parameters

• x – An instance of type <real>. The angle, in radians.

Values

• y – An instance of type <float>.

Discussion Returns the tangent of its argument. The floating point precision of the result is given by the precision of x.

See also

• cos
  • sin
tanh Generic function

Summary Returns the hyperbolic tangent of its argument.
Signature tanh x => y

Parameters

• x – An instance of type <real>. The angle, in radians.
  • y – An instance of type <float>.

Discussion Returns the hyperbolic tangent of its argument. The floating point precision of the result is given by the precision of x.

See also

• cosh
  • sinh
The common-dylan module re-exports everything in the common-extensions and dylan modules. This is a convenience module that is very widely used.

It also re-exports these modules which are defined in the dylan library:

- The dylan Module
- The threads Module
- The finalization Module
8.1 The finalization Module

- What is finalization?
- How the finalization interface works
  - Registering objects for finalization
  - Draining the finalization queue
  - Finalizers
  - After finalization
  - Upon application exit
  - The effects of multiple registrations
  - The effects of resurrecting objects
  - The effects of finalizing objects directly
  - Finalization and weak tables
- Writing finalizers
  - Class-based finalization
  - Parallels with INITIALIZE methods
  - Simplicity and robustness
  - Singleton finalizers
- Using finalization in applications
  - How can my application drain the finalization queue automatically?
  - When should my application drain the finalization queue?

Open Dylan provides a finalization interface in the finalization module of common-dylan. This section explains finalization, the finalization interface provided, and how to use the interface in applications. Note that you must use finalization to be able to use the interface described in this documentation.
8.1.1 What is finalization?

The Memory Management Reference defines finalization as follows:

In garbage-collected languages, it is often necessary to perform actions on some objects after they are no longer in use and before their memory can be recycled. These actions are known as finalization or termination.

A common use of finalization is to release a resource when the corresponding “proxy” object dies, if the proxy object has indefinite extent and therefore more predictable tools like block () ... cleanup ... end won’t work.

For example, when interfacing Dylan code with foreign code that does not have automatic memory management, if an interface involves a Dylan object that references a foreign object it may be necessary to free the memory resources of the foreign object when the Dylan object is reclaimed.

8.1.2 How the finalization interface works

The following sections give a broad overview of how finalization works and how to use the interface.

Registering objects for finalization

Finalization works through cooperation with the garbage collector. Objects that are no longer referenced by the application that created them will eventually be discovered by Dylan’s garbage collector and are then available to be reclaimed.

By default, the garbage collector reclaims such objects without notifying your application. If it is necessary to finalize an object before it is reclaimed, your application must inform the garbage collector.

The garbage collector maintains a register of objects requiring finalization before being reclaimed. To add an object to the register, call the function finalize-when-unreachable on the object. Objects on the register are said to be finalizable.

If the garbage collector discovers that a finalizable object is no longer referenced by the application, it does not reclaim it immediately. Instead, it takes the object off its finalization register, and adds it to the finalization queue.

The finalization queue contains all the objects awaiting finalization. The garbage collector will not reclaim the objects until they have been finalized.

A simple example of registering a finalizer:

```dylan
define method initialize (lock :: <recursive-lock>, #key) => ()
    drain-finalization-queue();
    next-method();
    let res = primitive-make-recursive-lock(lock,
                                            lock.synchronization-name);
    check-synchronization-creation(lock, res);
    finalize-when-unreachable(lock);
end method;
```

The reasons for calling `drain-finalization-queue` are discussed below.

Note: The library containing this code must have `use finalization;` in its module definition.
**Draining the finalization queue**

Objects in the finalization queue wait there until the application drains it by calling the function `drain-finalization-queue`. This function finalizes every object in the queue.

The finalization queue is not normally drained automatically. See *How can my application drain the finalization queue automatically?* for details of how you can set up a thread to do so.

**Note:** The order in which objects in the finalization queue are finalized is not defined. Applications should not make any assumptions about finalization ordering.

**Finalizers**

The `drain-finalization-queue` function finalizes each object in the finalization queue by calling the generic function `finalize` on it. You should define methods for `finalize` on those classes whose instances may require finalization. These methods are called *finalizers*.

The recommended interface to finalization is through `finalize-when-unreachable` and `drain-finalization-queue`, but calling `finalize` on an object directly is also permitted. If you are certain you are finished with an object, it may be desirable to do so. For example, you might want to finalize an object created in a local binding before it goes out of scope.

**Note:** Finalizable objects are only removed from the register if the garbage collector discovers that they are unreachable and moves them into the finalization queue. Calling `finalize` on an object directly does not affect its registration status.

The `drain-finalization-queue` function makes each call to `finalize` inside whatever dynamic handler environment is present when `drain-finalization-queue` is called. If the call to `drain-finalization-queue` is aborted via a non-local exit during a call to `finalize`, the finalization queue retains all the objects that had been added to it but which had not been passed to `finalize`.

There is a default method for `finalize` on `<object>`. The method does nothing. It is available so that it is safe for all finalizers to call `next-method`, a practice that we strongly encourage. See *Writing finalizers*.

**After finalization**

Once an object in the finalization queue has been finalized, it typically becomes available for reclamation by the garbage collector. Because it has been taken off the garbage collector’s finalization register, it will not be queued up for finalization again.

**Note:** There are exceptions to this rule; see *The effects of multiple registrations* and *The effects of resurrecting objects*.

**Upon application exit**

There are no guarantees that objects which are registered for finalization will actually be finalized before the application exits. This is not a problem on many operating systems, which free any resources held by a process when it exits.

Where it is necessary to guarantee an action at the time the application exits, you should use a more explicit mechanism.
The effects of multiple registrations

Sometimes objects are registered for finalization more than once. The effects of multiple registration are defined as follows:

Calling `finalize-when-unreachable` on an object \( n \) times causes that object to be added to the finalization queue up to \( n \) times, where \( n \) is greater than or equal to zero. There is no guarantee that the object will be added exactly \( n \) times.

Note that this definition so general that it does not guarantee that any object will ever be added to be finalization queue. In practice, Common Dylan’s implementation guarantees that an object is added to the queue at least once whenever an object has been determined to be unreachable by the garbage collector.

To remain robust under multiple registration, finalizers should be idempotent: that is, the effect of multiple `finalize` calls on an object should is the same as the effect of a single call.

The effects of resurrecting objects

If a finalizer makes an object reachable again, by storing a reference to the object in a variable, slot, or collection, we say it has resurrected it. An object may also be resurrected if it becomes reachable again when some other object is resurrected (because it is directly or indirectly referenced by that other object).

Resurrecting objects has pitfalls, and must be done with great care. Since finalizers typically destructively modify objects when freeing their resources, it is common for finalization to render objects unusable. We do not recommend resurrection if there is any possibility of the object being left in an unusable state, or if the object references any other objects whose transitive closure might include an object left in such a state by another call to `finalize`.

If you do resurrect objects, note that they will not be finalized again unless you re-register them.

The effects of finalizing objects directly

Any object that has been finalized directly, through the application itself calling `finalize` on it, may not yet be unreachable. Like any normal object it only becomes eligible for reclamation when it is unreachable. If such an object was also registered for finalization using `finalize-when-unreachable`, it can end up being finalized again via the queue mechanism.

Finalization and weak tables

If an object is both registered for finalization and is weakly referred to from a weak table, finalization occurs first, with weak references being removed afterwards. That is, reachability is defined in terms of strong references only, as far as finalization is concerned. Weak references die only when an object’s storage is finally reclaimed.

For more on weak tables, see `Weak tables`.

8.1.3 Writing finalizers

Because the default `finalize` method, on `<object>`, does nothing, you must define your own `finalize` methods to get results from the finalization interface. This section contains useful information about writing finalizers.

Class-based finalization

If your application defines a class for which all instances require finalization, call `finalize-when-unreachable` in its `initialize` method.
Parallels with INITIALIZE methods

The default method on `<object>` is provided to make it safe to call `next-method` in all finalizers. This situation is parallel to that for class `initialize` methods, which call `next-method` before performing their own initializations. By doing so, `initialize` methods guarantee that the most specific initializations occur last.

By contrast, finalizers should call `next-method` last, in case they depend on the superclass finalizer not being run.

Simplicity and robustness

Write finalizers that are simple and robust. They might be called in any context, including within other threads; with careful design, your finalizers will work in most or all possible situations.

A finalizer might be called on the same object more than once. This could occur if the object was registered for finalization more than once, or if your application registered the object for finalization and also called `finalize` on it directly. To account for this, write finalizers that are idempotent: that is, the effect of multiple calls is the same as the effect of a single call. See *The effects of multiple registrations* for more on the effects of multiple registrations.

Remember that the order in which the finalization queue is processed is not defined. Finalizers cannot make assumptions about ordering.

This is particularly important to note when writing finalizers for classes that are typically used to form circular or otherwise interestingly connected graphs of objects. If guarantees about finalization in graphs of objects are important, we suggest registering a root object for finalization and making its finalizer traverse the graph (in some graph-specific well-ordered fashion) and call the `finalize` method for each object in the graph requiring finalization.

Singleton finalizers

Do not write singleton methods on `finalize`. The singleton method itself would refer to the object, and hence prevent it from becoming unreachable.

8.1.4 Using finalization in applications

This section answers questions about using finalization in an application.

How can my application drain the finalization queue automatically?

If you would prefer the queue to be drained asynchronously, use the automatic finalization interface. For more details, see `automatic-finalization-enabled?` and `automatic-finalization-enabled?-setter`.

Libraries that do not wish to depend on automatic finalization should not use those functions. They should call `drain-finalization-queue` synchronously at useful times, such as whenever they call `finalize-when-unreachable`.

Libraries that are not written to depend on automatic finalization should always behave correctly if they are used in an application that does use it.

When should my application drain the finalization queue?

If you do not use automatic finalization, drain the queue synchronously at useful points in your application, such as whenever you call `finalize-when-unreachable` on an object.

This section contains a reference description for each item in the finalization interface. These items are exported from the `common-dylan` library in a module called `finalization`.

8.1. The finalization Module
**automatic-finalization-enabled? Function**

Returns true if automatic finalization is enabled, and false otherwise.

**Signature**

`automatic-finalization-enabled? () => enabled?`

**Values**


**Discussion**

Returns true if automatic finalization is enabled, and false otherwise.

**See also**

- `automatic-finalization-enabled?-setter`
- `drain-finalization-queue`
- `finalize-when-unreachable`
- `finalize`

**automatic-finalization-enabled?-setter Function**

Sets the automatic finalization system state.

**Signature**

`automatic-finalization-enabled?-setter newval => ()`

**Parameters**

- `newval` – An instance of `<boolean>`.

**Discussion**

Sets the automatic finalization system state to `newval`.

The initial state is `#f`. If the state changes from `#f` to `#t`, a new thread is created which regularly calls `drain-finalization-queue` inside an empty dynamic environment (that is, no dynamic condition handlers). If the state changes from `#t` to `#f`, the thread exits.

**See also**

- `automatic-finalization-enabled?`
- `drain-finalization-queue`
- `finalize-when-unreachable`
- `finalize`

**drain-finalization-queue Function**

Calls `finalize` on every object in the finalization queue.

**Signature**

`drain-finalization-queue () => ()`

**Discussion**

Calls `finalize` on each object that is awaiting finalization.

Each call to `finalize` is made inside whatever dynamic handler environment is present when `drain-finalization-queue` is called. If the call to `drain-finalization-queue` is aborted via a non-local exit during a call to `finalize`, the finalization queue retains all the objects that had been added to it but which had not been passed to `finalize`.

The order in which objects in the finalization queue will be finalized is not defined. Applications should not make any assumptions about finalization ordering.

**See also**

- `finalize-when-unreachable`
• `finalize`
• `automatic-finalization-enabled?`
• `automatic-finalization-enabled?-setter`

**finalize-when-unreachable Function**
Registers an object for finalization.

**Signature** `finalize-when-unreachable object => object`

**Parameters**
- `object` – An instance of `<object>`.

**Values**
- `object` – An instance of `<object>`.

**Discussion**
 Registers `object` for finalization. If `object` becomes unreachable, it is added to the finalization queue rather than being immediately reclaimed.

`object` waits in the finalization queue until the application calls `drain-finalization-queue`, which processes each object in the queue by calling the generic function `finalize` on it.

The function returns its argument.

**See also**
- `finalize`
- `drain-finalization-queue`
- `automatic-finalization-enabled?`
- `automatic-finalization-enabled?-setter`

**finalize Generic function**
Finalizes an object.

**Signature** `finalize object => ()`

**Parameters**
- `object` – An instance of `<object>`.

**Discussion**
Finalizes `object`.

You can define methods on `finalize` to perform class-specific finalization procedures. These methods are called `finalizers`.

A default `finalize` method on `<object>` is provided.

The main interface to finalization is the function `drain-finalization-queue`, which calls `finalize` on each object awaiting finalization. Objects join the finalization queue if they become unreachable after being registered for finalization with `finalize-when-unreachable`. However, you can call `finalize` directly if you wish.

Once finalized, `object` is available for reclamation by the garbage collector, unless finalization made it reachable again. (This is called `resurrection`; see *The effects of resurrecting objects*.) Because the object has been taken off the garbage collector’s finalization register, it will not be

---

**8.1. The finalization Module**

187
added to the finalization queue again, unless it is resurrected. However, it might still appear in the queue if it was registered more than once.

Do not write singleton methods on `finalize`. A singleton method would itself reference the object, and hence prevent it from becoming unreachable.

**See also**
- `finalize`
- `finalize-when-unreachable`
- `drain-finalization-queue`
- `automatic-finalization-enabled?`
- `automatic-finalization-enabled?-setter`

### finalize(<object>) Method

Finalizes an object.

**Signature**

```
finalize object => ()
```

**Parameters**

- `object` – An instance of `<object>`.

**Discussion**  This method is a default finalizer for all objects. It does nothing, and is provided only to make `next-method` calls safe for all methods on `finalize`.

**See also**
- `finalize-when-unreachable`
- `finalize`
- `drain-finalization-queue`
- `automatic-finalization-enabled?`
- `automatic-finalization-enabled?-setter`

# 8.2 The dylan-primitives Module

## 8.2.1 Runtime System Functions

### Primitives for Machine Information

#### primitive-read-cycle-counter Primitive

**Signature**

```
() => (cycle-count)
```

**Values**

- `cycle-count` – An instance of `<raw-machine-word>`.

**Discussion**

On x86 and x86_64 architectures, this is equivalent to calling the `rdtsc` instruction. For full precision, this should only be used on 64 bit builds.

This is equivalent to using the LLVM intrinsic `llvm.readcyclecounter` or the Clang built-in `__builtin_readcyclecounter`.
This has not been implemented in the HARP back-end.

**primitive-read-return-address Primitive**

**Signature** () => (return-address)

**Values**

- **return-address** – An instance of `<raw-machine-word>`.

**Discussion**

Returns the address to which the current function will return when it exits. This yields the address of the code which invoked the current function.

For best results, invoke this primitive within a function which has been prevented from being inlined.

This is equivalent to using the LLVM intrinsic `llvm.returnaddress` or the Clang and GCC built-in `__builtin_return_address`.

This has not been implemented in the HARP back-end.

---

**Primitive Functions for the threads library**

This section describes in detail the arguments, values, and operations of the primitive functions.

**Threads**

**primitive-make-thread Primitive**

**Signature** (thread :: <thread>, function :: <function>) => ()

**Parameters**

- **thread** – An instance of `<thread>`.
- **function** – The initial function to run after the thread is created. An instance of `<function>`.

**Discussion** Creates a new OS thread and destructively modifies the container slots in the Dylan thread object with the handles of the new OS thread. The new OS thread is started in a way which calls the supplied Dylan function.

**primitive-destroy-thread Primitive**

**Signature** (thread :: <thread>) => ()

**Parameters**

- **thread** – An instance of `<thread>`.

**Discussion** Frees any runtime-allocated memory associated with the thread.

**primitive-initialize-current-thread Primitive**

**Signature** (thread :: <thread>) => ()

**Parameters**

- **thread** – An instance of `<thread>`.
Discussion The container slots in the Dylan thread object are destructively modified with the handles of the current OS thread. This function will be used to initialize the first thread, which will not have been started as the result of a call to primitive-make-thread.

**primitive-thread-join-single Primitive**

**Signature** (thread :: <thread>) => (error-code :: <integer>)

**Parameters**

- **thread** – An instance of <thread>.

**Values**

- **error-code** – An instance of <integer>. 0 = ok, anything else is an error, corresponding to a multiple join.

**Discussion** The calling thread blocks (if necessary) until the specified thread has terminated.

**primitive-thread-join-multiple Primitive**

**Signature** (thread-vector :: <simple-object-vector>) => (result)

**Parameters**

- **thread-vector** – A <simple-object-vector> containing <thread> objects

**Values**

- **result** – The <thread> that was joined, if the join was successful; otherwise, a <integer> indicating the error.

**Discussion** The calling thread blocks (if necessary) until one of the specified threads has terminated.

**primitive-thread-yield Primitive**

**Signature** () => ()

**Discussion** For co-operatively scheduled threads implementations, the calling thread yields execution in favor of another thread. This may do nothing in some implementations.

**primitive-current-thread Primitive**

**Signature** () => (thread-handle)

**Values**

- **thread-handle** – A low-level handle corresponding to the current thread.

**Discussion** Returns the low-level handle of the current thread, which is assumed to be in the handle container slot of one of the <thread> objects known to the Dylan library. This result is therefore NOT a Dylan object. The mapping from this value back to the <thread> object must be performed by the Dylan threads library, and not the primitive layer, because the <thread> object is subject to garbage collection, and may not be referenced from any low-level data structures.

**Simple Locks**

**primitive-make-simple-lock Primitive**

**Signature** (lock :: <portable-container>, name :: false-or(<byte-string>)) => ()

**Parameters**

- **lock** – An instance of <simple-lock>.
• **name** – The name of the lock (as a `<byte-string>`) or `#f`.

**Discussion** Creates a new OS lock and destructively modifies the container slot in the Dylan lock object with the handle of the new OS lock.

**primitive-destroy-simple-lock** **Primitive**

**Signature** `(lock :: <portable-container>) => ()`

**Parameters**

• **lock** – An instance of `<simple-lock>`.

**Discussion** Frees any runtime-allocated memory associated with the lock.

**primitive-wait-for-simple-lock** **Primitive**

**Signature** `(lock :: <portable-container>) => (error-code :: <integer>)`

**Parameters**

• **lock** – An instance of `<simple-lock>`.

**Values**

• **error-code** – 0 = ok

**Discussion** The calling thread blocks until the specified lock is available (unlocked) and then locks it. When the function returns, the lock is owned by the calling thread.

**primitive-wait-for-simple-lock-timed** **Primitive**

**Signature** `(lock :: <portable-container>, millisecs :: <integer>) => (error-code :: <integer>)`

**Parameters**

• **lock** – An instance of `<simple-lock>`.

• **millisecs** – Timeout period in milliseconds

**Values**

• **error-code** – 0 = ok, 1 = timeout expired

**Discussion** The calling thread blocks until either the specified lock is available (unlocked) or the timeout period expires. If the lock becomes available, this function locks it. If the function returns 0, the lock is owned by the calling thread, otherwise a timeout occurred.

**primitive-release-simple-lock** **Primitive**

**Signature** `(lock :: <portable-container>) => (error-code :: <integer>)`

**Parameters**

• **lock** – An instance of `<simple-lock>`.

**Values**

• **error-code** – 0 = ok, 2 = not locked

**Discussion** Unlocks the specified lock. The lock must be owned by the calling thread, otherwise the result indicates “not locked”.

**primitive-owned-simple-lock** **Primitive**

**Signature** `(lock :: <portable-container>) => (owned :: <integer>)`

**Parameters**

• **lock** – An instance of `<simple-lock>`.
Values

- **owned** – 0 = not owned, 1 = owned

**Discussion** Returns 1 if the specified lock is owned (locked) by the calling thread.

**Recursive Locks**

**primitive-make-recursive-lock** Primitive

**Signature** (lock :: <portable-container>, name :: false-or(<byte-string>)) => ()

**Parameters**

- **lock** – An instance of **recursive-lock**.
- **name** – The name of the lock (as a **byte-string**) or #f.

**Discussion** Creates a new OS lock and destructively modifies the container slot in the Dylan lock object with the handle of the new OS lock.

**primitive-destroy-recursive-lock** Primitive

**Signature** (lock :: <portable-container>) => ()

**Parameters**

- **lock** – An instance of **recursive-lock**.

**Discussion** Frees any runtime-allocated memory associated with the lock.

**primitive-wait-for-recursive-lock** Primitive

**Signature** (lock :: <portable-container>) => (error-code :: <integer>)

**Parameters**

- **lock** – An instance of **recursive-lock**.

**Values**

- **error-code** – 0 = ok

**Discussion** The calling thread blocks until the specified lock is available (unlocked or already locked by the calling thread). When the lock becomes available, this function claims ownership of the lock and increments the lock count. When the function returns, the lock is owned by the calling thread.

**primitive-wait-for-recursive-lock-timed** Primitive

**Signature** (lock :: <portable-container>, millisecs :: <integer>) => (error-code :: <integer>)

**Parameters**

- **lock** – An instance of **recursive-lock**.
- **millisecs** – Timeout period in milliseconds

**Values**

- **error-code** – 0 = ok, 1 = timeout expired

**Discussion** The calling thread blocks until the specified lock is available (unlocked or already locked by the calling thread). If the lock becomes available, this function claims ownership of the lock, increments an internal lock count, and returns 0. If a timeout occurs, the function leaves the lock unmodified and returns 1.
**primitive-release-recursive-lock Primitive**

Signature (lock :: <portable-container>) => (error-code :: <integer>)

Parameters
- **lock** – An instance of `<recursive-lock>`.

Values
- **error-code** – 0 = ok, 2 = not locked

Discussion Checks that the lock is owned by the calling thread, and returns 2 if not. If the lock is owned, its internal count is decremented by 1. If the count is then zero, the lock is then released.

**primitive-owned-recursive-lock Primitive**

Signature (lock :: <portable-container>) => (owned :: <integer>)

Parameters
- **lock** – An instance of `<recursive-lock>`.

Values
- **owned** – 0 = not owned, 1 = owned

Discussion Returns 1 if the specified lock is locked and owned by the calling thread.

### Semaphores

**primitive-make-semaphore Primitive**

Signature (lock :: <portable-container>, name :: false-or(<byte-string>), initial :: <integer>, max :: <integer>) => ()

Parameters
- **lock** – An instance of `<semaphore>`.
- **name** – The name of the lock (as a `<byte-string>`) or #f.
- **initial** – The initial value for the semaphore count.

Discussion Creates a new OS semaphore with the specified initial count and destructively modifies the container slot in the Dylan lock object with the handle of the new OS semaphore.

**primitive-destroy-semaphore Primitive**

Signature (lock :: <portable-container>) => ()

Parameters
- **lock** – An instance of `<semaphore>`.

Discussion Frees any runtime-allocated memory associated with the semaphore.

**primitive-wait-for-semaphore Primitive**

Signature (lock :: <portable-container>) => (error-code :: <integer>)

Parameters
- **lock** – An instance of `<semaphore>`.

Values
- **error-code** – 0 = ok
Discussion The calling thread blocks until the internal count of the specified semaphore becomes greater than zero. It then decrements the semaphore count.

**primitive-wait-for-semaphore-timed Primitive**

**Signature** (lock :: <portable-container>, millisecs :: <integer>) => (error-code :: <integer>)

**Parameters**
- lock – An instance of <semaphore>.
- millisecs – Timeout period in milliseconds

**Values**
- error-code – 0 = ok, 1 = timeout expired

**Discussion** The calling thread blocks until either the internal count of the specified semaphore becomes greater than zero or the timeout period expires. In the former case, the function decrements the semaphore count and returns 0. In the latter case, the function returns 1.

**primitive-release-semaphore Primitive**

**Signature** (lock :: <portable-container>) => (error-code :: <integer>)

**Parameters**
- lock – An instance of <semaphore>.

**Values**
- error-code – 0 = ok, 3 = count exceeded

**Discussion** This function checks that internal count of the semaphore is not at its maximum limit, and returns 3 if the test fails. Otherwise the internal count is incremented.

**Notifications**

**primitive-make-notification Primitive**

**Signature** (notification :: <portable-container>, name :: false-or(<byte-string>)) => ()

**Parameters**
- notification – An instance of <notification>.
- name – The name of the notification (as a <byte-string>) or #f.

**Discussion** Creates a new OS notification (condition variable) and destructively modifies the container slot in the Dylan lock object with the handle of the new OS notification.

**primitive-destroy-notification Primitive**

**Signature** (notification :: <portable-container>) => ()

**Parameters**
- notification – An instance of <notification>.

**Discussion** Frees any runtime-allocated memory associated with the notification.

**primitive-wait-for-notification Primitive**

**Signature** (notification :: <portable-container>, lock :: <portable-container>) => (error-code :: <integer>)

**Parameters**
• **notification** – An instance of `<notification>`.
• **lock** – An instance of `<simple-lock>`.

**Values**

• **error-code** – 0 = ok, 2 = not locked, 3 = other error

**Discussion** The function checks that the specified lock is owned by the calling thread, and returns 2 if the test fails. Otherwise, the calling thread atomically releases the lock and then blocks, waiting to be notified of the condition represented by the specified notification. When the calling thread is notified of the condition, the function reclaims ownership of the lock, blocking if necessary, before returning 0.

**primitive-wait-for-notification-timed Primitive**

**Signature**

```
(notification :: <portable-container>, lock :: <portable-container>, millisecs :: <integer>) => (error-code :: <integer>)
```

**Parameters**

• **notification** – An instance of `<notification>`.
• **lock** – An instance of `<simple-lock>`.
• **millisecs** – Timeout period in milliseconds

**Values**

• **error-code** – 0 = ok, 1 = timeout, 2 = not locked, 3 = other error

**Discussion** The function checks that the specified lock is owned by the calling thread, and returns 2 if the test fails. Otherwise, the calling thread atomically releases the lock and then blocks, waiting to be notified of the condition represented by the specified notification, or for the timeout period to expire. The function then reclaims ownership of the lock, blocking indefinitely if necessary, before returning either 0 or 1 to indicate whether a timeout occurred.

**primitive-release-notification Primitive**

**Signature**

```
(notification :: <portable-container>, lock :: <portable-container>) => (error-code :: <integer>)
```

**Parameters**

• **notification** – An instance of `<notification>`.
• **lock** – An instance of `<simple-lock>`.

**Values**

• **error-code** – 0 = ok, 2 = not locked

**Discussion** If the calling thread does not own the specified lock, the function returns the error value 2. Otherwise, the function releases the specified notification, notifying another thread that is blocked waiting for the notification to occur. If more than one thread is waiting for the notification, it is unspecified which thread is notified. If no threads are waiting, then the release has no effect.

**primitive-release-all-notification Primitive**

**Signature**

```
(notification :: <portable-container>, lock :: <portable-container>) => (error-code :: <integer>)
```

**Parameters**

• **notification** – An instance of `<notification>`.

---

8.2. The dylan-primitives Module 195
• **lock** – An instance of `<simple-lock>`.

**Values**

• **error-code** – 0 = ok, 2 = not locked

**Discussion** If the calling thread does not own the specified lock, the function returns the error value 2. Otherwise, the function releases the specified notification, notifying all other threads that are blocked waiting for the notification to occur. If no threads are waiting, then the release has no effect.

## Timers

### primitive-sleep Primitive

**Signature**  
(millisecs :: <integer>) => ()

**Parameters**

• **millisecs** – Time interval in milliseconds

**Discussion** This function causes the calling thread to block for the specified time interval.

## Thread Variables

### primitive-allocate-thread-variable Primitive

**Signature**  
(initial-value) => (handle-on-variable)

**Parameters**

• **initial-value** – A Dylan object that is to be the initial value of the fluid variable.

**Values**

• **handle-on-variable** – An OS handle on the fluid variable, to be stored as the immediate value of the variable. Variable reading and assignment will indirect through this handle. The handle is not a Dylan object.

**Discussion** This function creates a new thread-local variable handle, and assigns the specified initial value to the location indicated by the handle. The function must arrange to assign the initial value to the thread-local location associated with all other existing threads, too. The function must also arrange that whenever a new thread is subsequently created, it also has its thread-local location indicated by the handle set to the initial value.

## Simple Runtime Primitives

**primitive_allocate**(int size)

This is the interface to the memory allocator which might be dependent on the garbage collector. It takes a size in bytes as a parameter, and returns some freshly allocated memory which the run-time system knows how to memory-manage.

**primitive_byte_allocate**(int word-size, int byte-size)

This is built on the same mechanism as `primitive_allocate()`, but it is specifically designed for allocating objects which have Dylan slots, but also have a repeated slot of byte-sized elements, such as a byte string, or a byte vector. It takes two parameters, a size in `words` for the object slots (e.g., one for `class` and a second for `size`), followed by the number of bytes for the vector. The value returned from the primitive is the freshly allocated memory making up the string.
D\text{primitive\_fill\_E\_}(D\text{ storage[]}, \text{int } \text{size}, D\text{ value})

(The odd name is a result of name mangling from \text{primitive\_fill!). This takes a Dylan object (or a
pointer to the middle of one), a size, and a value. It inserts the value into as many slots as are specified by \text{size}.

D\text{primitive\_replace\_E\_}(D\text{ dst[]}, D\text{ src[]}, \text{int } \text{size})

(See \text{primitive\_fill\_E\_} re. name). This copies from the source vector into the destination vector as
many values as are specified in the \text{size} parameter.

D\text{primitive\_replace\_vector\_E\_}(SOV\text* } \text{dest}, SOV\text* } \text{source})

This is related to \text{primitive\_replace\_E\_}, except that the two arguments are guaranteed to be simple
object vectors, and they are self-sizing. It takes two parameters, ‘\text{dest}’, and ‘\text{source}’, and the data from ‘\text{source}’
is copied into ‘\text{dest}’. ‘\text{Dest}’ is returned.

D\text{primitive\_allocate\_vector}( \text{int } \text{size})

This is related to \text{primitive\_allocate()}, except that it takes a ‘\text{size}’ argument, which is the size of
repeated slots in a simple object vector (SOV). An object which is big enough to hold the specified indices is
allocated, and appropriately initialized, so that the ‘class’ field shows that it is an SOV, and the ‘\text{size}’ field shows
how big it is.

D\text{primitive\_copy\_vector}(D\text{ vector})

This takes a SOV as a parameter, and allocates a fresh SOV of the same size. It copies all the data that was
supplied from the old one to the new one, and returns the new one.

D\text{primitive\_initialize\_vector\_from\_buffer}(SOV\text* } \text{vector}, \text{int } \text{size}, D\text* } \text{buffer})

This primitive takes a pre-existing vector, and copies data into it from a buffer so as to initialize an SOV. The
primitive takes a SOV to be updated, a ‘\text{size}’ parameter (the specified size of the SOV), and a pointer to a buffer
which will supply the necessary data. The class and size values for the new SOV are set, and the data written to
the rest of the SOV. The SOV is returned.

D\text{primitive\_make\_string}(char\text* } \text{string})

This takes as a parameter a ‘C’ string with is zero-terminated, and returns a Dylan string with the same data
inside it.

D\text{primitive\_continue\_unwind}()

This is used as the last thing to be done at the end of an unwind-protect cleanup. It is responsible for determining
why the cleanup is being called, and thus taking appropriate action afterwards.

It handles 2 basic cases:

• a non-local exit
• a normal unwind-protect

In the first case we wish to transfer control back to some other location, but there is a cleanup that needs to be
done first. In this case there will be an unwind-protect frame on the stack which contains a marker to identify
the target of the non-local exit. Control can thus be transferred, possibly invoking another unwind-protect on
the way.

Alternatively, no transfer of control may be required, and unwind-protect can proceed normally. As a result of
evaluating our protected forms, the multiple values of these forms are stored in the unwind-protect frame. These
values are put back in the multiple values area, and control is returned.

D\text{primitive\_nlx}(Bind\text\_exit\_frame\text* } \text{target}, SOV\text* } \text{arguments})

This takes two parameters: a bind-exit frame which is put on the stack whenever a bind-exit frame is bound,
and an SOV of the multiple values that we wish to return to that bind-exit point. We then step to the bind-exit
frame target, while checking to see if there are any intervening unwind-protect frames. If there are, we put the
marker for our ultimate destination into the unwind-protect frame that has been detected on the stack between us
and our destination. The multiple values we wish to return are put into the unwind-protect frame. The relevant
cleanup code is invoked, and at the end of this a \text{primitive\_continue\_unwind()} should be called. This
should detect that there is further to go, and insert the multiple values into any intervening frames.
D\texttt{primitive\_inlined\_nlx} (Bind\_exit\_frame* target, D\ first\_argument)

This is similar to \texttt{primitive\_nlx()}, except that it is used when the compiler has been able to gain more information about the circumstances in which the non-local-exit call is happening. In particular it is used when it is possible to in-line the call, so that the multiple values that are being passed are known to be in the multiple values area, rather than having been created as an SOV. An SOV has to be built up from these arguments.

D* \texttt{primitive\_make\_box} (D\ object)

A box is a value-cell that is used for closed-over variables which are subject to assignment. The function takes a Dylan object, and returns a value-cell box which contains the object. The compiler deals with the extra level of indirection needed to get the value out of the box.

D* \texttt{primitive\_make\_environment} (int size, \ldots)

This is the function which makes the vector which is used in a closure. The arguments to this are either boxes, or normal Dylan objects. This takes an argument of ‘size’ for the initial arguments to be closed over, plus the arguments themselves. ‘Size’ arguments are built up into an SOV which is used as an environment.

**Entry Point Functions**

D \texttt{xep\_0} (FN* function, int argument\_count)

D \texttt{xep\_1} (FN* function, int argument\_count)

D \texttt{xep\_2} (FN* function, int argument\_count)

D \texttt{xep\_3} (FN* function, int argument\_count)

D \texttt{xep\_4} (FN* function, int argument\_count)

D \texttt{xep\_5} (FN* function, int argument\_count)

D \texttt{xep\_6} (FN* function, int argument\_count)

D \texttt{xep\_7} (FN* function, int argument\_count)

D \texttt{xep\_8} (FN* function, int argument\_count)

D \texttt{xep\_9} (FN* function, int argument\_count)

These are the XEP entry-point handlers for those Dylan functions which do not accept optional parameters. Each Dylan function has an external (safe) entry point with full checking. After checking, this calls the internal entry point, which is the most efficient available.

The compiler itself only ever generates code for the internal entry point. Any value put into the external entry point field of an object is a shared value provided by the runtime system. If the function takes no parameters, the value will be \texttt{xep0}; if it takes a single required parameter it will be \texttt{xep1}, and so on. There are values available for \texttt{xep0} to \texttt{xep9}. For more than nine required parameters, the \texttt{xep()} function is used.

\texttt{xep} (FN* function, int argument\_count, \ldots)

If the function takes more than nine required parameters, then the function will simply be called \texttt{xep}, the general function which will work in all such cases. The arguments are passed as ‘varargs’. This function will check the number of arguments, raising an error if it is wrong. It then sets the calling convention for calling the internal entry point. This basically means that the function register is appropriately set, and the implementation ‘mlist’ parameter is set to \#f.

D \texttt{optional\_xep} (FN* function, int argument\_count, \ldots)

This function is used as the XEP code for any Dylan function which has optional parameters. In this case, the external entry point conventions do not require the caller to have any knowledge of where the optionals start. The XEP code is thus responsible for separating the code into those which are required parameters, to be passed via the normal machine conventions, and those which are optionals, to be passed as a Dylan SOV. If the function object takes keywords, all the information about which keywords are accepted is stored in the function itself. The vector of optional parameters is scanned by the XEP code to see if any appropriate ones have been supplied.
If one is found, then the associated value is taken and used as an implicit parameter to the internal entry point. If a value is not supplied, then a suitable default parameter which is stored inside the function object is passed instead.

\[ \text{gf\_xep\_0} (\text{FN}* \text{function}, \text{int} \text{argument\_count}) \]
\[ \text{gf\_xep\_1} (\text{FN}* \text{function}, \text{int} \text{argument\_count}) \]
\[ \text{gf\_xep\_2} (\text{FN}* \text{function}, \text{int} \text{argument\_count}) \]
\[ \text{gf\_xep\_3} (\text{FN}* \text{function}, \text{int} \text{argument\_count}) \]
\[ \text{gf\_xep\_4} (\text{FN}* \text{function}, \text{int} \text{argument\_count}) \]
\[ \text{gf\_xep\_5} (\text{FN}* \text{function}, \text{int} \text{argument\_count}) \]
\[ \text{gf\_xep\_6} (\text{FN}* \text{function}, \text{int} \text{argument\_count}) \]
\[ \text{gf\_xep\_7} (\text{FN}* \text{function}, \text{int} \text{argument\_count}) \]
\[ \text{gf\_xep\_8} (\text{FN}* \text{function}, \text{int} \text{argument\_count}) \]
\[ \text{gf\_xep\_9} (\text{FN}* \text{function}, \text{int} \text{argument\_count}) \]

These primitives are similar to \text{xep\_0()} through \text{xep\_9()}, but deal with the entry points for generic functions. Generic functions do not require the ‘mlist’ parameter to be set, so a special optimized entry point is provided. These versions are for 0 - 9 required parameters. These functions call the internal entry point.

\[ \text{gf\_xep} (\text{FN}* \text{function}, \text{int} \text{argument\_count}, \ldots) \]

This primitive is similar to \text{xep()}, but deals with the entry points for generic functions. Generic functions do not require the ‘mlist’ parameter to be set, so a special optimized entry point is provided. This is the general version for functions which do not take optional arguments. This function calls the internal entry point.

\[ \text{gf\_optional\_xep} (\text{FN}* \text{function}, \text{int} \text{argument\_count}, \ldots) \]

This is used for all generic functions which take optional arguments. This function calls the internal entry point.

\[ \text{primitive\_basic\_iep\_apply} (\text{FN}* \text{f}, \text{int} \text{argument\_count}, \text{D} \text{a}[]) \]

This is used to call internal entry points. It takes three parameters: a Dylan function object (where the iep is stored in a slot), an argument count of the number of arguments that we are passing to the iep, and a vector of all of these arguments. This is a ‘basic’ IEP apply because is does no more than check the argument count, and call the IEP with the appropriate number of Dylan parameters. It does not bother to set any implementation parameters. Implementation parameters which could be set in by other primitives are ‘function’, and a ‘mlist’ (the list of next-methods). Not all IEPs care about the ‘function’ or ‘mlist’ parameters, but when the compiler calls \text{primitive\_basic\_iep\_apply()}, it has to make sure that any necessary ‘function’ or ‘mlist’ parameters have been set up.

\[ \text{primitive\_iep\_apply} (\text{FN}* \text{f}, \text{int} \text{argument\_count}, \text{D} \text{a}[]) \]

This is closely related to \text{primitive\_basic\_iep\_apply()}. It takes the same number of parameters, but it sets the explicit, implementation-dependent function parameter which is usually set to the first argument, and also sets the ‘mlist’ argument to ‘false’. This is the normal case when a method object is being called directly, rather than as part of a generic function.

\[ \text{primitive\_xep\_apply} (\text{FN}* \text{f}, \text{int} \text{argument\_count}, \text{D} \text{a}[]) \]

This is a more usual usage of apply, i.e., the standard Dylan calling convention being invoked by \text{apply}. It takes three parameters: the Dylan function to be called, the number of arguments being passed, and a vector containing all those arguments. This primitive relates to the external entry point for the function, and guarantees full type checking and argument count checking. This primitive does all that is necessary to conform with the xep calling convention of Dylan: i.e., it sets the ‘function’ parameter, it sets the argument count, and then calls the XEP for the function.
8.2.2 Compiler Primitives

**General Primitives**

**primitive-make-box Primitive**
Signature (object :: <object>) => <object>

**primitive-allocate Primitive**
Signature (size :: <raw-small-integer>) => <object>)

**primitive-byte-allocate Primitive**
Signature (word-size :: <raw-small-integer>, byte-size :: <raw-small-integer>) => <object>)

**primitive-make-environment Primitive**
Signature (size :: <raw-small-integer>) => <object>

**primitive-copy-vector Primitive**
Signature (vector :: <object>) => <object>

**primitive-make-string Primitive**
Signature (vector :: <raw-c-char*>) => <raw-c-char*>

**primitive-function-code Primitive**
Signature (function :: <object>) => <object>

**primitive-function-environment Primitive**
Signature (function :: <object>) => <object>

**Low-Level Apply Primitives**

**primitive-xep-apply Primitive**
Signature (function :: <object>, buffer-size :: <raw-small-integer>, buffer :: <object>) => :: <object>

**primitive-iep-apply Primitive**
Signature (function :: <object>, buffer-size :: <raw-small-integer>, buffer :: <object>) => <object>)

**primitive-true? Primitive**
Signature (value :: <raw-small-integer>) => <object>

Discussion
This primitive returns Dylan true if value is non-zero, and false if value is zero.

**primitive-false? Primitive**
Signature (value :: <raw-small-integer>) => <object>

Discussion
This is the complement of primitive-true?, returning #t if the value is 0, #f otherwise.

**primitive-equals? Primitive**
Signature (x :: <object>, y :: <object>) => <raw-c-int>
primitive-continue-unwind Primitive
  Signature () => <object>

primitive-nlx Primitive
  Signature (bind-exit-frame :: <raw-c-void*>, args :: <raw-c-void*>) => <raw-c-void>

primitive-inlined-nlx Primitive
  Signature (bind-exit-frame :: <raw-c-void*>, first-argument :: <raw-c-void*>) => <raw-c-void>

primitive-variable-lookup Primitive
  Signature (variable-pointer :: <raw-c-void*>) => <raw-c-void*>

primitive-variable-lookup-setter Primitive
  Signature (value :: <raw-c-void*>, variable-pointer :: <raw-c-void*>) => <raw-c-void*>

Integer Primitives

primitive-int? Primitive
  Signature (x :: <object>) => <raw-small-integer>

primitive-address-equals? Primitive
  Signature (x :: <raw-address>, y :: <raw-address>) => <raw-address>

primitive-address-add Primitive
  Signature (x :: <raw-address>, y :: <raw-address>) => <raw-address>

primitive-address-subtract Primitive
  Signature (x :: <raw-address>, y :: <raw-address>) => <raw-address>

primitive-address-multiply Primitive
  Signature (x :: <raw-address>, y :: <raw-address>) => <raw-address>

primitive-address-left-shift Primitive
  Signature (x :: <raw-address>, y :: <raw-address>) => <raw-address>

primitive-address-right-shift Primitive
  Signature (x :: <raw-address>, y :: <raw-address>) => <raw-address>

primitive-address-not Primitive
  Signature (x :: <raw-address>) => <raw-address>

primitive-address-and Primitive
  Signature (x :: <raw-address>, y :: <raw-address>) => <raw-address>

primitive-address-or Primitive
  Signature (x :: <raw-address>, y :: <raw-address>) => <raw-address>

primitive-small-integer-equals? Primitive
  Signature (x :: <raw-small-integer>, y :: <raw-small-integer>) => <raw-small-integer>

primitive-small-integer-not-equals? Primitive
  Signature (x :: <raw-small-integer>, y :: <raw-small-integer>) => <raw-small-integer>
primitive-small-integer-less-than? Primitive
  Signature (x :: <raw-small-integer>, y :: <raw-small-integer>) => <raw-small-integer>
primitive-small-integer-greater-than? Primitive
  Signature (x :: <raw-small-integer>, y :: <raw-small-integer>) => <raw-small-integer>
primitive-small-integer-greater-than-or-equal? Primitive
  Signature (x :: <raw-small-integer>, y :: <raw-small-integer>) => <raw-small-integer>
primitive-small-integer-negate Primitive
  Signature (x :: <raw-small-integer>) => <raw-small-integer>
primitive-small-integer-add Primitive
  Signature (x :: <raw-small-integer>, y :: <raw-small-integer>) => <raw-small-integer>
primitive-small-integer-subtract Primitive
  Signature (x :: <raw-small-integer>, y :: <raw-small-integer>) => <raw-small-integer>
primitive-small-integer-multiply Primitive
  Signature (x :: <raw-small-integer>, y :: <raw-small-integer>) => <raw-small-integer>
primitive-small-integer-divide Primitive
  Signature (x :: <raw-small-integer>, y :: <raw-small-integer>) => <raw-small-integer>
primitive-small-integer-modulo Primitive
  Signature (x :: <raw-small-integer>, y :: <raw-small-integer>) => <raw-small-integer>
primitive-small-integer-left-shift Primitive
  Signature (x :: <raw-small-integer>, y :: <raw-small-integer>) => <raw-small-integer>
primitive-small-integer-right-shift Primitive
  Signature (x :: <raw-small-integer>, y :: <raw-small-integer>) => <raw-small-integer>
primitive-small-integer-not Primitive
  Signature (x :: <raw-small-integer>) => <raw-small-integer>
primitive-small-integer-and Primitive
  Signature (x :: <raw-small-integer>, y :: <raw-small-integer>) => <raw-small-integer>
primitive-small-integer-or Primitive
  Signature (x :: <raw-small-integer>, y :: <raw-small-integer>) => <raw-small-integer>
primitive-small-integer-xor Primitive
  Signature (x :: <raw-small-integer>, y :: <raw-small-integer>) => <raw-small-integer>

In addition to the small-integer operators above, there are also definitions for three other integer types, defined in the same manner. The following table summarizes the relationship between these types and Dylan primitives.

Integer Types and Dylan Primitives
General Variety of Integer | Class of Primitive Parameters and Return Values | Value of type in Primitive Name primitive-type-operator
--- | --- | ---
Small Integer | `<raw-small-integer>` | `small-integer`
Big Integer | `<raw-big-integer>` | `big-integer`
Machine Integer | `<raw-machine-integer>` | `machine-integer`
Unsigned Machine Integer | `<raw-unsigned-machine-integer>` | `unsigned-machine-integer`

Float Primitives

**primitive-decoded-bits-as-single-float Primitive**

**Signature** (sign :: `<raw-small-integer>`, exponent :: `<raw-small-integer>`, significand :: `<raw-small-integer>`) => `<raw-single-float>`

**primitive-bits-as-single-float Primitive**

**Signature** (x :: `<raw-small-integer>`) => `<raw-single-float>`

**Discussion** Uses a custom emitter to map to a call to a function called `integer_to_single_float` in the runtime system.

**primitive-single-float-as-bits Primitive**

**Signature** (x :: `<raw-single-float>`) => `<raw-small-integer>`

**Discussion** Uses a custom emitter to map to a call to a function called `single_float_to_integer` in the runtime system.

**primitive-single-float-equals? Primitive**

**Signature** (x :: `<raw-single-float>`, y :: `<raw-single-float>`) => `<raw-c-int>`

**primitive-single-float-not-equals? Primitive**

**Signature** (x :: `<raw-single-float>`, y :: `<raw-single-float>`) => `<raw-c-int>`

**primitive-single-float-less-than? Primitive**

**Signature** (x :: `<raw-single-float>`, y :: `<raw-single-float>`) => `<raw-c-int>`

**primitive-single-float-less-than-or-equal? Primitive**

**Signature** (x :: `<raw-single-float>`, y :: `<raw-single-float>`) => `<raw-c-int>`

**primitive-single-float-greater-than? Primitive**

**Signature** (x :: `<raw-single-float>`, y :: `<raw-single-float>`) => `<raw-c-int>`

**primitive-single-float-greater-than-or-equal? Primitive**

**Signature** (x :: `<raw-single-float>`, y :: `<raw-single-float>`) => `<raw-c-int>`

**primitive-single-float-negate Primitive**

**Signature** (x :: `<raw-single-float>`) => `<raw-single-float>`

**primitive-single-float-add Primitive**

**Signature** (x :: `<raw-single-float>`, y :: `<raw-single-float>`) => `<raw-single-float>`

**primitive-single-float-subtract Primitive**

**Signature** (x :: `<raw-single-float>`, y :: `<raw-single-float>`) => `<raw-single-float>`

8.2. The dylan-primitives Module
primitive-single-float-multiply Primitive
     Signature (x :: <raw-single-float>, y :: <raw-single-float>) => <raw-single-float>

primitive-single-float-divide Primitive
     Signature (x :: <raw-single-float>, y :: <raw-single-float>) => <raw-single-float>

primitive-single-float-unary-divide Primitive
     Signature (x :: <raw-single-float>) => <raw-single-float>

Accessor Primitives

primitive-element Primitive
     Signature (array :: <object>, index :: <raw-small-integer>) => <object>
     Discussion This is used for de-referencing slots in the middle of Dylan objects, and thus potentially
invokes read-barrier code. It takes two parameters: a Dylan object, and an index which is the
‘word’ index into the object. It returns the Dylan value found in that corresponding slot.

primitive-element-setter Primitive
     Signature (new-value :: <object>, array :: <object>, index :: <raw-small-integer>) => <object>
     Discussion This is the assignment operator corresponding to primitive-element, which is used to
change the value of a Dylan slot. This takes an extra initial parameter which is the new value to
put into the object. The new value is stored in the appropriate object at the given index.

primitive-byte-element Primitive
     Signature (array <object>, base-index :: <raw-small-integer>, byte-offset :: <raw-small-integer>) => <raw-c-char>
     Discussion This is similar to primitive-element, but deals with byte vectors. It takes a new value and
a Dylan object, along with a base offset and a byte offset. The base offset, expressed in words,
and the byte offset, expressed in bytes, are added, and the byte found at that location is returned.

primitive-byte-element-setter Primitive
     Signature (new-value :: <raw-c-char>) array :: <object>, base-index :: <raw-small-integer>, byte-
offset :: <raw-small-integer>) => <raw-c-char>
     Discussion This is the corresponding setter for primitive-byte-element.

primitive-fill! Primitive
     Signature (array :: <object>, size :: <raw-small-integer>, value :: <object>) => <object>

primitive-replace! Primitive
     Signature (new-array :: <object>, array :: <object>, size :: <raw-small-integer>) => <object>

primitive-replace-bytes! Primitive
     Signature (dst :: <raw-c-void*>, src :: <raw-c-void*>, size :: <raw-c-int>) => <raw-c-void>

The following primitives, named primitive-type-at and primitive-type-at-setter load or store, respectively, a value of
the designated type at the specified address.

primitive-untyped-at Primitive
     Signature (address :: <raw-pointer>) => <raw-untyped>

primitive-untyped-at-setter Primitive
8.2. The dylan-primitives Module

Signature (new-value :: <raw-untyped>, address :: <raw-pointer>) => <raw-untyped>

primitive-pointer-at Primitive
Signature (address :: <raw-pointer>) => <raw-pointer>

primitive-pointer-at-setter Primitive
Signature (new-value :: <raw-pointer>, address :: <raw-pointer>) => <raw-pointer>

primitive-byte-character-at Primitive
Signature (address :: <raw-pointer>) => <raw-byte-character>

primitive-byte-character-at-setter Primitive
Signature (new-value :: <raw-byte-character>, address :: <raw-pointer>) => <raw-byte-character>

primitive-small-integer-at Primitive
Signature (address :: <raw-pointer>) => <raw-small-integer>

primitive-small-integer-at-setter Primitive
Signature (new-value :: <raw-small-integer>, address :: <raw-pointer>) => <raw-small-integer>

primitive-big-integer-at Primitive
Signature (address :: <raw-pointer>) => <raw-big-integer>

primitive-big-integer-at-setter Primitive
Signature (new-value :: <raw-big-integer>, address :: <raw-pointer>) => <raw-big-integer>

primitive-machine-integer-at Primitive
Signature (address :: <raw-pointer>) => <raw-machine-integer>

primitive-machine-integer-at-setter Primitive
Signature (new-value :: <raw-machine-integer>, address :: <raw-pointer>) => <raw-machine-integer>

primitive-unsigned-machine-integer-at Primitive
Signature (address :: <raw-pointer>) => <raw-unsigned-machine-integer>

primitive-unsigned-machine-integer-at-setter Primitive
Signature (new-value :: <raw-unsigned-machine-integer>, address :: <raw-pointer>) => <raw-unsigned-machine-integer>

primitive-single-float-at Primitive
Signature (address :: <raw-pointer>) => <raw-single-float>

primitive-single-float-at-setter Primitive
Signature (new-value :: <raw-single-float>, address :: <raw-pointer>) => <raw-single-float>

primitive-double-float-at Primitive
Signature (address :: <raw-pointer>) => <raw-double-float>

primitive-double-float-at-setter Primitive
Signature (new-value :: <raw-double-float>, address :: <raw-pointer>) => <raw-double-float>

primitive-extended-float-at Primitive
Signature (address :: <raw-pointer>) => <raw-extended-float>
primitive-extended-float-at-setter Primitive

Signature (new-value :: <raw-extended-float>, address :: <raw-pointer>) => <raw-extended-float>

primitive-signed-8-bit-integer-at Primitive

Signature (address :: <raw-pointer>) => <raw-signed-8-bit-integer>

primitive-signed-8-bit-integer-at-setter Primitive

Signature (new-value :: <raw-signed-8-bit-integer>, address :: <raw-pointer>) => <raw-signed-8-bit-integer>

primitive-unsigned-8-bit-integer-at Primitive

Signature (address :: <raw-pointer>) => <raw-unsigned-8-bit-integer>

primitive-unsigned-8-bit-integer-at-setter Primitive

Signature (new-value :: <raw-unsigned-8-bit-integer>, address :: <raw-pointer>) => <raw-unsigned-8-bit-integer>

primitive-signed-16-bit-integer-at Primitive

Signature (address :: <raw-pointer>) => <raw-signed-16-bit-integer>

primitive-signed-16-bit-integer-at-setter Primitive

Signature (new-value :: <raw-signed-16-bit-integer>, address :: <raw-pointer>) => <raw-signed-16-bit-integer>

primitive-unsigned-16-bit-integer-at Primitive

Signature (address :: <raw-pointer>) => <raw-unsigned-16-bit-integer>

primitive-unsigned-16-bit-integer-at-setter Primitive

Signature (new-value :: <raw-unsigned-16-bit-integer>, address :: <raw-pointer>) => <raw-unsigned-16-bit-integer>

primitive-signed-32-bit-integer-at Primitive

Signature (address :: <raw-pointer>) => <raw-signed-32-bit-integer>

primitive-signed-32-bit-integer-at-setter Primitive

Signature (new-value :: <raw-signed-32-bit-integer>, address :: <raw-pointer>) => <raw-signed-32-bit-integer>

primitive-unsigned-32-bit-integer-at Primitive

Signature (address :: <raw-pointer>) => <raw-unsigned-32-bit-integer>

primitive-unsigned-32-bit-integer-at-setter Primitive

Signature (new-value :: <raw-unsigned-32-bit-integer>, address :: <raw-pointer>) => <raw-unsigned-32-bit-integer>

primitive-signed-64-bit-integer-at Primitive

Signature (address :: <raw-pointer>) => <raw-signed-64-bit-integer>

primitive-signed-64-bit-integer-at-setter Primitive

Signature (new-value :: <raw-signed-64-bit-integer>, address :: <raw-pointer>) => <raw-signed-64-bit-integer>

primitive-unsigned-64-bit-integer-at Primitive
8.3 The threads Module

The Threads module provides a portable threads interface for Dylan. The Threads module is designed to map easily and efficiently onto the threads facilities provided by all common operating systems.

All documented bindings are exported from the module threads in the dylan and common-dylan libraries.

Contents

- The threads Module
  - Multi-thread semantics
    - Atomicity
    - Ordering
    - Explicit synchronization
    - Conditional update
    - The dynamic environment
    - Thread variables
    - Dynamic binding
  - Thread safety in client libraries
8.3.1 Multi-thread semantics

The Threads module provides multiple threads of control within a single space of objects and module variables. Each thread runs in its own independent stack. The mechanism by which the threads are scheduled is not specified, and it is not possible to determine how the execution of instructions by different threads will be interleaved. No mechanism is provided to call a function on an existing thread other than the current thread. Neither is there a mechanism to signal an exception on a thread other than the current thread.

Atomicity

In general, the Threads module guarantees that assignments to slots and variables are atomic. That is, after an assignment, but before synchronization, another thread will see either the old value or the new value of the location. There is no possibility of seeing a half-way state.

In some circumstances, when a slot or a variable is specialized to be of a particularly constrained type, the Threads module does not guarantee atomicity of assignments. Such a type may include a subtype of `<double-float>` or a subtype of `<extended-float>`. It may not include any other type that is either defined in the current specification of the Dylan language, or that could be created from standard facilities provided by the current specification of the
language. This restriction of the atomicity guarantee is intended to permit implementations to represent the values of such slots or variables in a form which uses more space than a normal Dylan value, for optimal efficiency.

For those cases where the implementation does not provide the atomicity guarantee, the results of accessing a normal variable are undefined if:

- The read could proceed in parallel with some write of the same location
- Two writes of the same location could have proceeded in parallel since the last non-parallel write

Two memory references proceed in parallel if they are not explicitly sequentialized, either by being in a single thread, or by explicit inter-thread synchronization.

Programmers should guard against the possibility of undefined values by using explicit inter-thread synchronization.

**Ordering**

The ordering of visibility of side effects performed in other threads is undefined, unless explicit synchronization is used. Implementations of the module may guarantee that the visibility of side-effects performed by another thread is ordered according to the control flow of the other thread (strong ordering), but multi-processor implementations might not be strongly ordered. Portable code should not assume strong ordering, and should use explicit synchronization where the order of side effects is important. There is currently no module introspection facility to determine if the implementation is strongly or weakly ordered.

Because of the possibility of weak ordering, the compiler is free to assume that the effects of other threads may be ignored between explicit synchronization points, and it may perform any optimizations which preserve the semantics of a single-thread model regardless of their effects on other threads — for example, common sub-expression elimination, or changing the order of evaluation.

**Explicit synchronization**

The Threads module provides low-level synchronization functions which control the ordering of operations with respect to other threads, and control when the side effects that have been performed within one thread become visible within other threads.

At a higher level, the Threads module provides a variety of synchronization facilities, described below. These facilities include mutual-exclusion locks, semaphores and notifications. Each facility guarantees that when synchronization has been achieved, all the side effects of another thread are visible, at least up to the point where that other thread last released the synchronization facility.

An appropriate synchronization must be used to guard side-effects on state if there is any possibility of those side-effects either being corrupted by another thread or corrupting another thread. For example, a function which assigns to two slots of an object may require the use of a lock to guarantee that other threads never observe the object in a partly updated state.

It is up to module designers to document when synchronization is not performed internally, and when synchronization protocols must be used by clients. The implications for the Dylan library, and some other low-level libraries, are discussed in *Thread safety in client libraries*.

**Conditional update**

In addition to the synchronization primitives, the module provides a conditional update mechanism which is not synchronized, but which tests whether the value in a variable or slot has changed and atomically updates it if not.

By using conditional updates, a thread can confirm (or deny) that there has been no interference from other threads, without any need for a blocking operation. This is more efficient for those circumstances where interference is not disastrous and it is possible to recompute the update.
For example, a function which increments the value of a variable might use a conditional update to store the new value into place, in order to guarantee a numeric sequence for the variable. In this example, the function might loop until the conditional update has succeeded.

It is possible to achieve synchronization by looping until a conditional update is successful, and then synchronizing side effects. This is not recommended, because the busy-waiting state during the loop may disallow other threads from running. Normally, conditional update should be used only when it is expected to succeed. If it is likely that the conditional update might fail multiple times around the loop, then either the number of times around the loop should be limited, or a blocking function from the Threads module should be used within the loop.

**The dynamic environment**

Dylan has an implicit notion of a *dynamic environment*, corresponding to language constructs with *dynamic extent*. For example, the `block` construct can introduce `cleanup-clauses`, and the *body* of the block is executed in a dynamic environment in which those cleanup-clauses are active. *Handlers* and *exit procedures* are other examples of language features related to the dynamic environment.

The dynamic environment is defined to be thread-local. When a new thread is created, it starts with a fresh dynamic environment. It is an error to attempt to use a handler or a non-local exit function belonging to another thread. It is impossible to use an unwind-protect cleanup from another thread.

Although the binding of condition handlers only affects the dynamic environment of the current thread, unhandled conditions are passed to the global generic function `default-handler`. This function might *call the debugger*. The Threads module does not define what calling the debugger means.

Note that in Dylan, unlike in C and C++, *lexical* variables (that is, local, or `let`-bound variables) have indefinite extent — that is, have a lifetime independent of the function or block in which they were created — and are not bound in the dynamic environment. Because those variables are in general potentially global, you may need to explicitly synchronize accesses to them.

**Thread variables**

The Threads module provides a new type of variable: a *thread* variable, also known as a *thread-local* variable. These variables are similar to normal module variables in the sense that they are visible according to the same scoping rules and have the same semantics in a single-threaded program. However, in contrast to a normal variable, assignments to a thread variable in one thread are not visible when evaluating the variable in another thread.

Whenever a thread is created, the value of each thread variable is initialized to a thread-independent value resulting from a once-only evaluation of the initialization expression of the thread variable definition.

See page *thread* for details of the *thread* adjective to define variable.

**Dynamic binding**

The Threads module exports a macro for dynamic binding. A *binding* is a mapping between a variable and a `value-cell` which holds the variable's value. A *dynamic* binding is a binding which has dynamic extent, and shadows any outermost bindings. Dynamic bindings can be considered to be a property of the dynamic environment.

Thread variables can have new dynamic bindings created for them with the macro `dynamic-bind`. Thread variables inherently have thread-local bindings, so it is possible to re-bind a thread variable dynamically using the Dylan construct `*block* ... *cleanup*`. The `dynamic-bind` macro can be implemented in this way.

The thread-local nature of dynamically bindable variables may not be optimal for all problem domains. For instance a shared, global, outermost binding may be desirable, or alternatively, a thread may want to inherit current bindings from the parent thread at creation time, giving a “fork”-type model of state inheritance. These alternatives are not pursued in this module, but they might be an interesting area for future research.
8.3.2 Thread safety in client libraries

If an application uses multiple threads, then there may be thread safety requirements for any library that can be called simultaneously by multiple threads, even if the called library does not use the Threads library directly.

This section is about thread safety in any library that is designed to be used in a multi-threaded application.

General requirements

A library’s designer is responsible for documenting which features of the library offer built-in synchronization and which do not. While there is no definitive rule that can assist designers in this documentation, the following guidelines may be useful.

If a client of the library forgets to use a synchronization feature when one is necessary, the library designer should ensure that the effect of the lack of synchronization is limited to a small unit — probably a single object. In cases where the designer cannot guarantee that the effect will be limited, the library should either implement the synchronization internally, or provide a macro for clients to use instead.

Library implementors must ensure that the library provides implicit synchronization for any hidden global state which is maintained by the library. Library designers may choose whether the library should offer implicit synchronization of the state of objects managed by the library. The interface is more convenient if the synchronization is implicit, but it may be more efficient to rely on explicit synchronization by the client. Library designers should always document the choice they make.

Effects on the Dylan library

The definition of the Dylan library is not changed with the addition of the Threads module. The implementation ensures that all hidden global state (such as the symbol table and any generic function caches) is implicitly synchronized. Those functions in the Dylan library which are defined to modify the state of objects are not defined to provide implicit synchronization. However, implementations are expected to ensure that synchronization bugs in Dylan programs will not cause obscure errors that cannot be explained in terms of the semantics of Dylan language constructs.

The library guarantees that `element` and `element-setter` will be atomic for all of Dylan’s non-stretchy built-in collection classes, and for `<table>`, except for subclasses of `<string>`, and limited collections where the elements are constrained to be either of a type for which slots and variables do not guarantee atomicity (see Atomicity) or a subtype of `<character>`, or of a proper subtype of `<integer>`. This design is intended to permit implementations to use efficient representations for element values, which use either more or less space than a normal Dylan value. It is undefined whether any of the other standard Dylan functions are atomic. Where atomicity is not guaranteed, clients should guard against unexpected behavior by using explicit synchronization, as appropriate.

8.3.3 The Threads class hierarchy

- s - sealed | o - open
- p - primary | f - free
- c - concrete | a - abstract
- u - uninstantiable | i - instantiable

8.3.4 Basic features

This section documents basic features of the Threads module: operations on threads and low-level synchronization.
Fig. 1: Threads class hierarchy.
Low-level synchronization

**sequence-point Function**

Tells the compiler that it must consider the possibility of visible side effects from other threads at the point of the call.

**Signature**  sequence-point () => ()

**Discussion**

Tells the compiler that it must consider the possibility of visible side effects from other threads at the point of the call.

Normally, the compiler is not obliged to consider this possibility, and is free to rearrange program order provided that the reordering cannot be detected within a thread.

Calling this function effectively prohibits the compiler from rearranging the order of reads or writes from or to global data, relative to the call. This function may disallow compiler optimizations, leading to less efficient code — even for strongly ordered machines.

**synchronize-side-effects Function**

As *sequence-point*, with the addition that all side effects that have been performed within the calling thread are made visible within all other threads.

**Signature**  synchronize-side-effects () => ()

**Discussion**

A call to this function implies all the constraints to the compiler of a call to *sequence-point*. In addition it ensures that all side effects that have been performed within the calling thread are made visible within all other threads. Hence, no side effect performed after the call can be visible to other threads before side effects performed before the call. On a strongly ordered machine, this function might legitimately be performed as a null operation.

Some of the standard synchronization functions in the Threads module also ensure the visibility of side effects and act as sequence points, as if by a call to this function. This is defined to happen as follows:

- Immediately before a thread exits and becomes available for joining with *join-thread*
- Before *thread-yield* yields control
- After *wait-for* achieves synchronization (for all methods provided by the Threads module)
- Upon entry to *release* (for all methods provided by the Threads module)
- Upon entry to *release-all*

**Example** This example uses low-level synchronization to implement a class for performing lazy evaluation in a thread-safe manner, without the need for locks.

The class guarantees that the value will not be computed until it is needed, although it does not guarantee that it will not be computed more than once concurrently. This might be useful for memorization purposes.

The class uses 3 slots: one for a function which may be used to compute the value, one for a boolean indicating whether the value is already known, and one for the value itself, if known.

It is essential that no instance can ever be observed in a state where the boolean indicates a known value before the value is present. The low-level synchronization functions ensure this cannot happen.
Operations on threads

<thread> Instantiable Sealed Class

The class of threads.

Superclasses <object>

Init-Keywords

- function – An instance of <function>. Required.
- priority – A signed integer.
- name – An instance of <string>.

Discussion

The class representing a thread of control executing function.

The function is called with no arguments in the empty dynamic environment of the new thread. The thread terminates when the function returns.

The function is executable immediately. You can suspend a new thread (almost) immediately on creation by arranging for it to synchronize on an unavailable resource upon entry to the function.
The optional *priority* keyword provides a scheduling priority for the thread. The higher the value, the greater the priority. The default value is zero, which is also the value of the constant `$normal-priority`, one of several constants that correspond to useful priority levels. The module offers no way to change the priority of a thread dynamically.

The following constants, listed in order of increasing value, may be useful as values for the optional *priority* keyword.

- `$low-priority`
- `$background-priority`
- `$normal-priority`
- `$interactive-priority`
- `$high-priority`

The *name* keyword is a string that is used as the function’s name for convenience purposes, such as debugging.

**Operations**

The class `<thread>` provides the following operations:

- *thread-name* Returns the name of a thread, or `#f` if no name was supplied.
- *thread-id* Returns the ID of a thread.
- *join-thread* Blocks until one of the specified threads has terminated, and returns the values of its function.

**thread-name Function**

Returns the name of a thread.

**Signature**  thread-name thread => name-or-false

**Parameters**

- **thread** – An instance of `<thread>`.

**Values**

- **name-or-false** – An instance of `type-union(<string>, singleton(#f))`.

**Discussion**  Returns the name of *thread* as a string. If *thread* does not have a name, this function returns `#f`.

**thread-id Generic function**

Returns the thread ID of a thread.

**Signature**  thread-id thread => thread-id

**Parameters**

- **thread** – An instance of `<thread>`.

**Values**

- **thread-id** – An instance of `<integer>`.

**Discussion**  Returns the thread ID of a thread. This is similar to the process ID for an operating system process.

This is a value controlled by the underlying operating system. It is most useful when trying to correlate thread activity with reports from other tools.
**join-thread Function**

Waits for another, existing, thread to terminate, and then returns the values of its function.

**Signature**  
join-thread thread #rest threads => thread-joined #rest results

**Parameters**

- **thread** – An instance of `<thread>`. A thread to join.
- **threads (#rest)** – Instances of `<thread>`. More threads to join.

**Values**

- **thread-joined** – An instance of `<thread>`. The thread that was joined.
- **#rest results** – Zero or more instances of `<object>`. The values returned from the thread that was joined.

**Conditions**

An implementation of `join-thread` is permitted to signal the following condition:

**<duplicate-join-error>** A condition of this class (a subclass of `<error>`) may be signalled when a thread is passed to `join-thread`, if that thread has already been joined by an earlier call to `join-thread`, or if that thread is currently active in another call to `join-thread`.

**Discussion**

Waits for another, existing, thread to terminate, by blocking if necessary, and then returns the values of its function. The function returns the thread object that was joined, along with any values its function returns.

If more than one thread is passed to `join-thread`, the current thread blocks until the first of those threads terminates. The values returned are those of the first thread to terminate.

If one or more of the multiple threads has already terminated at the time of the call, then one of those terminated threads is joined. When more than one thread has already terminated, it is undefined which of those threads the implementation will join.

It is an error to pass a thread to `join-thread` if it has already been joined in a previous call to `join-thread`. It is an error to pass a thread to `join-thread` if that thread is also being processed by another simultaneous call to `join-thread` from another thread.

**thread-yield Function**

Force the current thread to yield control to the part of the implementation responsible for scheduling threads.

**Signature**  
thread-yield () => ()

**Discussion** Forces the current thread to yield control to the part of the implementation responsible for scheduling threads. Doing so may have the effect of allowing other threads to run, and may be essential to avoid deadlock in a co-operative scheduling environment.

**current-thread Function**

Returns the current thread.

**Signature**  
current-thread () => thread

**Values**

- **thread** – An instance of `<thread>.

**Discussion** Returns the current thread.

**current-thread-id Function**

Returns the ID of the current thread.
Signature current-thread-id () => thread-id

Values
- thread-id – An instance of <integer>.

Discussion Returns the ID of the current thread.

See also
- thread-id

### 8.3.5 Synchronization protocol

**Basic features**

<synchronization> Open Abstract Class

The class of objects that are used for inter-thread synchronization.

Superclasses <object>

Init-Keywords
- name – An instance of <string>.

Discussion

The class of objects that are used for inter-thread synchronization.

There is no explicit mechanism in the module to block on a number of synchronization objects simultaneously, until synchronization can be achieved with one of them. This mechanism can be implemented by creating a new thread to wait for each synchronization object, and arranging for each thread to release a notification once synchronization has been achieved.

The name keyword is a string that is used as the synchronization object’s name for convenience purposes, such as debugging.

Operations

The class <synchronization> provides the following operations:

- wait-for Block until synchronization can be achieved.
- release Release the object to make it available for synchronization.
- synchronization-name Returns the name of the synchronization object.

**wait-for** Open Generic function

Blocks until a synchronization object is available.

Signature wait-for object #key timeout => success

Parameters

- object – An instance of <synchronization>.
- timeout – Time-out interval. If the value is #f (the default), the time-out interval never elapses. Otherwise the value should be a <real>, corresponding to the desired interval in seconds.

Values

- success – An instance of <boolean>.
Discussion

Blocks until a synchronization object is available.

This function is the basic blocking primitive of the Threads module. It blocks until object is available and synchronization can be achieved, or the timeout interval has expired. A non-blocking synchronization may be attempted by specifying a timeout of zero. Individual methods may adjust the state of the synchronization object on synchronization. The function returns #t if synchronization is achieved before the timeout interval elapses; otherwise it returns #f.

**release** Open Generic function

Releases a synchronization object.

**Signature**

release object #key => ()

**Parameters**

- **object** – An instance of <synchronization>.

**Discussion** Releases the supplied synchronization object, object, potentially making it available to other threads. Individual methods describe what this means for each class of synchronization. This function does not block for any of the subclasses of <synchronization> provided by the module.

**synchronization-name** Open Generic function

Returns the name of a synchronization object.

**Signature**
synchronization-name object => name-or-false

**Parameters**

- **object** – An instance of <synchronization>.

**Values**

- **name-or-false** – An instance of type-union(<string>, singleton(#f)).

**Discussion** Returns the name of the synchronization object, object, if it was created with the name init-keyword. Otherwise #f is returned.

**Locks**

**<lock>** Open Abstract Instantiable Class

The class of locks.

**Superclasses** <synchronization>

**Discussion**

Locks are synchronization objects which change state when they are claimed (using wait-for), and revert state when released (using release).

It is normally necessary for programs to ensure that locks are released, otherwise there is the possibility of deadlock. Locks may be used to restrict the access of other threads to shared resources between the synchronization and the release. It is common for a protected operation to be performed by a body of code which is evaluated in a single thread between synchronization and release. A macro with-lock is provided for this purpose. When a thread uses a lock for mutual-exclusion in this way, the thread is said to own the lock.

<lock> has no direct instances; calling make on <lock> returns an instance of <simple-lock>. 

Operations

The class `<lock>` provides the following operations:

- **with-lock** Execute a body of code between `wait-for` and `release` operations.

**with-lock Statement Macro**

Holds a lock while executing a body of code.

**Macro Call**

```dylan
with-lock (*lock*, #key *keys*)
   *body*
   [failure *failure-expr*]
end
```

**Parameters**

- **lock** – An instance of `<lock>`.
- **keys** – Zero or more of the keywords provided by `wait-for`.
- **body** – A body of Dylan code.

**Values**

- **values** – Zero or more instances of `<object>`.

**Conditions** `with-lock` may signal a condition of the following class (a subclass of `<serious-condition>`):

- `<timeout-expired>` This is signalled when `with-lock` did not succeed in claiming the lock within the timeout period.

**Discussion** Execute the `body` with `lock` held. If a `failure` clause is supplied, then it will be evaluated and its values returned from `with-lock` if the lock cannot be claimed (because a time-out occurred). The default, if no `failure` clause is supplied, is to signal an exception of class `<timeout-expired>`. If there is no failure, `with-lock` returns the results of evaluating the body.

**Example** If no `failure` clause is supplied, the macro expands into code equivalent to the following:

```dylan
let the-lock = *lock*;
if (wait-for(the-lock, *keys ...*))
   block ()
   *body*...
   cleanup
   release(the-lock)
end block
else
   signal(make(<timeout-expired>,
                synchronization: the-lock)
end if
```

Semaphores

**<semaphore>** Open Primary Instantiable Class

The class of traditional counting semaphores.

**Superclasses** `<lock>`

**Init-Keywords**

8.3. The threads Module
• **initial-count** – A non-negative integer, corresponding to the initial state of the internal counter. The default value is 0.

• **maximum-count** – A non-negative integer corresponding to the maximum permitted value of the internal counter. The default value is the largest value supported by the implementation, which is the value of the constant $semaphore-maximum-count-limit$. This constant will not be smaller than 10000.

**Discussion**

The `<semaphore>` class is a class representing a traditional counting semaphore. An instance of `<semaphore>` contains a counter in its internal state. Calling `release` on a semaphore increments the internal count. Calling `wait-for` on a semaphore decrements the internal count, unless it is zero, in which case the thread blocks until another thread releases the semaphore.

Semaphores are less efficient than exclusive locks, but they have asynchronous properties which may be useful (for example for managing queues or pools of shared resources). Semaphores may be released by any thread, so there is no built-in concept of a thread owning a semaphore. It is not necessary for a thread to release a semaphore after waiting for it — although semaphores may be used as locks if they do.

**wait-for(<semaphore>)** Sealed Method

Claims a semaphore object.

**Signature**  
wait-for object #key timeout => success

**Parameters**

• **object** – An instance of `<semaphore>`. The semaphore object to wait for.

• **timeout (#key)** – Time-out interval. If the value is #f (the default), the time-out interval never elapses. Otherwise the value should be a `<real>`, corresponding to the desired interval in seconds.

**Values**

• **success** – An instance of `<boolean>`.  

**Discussion**  
Decrements the internal count of the semaphore object, blocking if the count is zero.

See also

• `wait-for`.

**release(<semaphore>)** Sealed Method

Releases a semaphore object.

**Signature**  
release object #key => ()

**Parameters**

• **object** – An instance of `<semaphore>`.  

**Conditions**

An implementation of this `release` method is permitted to signal a condition of the following class, which is a subclass of `<error>`:

`<count-exceeded-error>` **This may be signalled when an attempt is made to release a `<semaphore>` when the internal counter is already at its maximum count.**

**Discussion**  
Releases a semaphore object, by incrementing its internal count.

See also

• `release`. 
**Exclusive locks**

*<exclusive-lock> Open Abstract Instantiable Class*

The class of locks which prohibit unlocking by threads that do not own the lock.

**Superclasses** *<lock>*

**Discussion**

The class of locks which prohibit unlocking by threads that do not own the lock.

The notion of ownership is directly supported by the class, and a thread can test whether an `<exclusive-lock>` is currently owned. An instance of `<exclusive-lock>` can only be owned by one thread at a time, by calling `wait-for` on the lock.

Once owned, any attempt by any other thread to wait for the lock will cause that thread to block. It is an error for a thread to release an `<exclusive-lock>` if another thread owns it.

`<exclusive-lock>` has no direct instances; calling `make` on `<exclusive-lock>` returns an instance of `<simple-lock>`.

**Operations**

The class `<exclusive-lock>` provides the following operations:

- **owned?** Tests to see if the lock has been claimed by the current thread.

**release(<exclusive-lock>) Sealed Method**

Releases an exclusive lock.

**Signature** release object #key => ()

**Parameters**

- **object** – An instance of `<exclusive-lock>`.

**Conditions**

Implementations of `release` methods for subclasses of `<exclusive-lock>` are permitted to signal a condition of the following class, which is a subclass of `<error>`:

- `<not-owned-error>` This may be signalled when an attempt is made to release an `<exclusive-lock>` when the lock is not owned by the current thread.

**Discussion**

Releases a lock that is owned by the calling thread. It is an error if the lock is not owned.

The Threads module does not provide a method on `release` for `<exclusive-lock>`, which is an open abstract class. Each concrete subclass will have an applicable method which may signal errors according to the protocol described above.

**owned? Open Generic function**

Tests whether an exclusive lock has been claimed by the current thread.

**Signature** owned? object => owned?

**Parameters**

- **object** – An instance of `<exclusive-lock>`.

**Values**

- **owned?** – An instance of `<boolean>`.

**Discussion** Tests whether the exclusive lock has been claimed by the current thread.
Recursive locks

<recursive-lock> Open Primary Instantiable Class
The class of locks that can be locked recursively.

Superclasses <exclusive-lock>

Discussion A thread can lock a <recursive-lock> multiple times, recursively, but the lock
must later be released the same number of times. The lock will be freed on the last of these
releases.

wait-for(<recursive-lock>) Sealed Method

Summary Claims a recursive lock.

Signature wait-for object #key timeout => success

Parameters

• object – An instance of <recursive-lock>.
• timeout (#key) – Time-out interval. If the value is #f (the default), the time-out interval
  never elapses. Otherwise the value should be a <real>, corresponding to the desired
  interval in seconds.

Values

• success – An instance of <boolean>.

Discussion Claims a recursive lock, blocking if it is owned by another thread.

See also

• wait-for.

release(<recursive-lock>) Sealed Method

Releases a recursive lock.

Signature release object #key => ()

Parameters

• object – An instance of <recursive-lock>.

Discussion Releases a recursive lock, and makes it available if it has been released as many times as
it was claimed with wait-for.

owned?(<recursive-lock>) Sealed Method

Tests whether a recursive lock has been claimed by the current thread.

Signature owned? object => owned?

Parameters

• object – An instance of <recursive-lock>.

Values

• owned? – An instance of <boolean>.

Discussion Tests whether a recursive lock has been claimed by the current thread.
Simple locks

<simple-lock> Open Primary Instantiable Class
A simple and efficient lock.

Superclasses <exclusive-lock>

Discussion The <simple-lock> class represents the most simple and efficient mutual exclusion synchronization primitive. It is an error to lock a <simple-lock> recursively. An attempt to do so might result in an error being signalled, or deadlock occurring.

wait-for(<simple-lock>) Sealed Method
Claims a simple lock.

Signature  wait-for object #key timeout => success

Parameters

• object – An instance of <simple-lock>.
• timeout (#key) – Time-out interval. If the value is #f (the default), the time-out interval never elapses. Otherwise the value should be a <real>, corresponding to the desired interval in seconds.

Values

• success – An instance of <boolean>.

Discussion Claims a simple lock, blocking if it is owned by another thread.

See also

• wait-for.

release(<simple-lock>) Sealed Method
Releases a simple lock.

Signature  release object #key => ()

Parameters

• object – An instance of <simple-lock>.

Discussion Releases a simple lock.

See also

• release.

owned?(<simple-lock>) Sealed Method
Tests whether a simple lock has been claimed by the current thread.

Signature  owned? object => owned?

Parameters

• object – An instance of <simple-lock>.

Values

• owned? – An instance of <boolean>.

Discussion Tests whether a simple lock has been claimed by the current thread.
Multiple reader / single writer locks

<read-write-lock> Open Primary Instantiable Class
The class of locks that can have multiple readers but only one writer.

Superclasses <exclusive-lock>

Discussion
The class of locks that can have multiple readers but only one writer.

The <read-write-lock> class can be locked in either of two modes, read and write. A write lock is exclusive, and implies ownership of the lock. However, a read lock is non-exclusive, and an instance can be locked multiple times in read mode, whether by multiple threads, recursively by a single thread, or a combination of both.

A <read-write-lock> can only be locked in write mode if the lock is free, and the operation will block if necessary. It can only be freed by the thread that owns it.

A <read-write-lock> can be locked in read mode provided that it is not owned with a write lock. The operation will block while the lock is owned. Each time it is locked in read mode, an internal counter is incremented. This counter is decremented each time a read-mode lock is released. The lock is freed when the counter becomes zero.

The <read-write-lock> class is less efficient than the other lock classes defined in the Threads module. However, it provides an efficient and convenient means to protect data that is frequently read and may occasionally be written by multiple concurrent threads.

wait-for(<read-write-lock>) Sealed Method
Claims a read-write lock.

Signature wait-for object #key timeout mode

Parameters
• object – An instance of <read-write-lock>.
• timeout (#key) – Time-out interval. If the value is #f (the default), the time-out interval never elapses. Otherwise the value should be a <real>, corresponding to the desired interval in seconds.
• mode (#key) – The mode of the lock to wait for. Valid values are #"read" (the default) and #"write", which wait for locks in read mode and write mode respectively.

Values
• success – An instance of <boolean>.

Discussion
Claims a read-write lock, blocking if necessary. The behavior depends on the value of mode:
• #"read" If there is a write lock, blocks until the lock becomes free. Then claims the lock by incrementing its internal read-lock counter.
• #"write" First waits until the lock becomes free, by blocking if necessary. Then claims exclusive ownership of the lock in write mode.

If the claim is successful, this method returns true; otherwise it returns false.

release(<read-write-lock>) Sealed Method
Releases a read-write-lock.

Signature release object #key => ()
Parameters

• **object** – An instance of `<read-write-lock>`.

Discussion

Releases a read-write lock.

If the lock is owned by the calling thread, it is freed. If the lock is locked in read mode, the count of the number of locks held is decremented; the lock is freed if the count becomes zero. Otherwise it is an error to release the lock, and an implementation is permitted to signal a `<not-owned-error>` condition.

**owned? (<read-write-lock>)** Sealed Method

Tests whether a read-write lock is owned — that is, has been locked in write mode — by the current thread.

**Signature**  
`owned? object => owned?`

**Parameters**

• **object** – An instance of `<read-write-lock>`.

**Values**

• **owned?** – An instance of `<boolean>`.

**Discussion** Tests whether a read-write lock is owned — that is, has been locked in write mode — by the current thread.

Notifications

**<notification>** Instantiable Sealed Class

The class of objects that can be used to notify threads of a change of state elsewhere in the program.

**Superclasses** `<synchronization>`

**Init-Keywords**

• **lock** – An instance of `<simple-lock>`. Required.

**Discussion**

The class of objects that can be used to notify threads of a change of state elsewhere in the program. Notifications are used in association with locks, and are sometimes called *condition variables*. They may be used to support the sharing of data between threads using *monitors*. Each `<notification>` is permanently associated with a `<simple-lock>`, although the same lock may be associated with many notifications.

The required *lock* is associated with the notification, and it is only possible to wait for, or release, the notification if the lock is owned.

Threads wait for the change of state to be notified by calling `wait-for`. Threads notify other threads of the change of state by calling `release`.

**Operations**

The class `<notification>` provides the following operations:

• **associated-lock** Returns the lock associated with the notification object.

• **wait-for** Wait for the notification of the change in state. The associated lock must be owned, and is atomically released before synchronization, and reclaimed after.

• **release** Notify the change of state to a single waiting thread. This has no effect on the associated lock, which must be owned.
• release-all Notify the change of state to all waiting threads. This has no effect on the associated lock, which must be owned.

Example This example shows how to use a notification and an associated lock to implement a queue. The variable *queue* is the actual queue object (a <deque>). Queue access is performed by interlocking pushes and pops on the <deque>. The *queue* variable can be a constant, since it is the <deque> which is mutated and not the value of *queue*.

```dylan
define constant *queue* = make(<deque>);
```

The variable *lock* is used to isolate access to the queue

```dylan
define constant *lock* = make(<lock>);
```

The variable *something-queued* is a notification which is used to notify other threads that an object is being put onto an empty queue.

```dylan
define constant *something-queued* = make(<notification>, lock: *lock*);
```

The function put-on-queue pushes an object onto the queue. If the queue was initially empty, then all threads which are waiting for the queue to fill are notified that there is a new entry.

```dylan
define method put-on-queue (object) => ()
  with-lock (*lock*)
  if (*queue*.empty?)
    release-all(*something-queued*)
  end;
  push(*queue*, object)
  end with-lock
end method;
```

The get-from-queue function returns an object from the queue. If no object is immediately available, then it blocks until it receives a notification that the queue is no longer empty. After receiving the notification it tests again to see if an object is present, in case it was popped by another thread.

```dylan
define method get-from-queue () => (object)
  with-lock (*lock*)
  while (*queue*.empty?)
    wait-for(*something-queued*)
  end;
  pop(*queue*)
  end with-lock
end method;
```

associated-lock Function

Returns the lock associated with the notification object supplied.

Signature associated-lock notification => lock

Parameters

• notification – An instance of <notification>.

Values

• lock – An instance of <simple-lock>.

Discussion Returns the lock associated with the notification object notification.
**wait-for(<notification>)** Sealed Method

Wait for another thread to release a notification.

**Signature**

\[
\text{wait-for } \text{notification } \#\text{key } \text{timeout} \Rightarrow \text{success}
\]

**Parameters**

- **notification** – An instance of `<notification>.
- **timeout** (`#key`) – Time-out interval. If the value is `#f` (the default), the time-out interval never elapses. Otherwise the value should be a `<real>`, corresponding to the desired interval in seconds.

**Values**

- **success** – An instance of `<boolean>.

**Discussion**

Wait for another thread to release *notification*. The lock associated with the notification must be owned. Atomically, the lock is released and the current thread starts blocking, waiting for another thread to release the notification. The current thread reclaims the lock once it has received the notification.

Note that the state should be tested again once *wait-for* has returned, because there may have been a delay between the *release* of the notification and the claiming of the lock, and the state may have been changed during that time. If a timeout is supplied, then this is used for waiting for the release of the notification only. The *wait-for* function always waits for the lock with no timeout, and it is guaranteed that the lock will be owned on return. The *wait-for* function returns `#f` if the notification wait times out.

**Conditions**

Implementations of this *wait-for* method are permitted to signal a condition of the following class, which is a subclass of `<error>`:

- **<not-owned-error>** Implementations can signal this error if the application attempts to wait for a notification when the associated lock is not owned by the current thread.

**release(<notification>)** Sealed Method

Releases a notification to one of the threads that are blocked and waiting for it.

**Signature**

\[
\text{release } \text{notification } \#\text{key} \Rightarrow ()
\]

**Parameters**

- **notification** – An instance of `<notification>.

**Conditions**

Implementations of this *release* method are permitted to signal a condition of the following class, which is a subclass of `<error>`:

- **<not-owned-error>** Implementations can signal this error if the application attempts to release a notification when the associated lock is not owned by the current thread.

**Discussion**

Releases *notification*, announcing the change of state to one of the threads which are blocked and waiting for it. The choice of which thread receives the notification is undefined. The receiving thread may not be unblocked immediately, because it must first claim ownership of the notification’s associated lock.

**release-all Function**

Release a notification to all the threads that are blocked and waiting for it.

**Signature**

\[
\text{release-all } \text{notification} \Rightarrow ()
\]
Parameters

- **notification** – An instance of `<notification>`.

Conditions

Implementations of the `release-all` function are permitted to signal a condition of the following class, which is a subclass of `<error>`:

- `<not-owned-error>` This may be signalled when an attempt is made to release a notification when the associated lock is not owned by the current thread.

Discussion Releases notification, announcing the change of state to all threads which are blocked and waiting for it. Those threads will then necessarily have to compete for the lock associated with the notification.

### 8.3.6 Timers

**sleep Function**

Blocks the current thread for a specified number of seconds.

**Signature** `sleep interval => ()`

**Parameters**

- **interval** – An instance of `<real>`.

**Discussion** Blocks the current thread for the number of seconds specified in `interval`.

### 8.3.7 Thread variables

**threadVariable definition adjective Macro**

An adjective to `define variable` for defining thread variables.

**Macro Call**

```
define thread variable *bindings* = *init* ;
```

**Discussion** An adjective to `define variable`. The construct `define thread variable` defines module variables in the current module which have thread-local bindings. The initialization expression is evaluated once, and is used to provide the initial values for the variables in each thread. The value of a thread variable binding may be changed with the normal assignment operator `:=`. This assignment is not visible in other threads.

**Example**

```
define thread variable *standard-output*
  = make(<standard-output-stream>);
```

### 8.3.8 Dynamic binding

**dynamic-bind Statement Macro**

Executes a body of code in a context in which variables are dynamically rebound.

**Macro Call**
**Dynamic Bind**

`dynamic-bind (*place1* = *init1*, *place2* = *init2*, ...)`  
`*body*`  
`end;`

**Discussion**

Executes `body` with the specified `places` rebound in the dynamic environment, each place being initialized to the results of evaluating the initialization expressions. In other words, the places are initialized to new values on entry to the body but restored to their old values once the body has finished executing, whether because it finishes normally, or because of a non-local transfer of control. Typically, each `place` is a thread variable.

If the `place` is a `name`, it must be the name of a thread variable in the module scope.

**Example**  The following example shows the dynamic binding of a single variable.

```
dynamic-bind (*standard-output* = new-val())  
top-level-loop ()  
end;```

This expands into code equivalent to the following:

```
begin  
let old-value = *standard-output*;  
block ()  
  *standard-output* := new-val();  
  top-level-loop()  
cleanup  
  *standard-output* := old-value  
end
end```

**An extended form of dynamic-bind**

Some implementations of the Threads module may provide an extended form of `dynamic-bind` for binding places other than variables. The implementation of this extended form requires the use of non-standard features in the Dylan macro system, and hence cannot be written as a portable macro. These non-standard extensions are subject to discussion amongst the Dylan language designers, and may eventually become standard features. Until such time as standardization occurs, implementations are not mandated to implement the extended form of `dynamic-bind`, and portable code should not depend upon this feature.

The extended form is described below.

**Dynamic-bind (extended) Statement Macro**

Executes a body of code in a context in which variables or other places are dynamically rebound.

**Macro Call**

```
dynamic-bind (*place1* = *init1*, *place2* = *init2*, ...)  
*body*  
end;```

(This is the same as the simple form.)

**Discussion**

If `place` is not a `name`, then it may have the syntax of a call to a function. This permits an extended form for `dynamic-bind`, by analogy with the extended form for `:=`. In this case, if
the place appears syntactically as `name(arg1, ... argn)`, then the macro expands into a call to the function

\[
\text{name-dynamic-binder(*init*, *body-method*, *arg1*, ... *argn*)}
\]

where `init` is the initial value for the binding, and `body-method` is function with no parameters whose body is the body of the `dynamic-bind`. The extended form also permits the other `. and [] syntaxes for function calls.

There are no features in the current version of the Threads module which make use of the extended form of `dynamic-bind`.

**Example** The following example shows the extended form of `dynamic-bind`.

```dylan
dynamic-bind (object.a-slot = new-slot-val())
    inner-body(object)
end;
```

This expands into code equivalent to the following:

```dylan
a-slot-dynamic-binder(new-slot-val(),
    method () inner-body(object) end, object)
```

### 8.3.9 Locked variables

**locked Variable definition adjective Macro**

Defines a locked variable.

**Macro Call**

```dylan
define locked variable *bindings* = *init* ;
```

**Discussion**

An adjective to `define variable`. The construct `define locked variable` defines module variables in the current module that can be tested and updated with `conditional-update!`, `atomic-increment!`, or `atomic-decrement!`.

Other threads are prevented from modifying the locked variable during the conditional update operation by means of a low-level locking mechanism, which is expected to be extremely efficient.

**Operations**

- `conditional-update!` Atomically compare and conditionally assign to the variable.
- `atomic-increment!` Atomically increment the variable.
- `atomic-decrement!` Atomically decrement the variable.

**Example**

```dylan
define locked variable *number-detected* = 0;
```
8.3.10 Conditional update

**conditional-update! Statement Macro**

Performs an atomic test-and-set operation.

**Macro Call**

```dylan
conditional-update!(\*local-name* = \*place*)
  \*body*
  [success \*success-expr* ]
  [failure \*failure-expr* ]
end
```

**Parameters**

- **local-name** – A Dylan variable-name.*bnf*.
- **place** – A Dylan variable-namebnf. If the implementation provides the extended form of `conditional-update!`, `place` can also be a function call.
- **body** – A Dylan body bnf.

**Values**

- **value** – See description.

**Discussion**

Performs an atomic test-and-set operation. Where appropriate, it should be implemented using dedicated processor instructions, and is expected to be extremely efficient on most platforms.

The value of the `place` is evaluated once to determine the initial value, which is then bound to the `local-name` as a lexical variable. The `body` is then evaluated to determine the new value for the place. The place is then conditionally updated — which means that the following steps are performed atomically:

1. The place is evaluated again, and a test is made to see if it has been updated since the initial evaluation. This may involve a comparison with the old value using `\==`, though implementations might use a more direct test for there having been an assignment to the place. It is undefined whether the test will succeed or fail in the case where the place was updated with a value that is identical to the old value when compared using \\=`=.`

2. If the value was found not to have been updated since the initial evaluation, the new value is stored by assignment. Otherwise the conditional update fails.

If the update was successful, then `conditional-update!` returns the result of the `success` expression, or returns the new value of the place if no `success` clause was supplied.

If the update failed, then `conditional-update!` signals a condition, unless a `failure` clause was given, in which case the value is returned.

If the `place` is a `name`, it must be the name of a `locked variable` in the current module scope. See `Locked variables`.

**Conditions** `conditional-update!` may signal a condition of the following class (which is a subclass of `<error>`), unless a `failure` clause is supplied.

- `<conditional-update-error>`

**Example** The following example does an atomic increment of `\*number-detected*`. 

8.3. The threads Module 231
Until (conditional-update!
  (current-val = *number-detected*)
  current-val + 1
  failure #f
  end conditional-update!)
end until

**atomic-increment! Function Macro**

Atomically increments a place containing a numeric value.

**Macro Call**

```
atomic-increment!(*place*);
atomic-increment!(*place*, *by*);
```

**Parameters**

- `place` – A Dylan variable-name. If the implementation provides the extended form of `conditional-update!`, `place` can also be a function call.

**Values**

- `new-value` – An instance of `<object>`.

**Discussion**

Atomically increments a place containing a numeric value.

The value of the `place` is evaluated one or more times to determine the initial value. A new value is computed from this value and `by`, by applying `+` from the Dylan module. The new value is atomically stored back into `place`.

The macro returns the new value of `place`.

The `place` must be a suitable place for `conditional-update!`.

Implementations of `atomic-increment!` are permitted to use `conditional-update!` (as in the described example), and hence can involve a loop and can cause `place` to be evaluated more than once. However, an atomic increment of a locked variable might be implemented by a more efficient non-looping mechanism on some platforms.

**Example**
The following example atomically increments `*number-detected*` by 2, and returns the incremented value.

```
atomic-increment!(*number-detected*, 2);
```

**atomic-decrement! Function Macro**

Atomically decrements a place containing a numeric value.

**Macro Call**

```
atomic-decrement!(*place*);
atomic-decrement!(*place*, *by*);
```

**Parameters**

- `place` – A Dylan variable-name. If the implementation provides the extended form of `conditional-update!`, `place` can also be a function call.
• **by** – An instance of `<object>`. Default value: 1.

**Values**

• **new-value** – An instance of `<object>`.

**Discussion** Atomically decrements a place containing a numeric value. It has the same semantics as `atomic-increment!` with the exception that the `place` is decremented.

### An extended form of `conditional-update!`

Some implementations of the Threads module may provide an extended form of `conditional-update!` for updating places other than locked variables. The implementation of this extended form requires the use of non-standard features in the Dylan macro system, and hence cannot be written as a portable macro. These non-standard extensions are subject to discussion amongst the Dylan language designers, and may eventually become features. Until such time as standardization occurs, implementations are not mandated to implement the extended form of `conditional-update!`, and portable code should not depend upon the feature.

**conditional-update! (extended) Statement Macro**

Performs an atomic test-and-set operation.

**Macro Call**

```dylan
conditional-update!(*local-name* = *place*)
 *body*
  [success *success-expr* ]
  [failure *failure-expr* ]
end
```

**Parameters**

• **local-name** – A Dylan variable-name `bnf`.

• **place** – A Dylan variable-name `bnf` or a function call.

• **body** – A Dylan body `bnf`.

**Discussion**

This extended form of `conditional-update!` additionally accepts a `place` that has the syntax of a call to a function. This extended form for `conditional-update!` is analogous to that for `:=`. In this case, if the `place` appears syntactically as

```dylan
*name* (*arg* 1, ... *arg* n)
```

The macro expands into this call:

```dylan
*name* -conditional-updater(*new-value*, *local-name*, *arg* 1, ... →*arg* n)
```

If the result of this function call is `#f`, the conditional update is deemed to have failed.
The hash-algorithms library provides consistent access to a variety of hash algorithms:

- md5
- sha1
- sha224
- sha256
- sha384
- sha512

Hashing an entire string at once can be done via the functions named after each hash:

```javascript
let digest = sha1("Some text");
```

If you want a printable digest, use `hexdigest (<byte-vector>)`:

```javascript
let hexdigest = hexdigest(sha1("Some text"));
```

If you want to hash multiple strings into a single digest (useful when streaming), you can use the `update-hash` and `digest` functions:

```javascript
let hash = make(<sha1>);
update-hash(hash, "Some");
update-hash(hash, " ");
update-hash(hash, "text");
let digest = digest(hash);
// hexdigest works on hashes as well:
let hexdigest = hexdigest(hash);
```

### 9.1 The hash-algorithms Module

**<hash> Class**

**Superclasses**  `<object>`

**digest-size Generic function**

Returns the digest size of the hash algorithm.

**Signature**  `digest-size (hash) => (digest-size)`

**Parameters**
hash – An instance of `<hash>`.

Values

digest-size – An instance of `<integer>`.

**block-size**

Generic function

Returns the block size of the hash algorithm.

**Signature**

`block-size (hash) => (block-size)`

**Parameters**

• **hash** – An instance of `<hash>`.

Values

• **block-size** – An instance of `<integer>`.

**update-hash**

Generic function

Add more data to the hash.

**Signature**

`update-hash (hash, input) => ()`

**Parameters**

• **hash** – An instance of `<hash>`.

• **input** – An instance of `<byte-string>, <buffer> or <byte-vector>`.

**Discussion**

Add more data to the hash. This is useful when streaming data or the data is available in multiple strings and you wish to avoid the overhead of concatenation.

Calling `update-hash` multiple times is equivalent to calling it once with a concatenation of the arguments:

```
let hash-separate = make(<sha1>);
update-hash(hash-separate, "Some");
update-hash(hash-separate, " ");
update-hash(hash-separate, "text");
let digest-separate = digest(hash-separate);

let hash-combined = make(<sha1>);
update-hash(hash-combined, "Some text");
let digest-combined = digest(hash-combined);
// digest-separate and digest-combined will be the same
```

See also

• digest

• `hexdigest(<hash>)`

• `hexdigest(<byte-vector>)`

**digest**

Generic function

**Signature**

`digest (hash) => (digest)`

**Parameters**

• **hash** – An instance of `<hash>`.

Values
• `digest` – An instance of `<byte-vector>`.

Discussion

The return value `digest` is binary data and may include null bytes. To display this result in text form, use `hexdigest(<hash>)` or `hexdigest(<byte-vector>)`.

Use `update-hash` to add data to the hash.

See also

• `update-hash`
• `hexdigest(<hash>)`
• `hexdigest(<byte-vector>)`

hexdigest(<hash>) Method

Returns the digest for the given hash as a hexadecimal string.

Signature `hexdigest (hash) => (hexdigest)`

Parameters

• `hash` – An instance of `<hash>`.

Values

• `hexdigest` – An instance of `<byte-string>`.

See also

• `digest`
• `hexdigest(<byte-vector>)`

hexdigest(<byte-vector>) Method

Returns the digest given as a hexadecimal string.

Signature `hexdigest (digest) => (hexdigest)`

Parameters

• `digest` – An instance of `<byte-vector>`.

Values

• `hexdigest` – An instance of `<byte-string>`.

See also

• `digest`
• `hexdigest(<hash>)`

9.1.1 MD5

<md5> Class

Superclasses `<hash>`

md5 Function

Signature `md5 (input) => (digest)`

Parameters

• `input` – An instance of `<byte-string>, <buffer>` or `<byte-vector>`.
9.1.2 SHA-1

<sha1> Class

Superclasses <hash>

sha1 Function

Signature sha1 (input) => (digest)

Parameters

• input – An instance of <byte-string>, <buffer> or <byte-vector>.

Values

• digest – An instance of <byte-vector>.

9.1.3 SHA-2

<sha256> Class

Superclasses <hash>

sha256 Function

Signature sha256 (input) => (digest)

Parameters

• input – An instance of <byte-string>, <buffer> or <byte-vector>.

Values

• digest – An instance of <byte-vector>.

<sha224> Class

Superclasses <hash>

sha224 Function

Signature sha224 (input) => (digest)

Parameters

• input – An instance of <byte-string>, <buffer> or <byte-vector>.

Values

• digest – An instance of <byte-vector>.

<sha384> Class

Superclasses <hash>

sha384 Function

Signature sha384 (input) => (digest)

Parameters

• input – An instance of <byte-string>, <buffer> or <byte-vector>. 

Values

• digest – An instance of <byte-vector>.
Values

- **digest** – An instance of `<byte-vector>`.

**<sha512> Class**

**Superclasses** `<hash>`

**sha512 Function**

**Signature** `sha512 (input) => (digest)`

**Parameters**

- **input** – An instance of `<byte-string>`, `<buffer>` or `<byte-vector>`.

**Values**

- **digest** – An instance of `<byte-vector>`.
10.1 The format Module

The Format module is exported from the IO library. This module extends the functionality of the format strings described in Dylan's condition system and provides two new functions for processing the extended format strings. The Format module is a small module, but it uses the printing modules and some of the Streams module. The print and pprint Modules and The streams Module give full details of the Print and Streams libraries.

The format module exports all the identifiers described in this document.

10.1.1 Control strings

The Format module's format strings, or control strings, offer the same directives as Dylan's format strings offer, but Format provides a few more directives, and permits a single argument to all format directives.

The argument is an integer that must appear contiguously between the dispatch character, %, and the format directive. The argument indicates a printing field in which to justify the output of the directive. A positive integer indicates that the output should be flush right within the field, and a negative integer indicates the output should be flush left within the field. If the output length is greater than the field's width, then output occurs as if there were no field specification. The following are examples of valid format directives:

\%s
\%S
\%15d
\%-10=

The directives are:

- \%s Prints the next format argument as a message by calling the function print-message on the format argument and the stream. This directive is the same as Dylan’s %s format-string directive except for two features: (i) this module's %s directive outputs character objects, and (ii) you can extend the %s functionality by adding methods to print-message.
- \%= Prints the next format argument by calling the print function from the Print module on the format argument and the stream. You can extend the %= functionality by adding methods to the print-object function from the Print module.
- \%c Print the next format argument, which must be a character, according to Dylan’s %s format-string directive. This module’s %c directive is the same as this module’s %s directive.
- \%d Prints a decimal representation of the next format argument, which must be an integer.
- \%b Prints a binary representation of the next format argument, which must be an integer.
- \%o Prints an octal representation of the next format argument, which must be an integer.
• %x Prints a hexadecimal representation of the next format argument, which must be an integer.
• %m Invokes the next format argument, which must be a function, on the stream passed to format.
• %% Outputs a single % character.

10.1.2 The format Module

This section contains a reference entry for each item exported from the Format module.

format Generic function
Outputs a control string to a stream.

Signature
format stream control-string #rest arguments => ()

Parameters
• stream – An instance of <stream>. The stream to which formatted output should be sent.
• control-string – An instance of <string>. A string containing format directives.
• arguments (#rest) – Instances of <object>.

Discussion
Sends output to stream according to the format directives in control-string. Each directive consumes one argument from arguments. See Control strings for a description of the control strings that can be used.

The control-string contents that are not part of any directive are output directly to stream, as if by the Streams module’s write function.

format(<byte-string>) Method
Outputs a control string to a stream.

Parameters
• stream – An instance of <stream>.
• control-string – An instance of <byte-string>.
• arguments (#rest) – Instances of <object>.

Discussion
There is one method for format, and it is specialized to <byte-string>.

format-to-string Generic function
Returns a formatted string based on a format control string.

Signature
format-to-string control-string #rest arguments => result

Parameters
• control-string – An instance of <string>.
• arguments (#rest) – Instances of <object>.

Values
• result – An instance of <string>.

Discussion
Calls format to produce output according to control-string and returns the output as a string.

format-to-string(<byte-string>) Method
Returns a formatted string based on a format control string.
Parameters

- control-string – An instance of <byte-string>.
- arguments (#rest) – Instances of <object>.

Values

- result – An instance of <byte-string>.

Discussion

There is one method for format-to-string. The control-string argument must be a <byte-string>. Result is a <byte-string>.

**print-message Generic function**

Prints an object to a stream.

Parameters

- object – An instance of <object>.
- stream – An instance of <stream>.

Discussion

Prints object to stream.

Methods for this function should print objects as a message, as opposed to printing them in any form intending to represent Dylan data, literal syntax, and so on.

For example, printing a condition object with this function presents the condition as an error message, but printing the condition object with the print function from the Print module prints the condition in some form such as:

```
{Simple-error}
```

See the individual methods for the details of how this function prints various objects. This function exists to define the behavior of the %s format directive and to allow users the ability to extend the %s directive. Users should have little need to call this function directly.

**print-message(<condition>) Sealed Method**

Prints a condition to a stream as an error message.

Parameters

- condition – An instance of <condition>.
- stream – An instance of <stream>.

Discussion

Prints condition as an error message, as described for the Dylan %s format directive. You should not specialize the print-message protocol for subclasses of <condition>, but instead extend the print-message protocol to new condition objects by specializing methods on report-condition.

Note: This doesn’t actually work. Fix.

**print-message(<symbol>) Sealed Method**

Prints a symbol to a stream.

Signature

print-message symbol stream => ()

Parameters
• **symbol** – An instance of `<symbol>`.
• **stream** – An instance of `<stream>`.

**Discussion**

Prints `symbol` to `stream` by converting it to a string with the `as` function and then writing the string with the `write` function from the Streams module.

**print-message(<string> or <character>)** Sealed Method

Prints an object to a stream.

**Signature**  
`print-message object stream => ()`

**Parameters**

- **object** – An instance of `type-union(<string>, <character>)`.
- **stream** – An instance of `<stream>`.

**Discussion**  
Prints `object` to `stream` by calling the `write` function from the Streams module.

### 10.2 The format-out Module

The Format-Out module is a convenient repackaging of two libraries that provides a simple way to send text to the platform’s standard output stream. For this purpose, Format-Out uses the Format module and the Standard-IO module and defines the functions `format-out`, `force-out`, `format-err`, and `force-err`. The Format-Out module exports all the identifiers described in this document.

*The format Module* and *The standard-io Module* give full details of the Format and Standard-IO libraries.

#### 10.2.1 The format-out Module

This section contains a reference entry for each item exported from the `format-out` module.

**format-out Generic function**

Formats its arguments on the standard output.

**Signature**  
`format-out control-string #rest arguments => ()`

**Parameters**

- **control-string** – An instance of `<string>`.
- **arguments (#rest)** – Instances of `<object>`.

**Discussion**

Calls the `format` function from the `format` module on `*standard-output*` from the `standard-io` module, `control-string`, and `arguments`.

This function is thread-safe.

**See also**

- `format`
- `*standard-output*`

**format-out(<byte-string>)** Sealed Method

Formats its arguments on the standard output.

**Signature**  
`format-out control-string #rest arguments => ()`
Parameters

- **control-string** – An instance of `<byte-string>`.
- **arguments (#rest)** – Instances of `<object>`.

Discussion

Formats its arguments on the standard output. There is one method for `format-out`, and it is specialized to instances of `<byte-string>`.

This function is thread-safe.

**force-out Function**

Forces pending output from `*standard-output*` to the operating system.

**Signature**

`force-out () => ()`

**Discussion**

Forces pending output from `*standard-output*` to the operating system using `force-output`.

This function is thread-safe.

**format-err Generic function**

Formats its arguments on the standard error.

**Signature**

`format-err control-string #rest arguments => ()`

**Parameters**

- **control-string** – An instance of `<string>`.
- **arguments (#rest)** – Instances of `<object>`.

**Discussion**

Calls the `format` function from the `format` module on `*standard-error*` from the `standard-io` module, `control-string`, and `arguments`.

This function is thread-safe.

**See also**

- `format`
- `*standard-error*`

**format-err(<byte-string>) Sealed Method**

Formats its arguments on the standard error.

**Signature**

`format-err control-string #rest arguments => ()`

**Parameters**

- **control-string** – An instance of `<byte-string>`.
- **arguments (#rest)** – Instances of `<object>`.

**Discussion**

Formats its arguments on the standard error. There is one method for `format-err`, and it is specialized to instances of `<byte-string>`.

This function is thread-safe.

**force-err Function**

Forces pending output from `*standard-error*` to the operating system.

10.2. The format-out Module
Signature  force-err () => ()

Discussion

Forces pending output from *standard-error* to the operating system using force-output.

This function is thread-safe.

10.3 The print and pprint Modules

The IO library’s printing modules provide an interface that outputs an object in Dylan literal syntax if the object can be represented as a Dylan literal, and otherwise, outputs the object in an implementation-dependent manner. There are two functions, print and print-object. The print function accepts keyword arguments that form a print request, controlling features such as circular printing, how deep within a data structure to print, how many elements in long sequences to print before using an ellipsis notation, whether pretty printing is desired, and so on. Users extend print’s ability to print various objects by adding methods to the print-object function. The print function handles most of the overhead to satisfy special print requests, outputting any special notations required, and it only calls print-object when it is necessary to print objects. Users should always call the print function to output objects, especially recursively from within print-object methods, to output an object’s components. Users should never call print-object directly.

The IO library exports two modules for use with printing, print and pprint. Reference entries for the interfaces exported from the print module can be found in The print Module, and reference entries for interfaces exported from the pprint module are in The pprint Module.

These modules use the Streams module. See The streams Module for full details of the Streams module.

10.3.1 Print functions

The Print module offers two functions for users to call to print objects, print and print-to-string.

The Print module exports the following variables which provide default values for calls to the print function. Their values are implementation-dependent.

- *print-level*
- *print-length*
- *print-circle?*
- *print-pretty?*
- *print-escape?*

10.3.2 Pretty printing

When writing print-object methods, you can ignore whether pretty printing is in effect. If you write your print-object method using pretty printing functions, then when pretty printing is in effect, the output is pretty printed. When pretty printing is not in effect, your method produces output as though you had not written it to use pretty printing. All print-object methods that are written to do pretty printing must call the pretty printing functions within the dynamic scope of a call to pprint-logical-block; otherwise, the pretty printing functions are no-ops.

The following interfaces are exported from the pprint module:

- *default-line-length*
- *print-miser-width*
10.3.3 The print Module

This section contains a reference entry for each item exported from the IO library’s print module.

**print Function**

Prints object to the specified stream.

**Signature**  
print object stream #key circle? escape? length level pretty? => ()

**Parameters**

- object – An instance of <object>.
- stream – An instance of <stream>.
- circle? (#key) – An instance of <boolean>. Default value: *print-circle?*.
- escape? (#key) – An instance of <boolean>. Default value: *print-escape?*.
- level (#key) – #f or an instance of <integer>. Default value: *print-level*.
- length (#key) – #f or an instance of <integer>. Default value: *print-length*.
- pretty? (#key) – An instance of <boolean>. Default value: *print-pretty?*.

**Discussion**

Prints object to stream according to the print request formed by the keyword arguments. A first call to print creates a printing stream to represent the print request, and recursive calls to print on this printing stream process the keyword arguments differently (see below). There are inspection functions for querying the print request. When print actually prints an object, it calls print-object. Though the inspection functions for querying the print request allow you to inspect any parameter of the print request, print-object methods should only need to call print-length. All other aspects of the print request are handled by print. There is one exception, which is described in Pretty printing.

The level keyword controls how deep into a nested data structure to print. The value #f indicates that there is no limit. The default, *print-level*, has no effect on recursive calls to print. Recursive calls to print may change the value of print-level explicitly, but print always uses a value to ensure the print request formed by the first call to print is never exceeded. For example, if a first call to print set the level to 5, and while at a depth of 3, a recursive call specified a level of 4, the recursive call would only descend 2 more levels, not 4.

The length keyword controls how many elements of a sequence to print before printing ellipsis notation (...). The value #f indicates that there is no limit. The print-length control can be interpreted loosely by some print-object methods to control how many elements of any kind of object to print; for example, the default <object> method might regard print-length to determine how many slot-name/value pairs to print. The default, *print-length*, has no effect on recursive calls to print. Recursive calls to print may change the value of print-length explicitly, but they may only decrease the value, never increase it.

The circle? keyword indicates whether printing should check all subcomponent references to make sure the printing process does not infinitely recurse through a data structure. Circular printing also tags objects that occur more than once when they are first printed, and later occurrences
are printed as a reference to the previously emitted tag. The default, \texttt{*print-circle?*}, has no effect on recursive calls to \texttt{print}. If \texttt{print-circle?} is already \#t, then it remains \#t throughout all recursive calls. If \texttt{print-circle?} is \#f, then recursive calls to \texttt{print} can change the value to \#t; however, when printing exits the dynamic scope of the call that changed the value to \#t, the value reverts back to \#f. If the original call to \texttt{print} specifies \texttt{circle?} as \#f, and dynamically distinct recursive calls turn circular printing on and off, all output generated while circular printing was on shares the same tagging space; that is, if \#1# is printed twice, once from each of two distinct recursive calls to \texttt{print}, then each \#1# is guaranteed to signify the same == object.

The \texttt{pretty?} keyword indicates whether printing should attempt to insert line breaks and indentation to format objects according to how programmers tend to find it easier to read data. The default, \texttt{*print-pretty?*}, has no effect on recursive calls to \texttt{print}. If \texttt{print-pretty?} is already \#t, then it remains \#t throughout all recursive calls. If \texttt{print-pretty?} is \#f, then recursive calls to \texttt{print} can change the value to \#t; however, when printing exits the dynamic scope of the call that changed the value to \#t, the value reverts back to \#f.

The \texttt{escape?} keyword indicates whether printing should show escape codes in strings and, by extension, whether other objects should be printed with additional identifying information.

\textbf{*print-circle?* Thread Variable}

Controls whether or not to detect circular structures when printing.

- **Type**: \langle boolean\rangle
- **Value**: \#f
- **Discussion**

  \texttt{*print-circle?*} holds the default value for whether or not to detect circular structures when printing. When \#t, circularity detection is enabled. The default is \#f.

  When calling \texttt{print} directly, \texttt{*print-circle?*} may be overridden by specifying a value for the \texttt{circle?} argument. It is dynamically bound by \texttt{print} so that its value is inherited by recursive calls.

  Note that when \texttt{*print-circle?*} is \#t attempts to print circular structures may result in failure to terminate.

  \textbf{See also}

  - \texttt{print} for more detail on how circular structures are handled.

\textbf{*print-escape?* Thread Variable}

Whether to print objects for debugging or for display to the end user.

- **Type**: \langle boolean\rangle
- **Value**: \#t
- **Discussion**

  Controls whether or not \texttt{print} displays extra identifying information for the object being printed, making it easier to distinguish from other objects of the same type for debugging purposes. For those who know Python, this is much like the difference between \texttt{__repr__} and \texttt{__str__}.

  For example, when \texttt{*print-escape?*} is true strings print exactly the way Dylan string literals look, for example "abc\n". When \texttt{*print-escape?*} is false the same string prints as \texttt{abc} with a trailing newline.

  \textbf{See also}
*print-length* • Thread Variable
Controls the number of elements of an expression to print.

Type false-or(<integer>)
Value #f
Discussion Controls how many elements to print at a given level of a nested expression. The default is unlimited (#f).

See also
• print

*print-level* • Thread Variable
Controls how deeply into a nested expression to print.

Type false-or(<integer>)
Value #f
Discussion Controls how many levels of a nested expression to print.

See also
• print

*print-pretty?* • Thread Variable
Controls whether or not pretty printing is used.

Type <boolean>
Value #f
Discussion Controls whether or not print does pretty printing by default. May be overridden via the pretty? keyword argument to print.

See also
• print

print-object Open Generic function
Prints an object to a stream.

Signature print-object object stream => ()
Parameters
• object – An instance of <object>.
• stream – An instance of <stream>.

Discussion
Prints an object to a stream. Extend the ability of print to print objects by adding methods to this generic function. When print actually prints an object, it calls print-object.

Never call print-object directly.

print-to-string Function
Calls print on object and returns the result as a string.

Signature print-to-string object #key circle? escape? level length pretty? => result
Parameters
• **object** – An instance of `<object>`.
• **level** (#key) – #f or an instance of `<integer>`. Default value: `*print-level*`.
• **length** (#key) – #f or an instance of `<integer>`. Default value: `*print-length*`.
• **circle?** (#key) – An instance of `<boolean>`. Default value: `*print-circle?*`.
• **pretty?** (#key) – An instance of `<boolean>`. Default value: `*print-pretty?*`.
• **escape?** (#key) – An instance of `<boolean>`. Default value: `*print-escape?*`.

**Values**

• **result** – An instance of `<byte-string>`.

**Discussion** Calls `print` to produce output according to the print request formed by the keyword arguments and returns the result as a string.

### 10.3.4 The `pprint` Module

This section contains a reference entry for each item exported from the IO library’s `pprint` module.

**default-line-length** Variable

Controls the default line length used by the pretty printer.

- **Type**: `<integer>`
- **Value**: 80

**Discussion** Controls the line length used by the pretty printer to determine how much output will fit on a single line. The value must be an integer.

**pprint-indent** Function

Specifies the indentation to use within the current logical block.

- **Signature**: `pprint-indent relative-to n stream => ()`

- **Parameters**
  - **relative-to** – One of `"block"` or `"current"`.
  - **n** – An instance of `<integer>`.
  - **stream** – An instance of `<stream>`.

**Discussion** Specifies the indentation to use within the current logical block. When `relative-to` is `"block"`, then `pprint-indent` sets the indentation to the column of the first character of the logical block plus `n`. When `relative-to` is `"current"`, then `pprint-indent` sets the indentation to the current column plus `n`.

**pprint-logical-block** Function

Groups printing into a logical block.

- **Signature**: `pprint-logical-block stream #key prefix per-line-prefix body suffix column => ()`

- **Parameters**
  - **stream** – An instance of `<stream>`.
  - **prefix** (#key) – #f or an instance of `<byte-string>`.
  - **per-line-prefix** (#key) – #f or an instance of `<byte-string>`.
  - **body** (#key) – An instance of `<function>`.
  - **suffix** (#key) – #f or an instance of `<byte-string>`.
• **column**(*#key*) – A *limited* instance of `<integer>`, minimum 0.

**Discussion**

Groups printing into a logical block. The logical block provides boundaries for new levels of indentation, affects `"linear"` newlines, and so on. *Prefix* is a string to print at the beginning of the logical block. The block’s indentation is automatically set to be one character position greater than the column in which *prefix* ends. Alternatively, *per-line-prefix* is a string to print on every line of the logical block. This function signals an error if it is called with both *prefix* and *per-line-prefix* supplied as non-`#f`. *Suffix* is a string to print at the end of the logical block. *Column* advises the pretty printer as to the current column of the output stream (the default is zero). The *column* argument may be ignored entirely by some methods, and it may be ignored in some cases by methods that can better determine the stream’s current output column.

The *body* keyword must be a function that can take one argument, and this argument is a stream. The *body* function should use the stream argument passed to it; the *body* function should not close over the stream argument to *pprint-logical-block*.*pprint-logical-block* wraps stream with a pretty printing stream when stream is any other kind of stream. If stream is already a pretty printing stream, then the *body* function is called on stream.

All *print-object* methods that are written to do pretty printing must call the other pretty printing functions within the dynamic scope of a call to *pprint-logical-block*; otherwise, the pretty printing functions are no-ops.

### *pprint-newline Function*

Announces a conditional newline to the pretty printer.

**Signature**  
`pprint-newline kind stream => ()`

**Parameters**

- **kind** – One of `"fill", "linear", "miser", "mandatory"`.
- **stream** – An instance of `<stream>`.

**Discussion**

Announces a conditional newline to the pretty printer. The pretty printer emits a newline depending on the *kind* and the state of the pretty printer’s current line buffer. The *kind* argument has roughly the following meanings:

- `"fill"` Emit a newline if the current *section* of output does not fit on one line.
- `"linear"` Emit a newline if any `"linear"` newline in the current *section* needs to be emitted. That is, if a current *section* of output cannot fit on one line, and any one of the `"linear"` newlines in the section needs to be emitted, then emit them all.
- `"miser"` Emit a newline as if it were a `"linear"` newline, but only when *miser mode* is in effect. Miser style is in effect when a logical block starts past a particular column of output.
- `"mandatory"` Emit a newline always. Establish that any containing *sections* cannot be printed on a single line so that `"linear"` and `"miser"` newlines will be emitted as appropriate.

### *pprint-tab Function*

Announces a tab to the pretty printer.

**Signature**  
`pprint-tab kind colnum colinc stream => ()`

**Parameters**
• **kind** – One of "line", "line-relative", "section", "section-relative".

• **colnum** – An instance of `<integer>`.

• **colinc** – An instance of `<integer>`.

• **stream** – An instance of `<stream>`.

**Discussion**

Announces a tab to the pretty printer. The *colnum* and *colinc* arguments have meaning based on the value of *kind*:

• "line" Tab to output column *colnum*. If the output is already at or beyond *colnum*, then add *colinc* to *colnum* until printing can continue at a column beyond the end of the output already on the line.

• "line-relative" Output *colnum* spaces. Then output enough spaces to tab to a column that is a multiple of *colinc* from the beginning of the line.

• "section" Similar to "line", but column counting is relative to the beginning of the current *section* rather than the beginning of the line.

• "section-relative" Similar to "line-relative", but column counting is relative to the beginning of the current *section* rather than the beginning of the line.

**print-miser-width** Variable

- **Type** false-or(<integer>)

- **Value** #f

**Discussion**

Controls miser mode. Pretty printing is in miser mode whenever a logical block (see `pprint-logical-block`) begins in a column of output that is greater than:

```
<default-line-length> = *print-miser-width*
```

The value must be an integer or #f (the default); #f indicates that the pretty printer should never enter miser mode.

### 10.4 The Streams Module

This document describes the Streams module, which allows you to establish and control input to and output from aggregates of data, such as files on disk, or sequences. This module, together with the Standard-IO module, provides similar functionality to the *java.io* package in Java. See *The standard-io Module*, for details about the Standard-IO module in Dylan.

*Concepts* discusses the basic concepts involved in streaming over data. *Stream classes* describes the different classes of stream available, and how to create them, and *Reading from and writing to streams* describes how to read from and write to them.

More specialized subjects are covered next: *Locking streams* discusses locking streams while they are in use; *Using buffered streams* describes using buffered streams; *Wrapper streams* describes wrapper streams; *Conditions* describes the different stream-specific error conditions that can be raised. For the most part, you do not have to worry about the information in these later sections when using streams.
Finally, *The streams Module Reference* gives complete details on all interfaces in the Streams module. Each entry in this section is arranged in alphabetical order.

### 10.4.1 Error conditions

This document uses two special terms in discussions of error conditions.

When it notes that something *is an error*, this means that the result is undefined. In particular, it does not*necessarily* mean that an error condition will be signalled. So, for instance, the following example text means only that the result of using `pull-stream-element` in the case described is undefined:

> It is an error to apply `pull-stream-element` to an element that has already been read from the stream.

A given function is only guaranteed to raise an exception in response to an error if the documentation for that function specifically states that it will signal an error. Note that the specific error condition that is signalled may depend on the program state; in such situations, the specific error condition is not stated in the documentation. Consider the following hypothetical example, which states that an implementation must signal an error, but does not say what error must be signalled:

> When `index` is a `<stream-index>`, if it is invalid for some reason, this function signals an error.

By contrast, the following example names the class of which the condition signalled is guaranteed to be a general instance:

> If the end of the stream is encountered and no value was supplied for `on-end-of-stream`, `read-last-element` signals an `<end-of-stream-error>` condition.

If the name of the condition class is given, applications are permitted to specialize error handlers on that class.

### 10.4.2 Goals of the module

The Streams module provides:

- A generic, easy-to-use interface for streaming over sequences and files. The same high-level interface for consuming or producing is available irrespective of the type of stream, or the types of the elements being streamed over.
- Efficiency, especially for the common case of file I/O.
- Access to an underlying buffer management protocol.

The Streams module does not address a number of related issues, including:

- A standard object-printing package such as Smalltalk’s `printOn` or Lisp’s `print-object`, or a formatted printing facility such as Lisp’s `format`. These facilities are provided by the *Print*, *Format*, and *Format-out* libraries. For convenience, the *Common Dylan* library also provides simple formatting capabilities.
- General object dumping and loading.
- A comprehensive range of I/O facilities for using memory-mapped files, network connections, and so on.
- An interface for naming files. The *Locators* module provides such an interface.
- An interface to operating system functionality, such as file renaming or deleting operations. The *File-System* module provides such an interface.
10.4.3 Concepts

A stream provides sequential access to an aggregate of data, such as a Dylan sequence or a disk file. Streams grant this access according to a metaphor of reading and writing: elements can be read from streams or written to them.

Streams are represented as Dylan objects, and all are general instances of the class `<stream>`, which the Streams module defines.

It is usual to say that a stream is established over the data aggregate. Hence, a stream providing access to the string "hello world" is said to be a stream over the string "hello world".

Streams permitting reading operations are called input streams. Input streams allow elements from the underlying data aggregate to be consumed. Conversely, streams permitting writing operations are called output streams. Output streams allow elements to be written to the underlying data aggregate. Streams permitting both kinds of operations are called input-output streams.

The Streams module provides a set of functions for reading elements from an input stream. These functions hide the details of indexing, buffering, and so on. For instance, the function `read-element` reads a single data element from an input stream.

The following expression binds `stream` to an input stream over the string "hello world":

```dylan
let stream = make(<string-stream>, contents: "hello world");
```

The first invocation of `read-element` on `stream` returns the character “h”, the next invocation “e”, and so on. Once a stream has been used to consume all the elements of the data, the stream is said to be at its end. This condition can be tested with the function `stream-at-end?`. The following code fragment applies `my-function` to all elements of the sequence:

```dylan
let stream = make(<sequence-stream>, contents: seq);
while (~stream-at-end?(stream))
  my-function(read-element(stream));
end;
```

When all elements of a stream have been read, further calls to `read-element` result in the `<end-of-stream-error>` condition being signaled. An alternative end-of-stream behavior is to have a distinguished end-of-stream value returned. You can supply such an end-of-stream value as a keyword argument to the various read functions; the value can be any object. Supplying an end-of-stream value to a read function is more concise than asking whether a stream is at its end on every iteration of a loop.

The Streams module also provides a set of functions for writing data elements to an output stream. Like the functions that operate upon input streams, these functions hide the details of indexing, growing an underlying sequence, buffering for a file, and so on. For instance, the function `write-element` writes a single data element to an output stream.

The following forms bind `stream` to an output stream over an empty string and create the string “I see!”, using the function `stream-contents` to access all of the stream’s elements.

```dylan
let stream = make(<byte-string-stream>, direction: #"output");
write(stream, "I see!");
stream-contents(stream);
```

Calling `write` on a sequence has the same effect as calling `write-element` on all the elements of the sequence. For more information about writing to streams, see Writing to streams.

Some streams are positionable; that is, any element of the stream can be accessed at any time. Positionable streams allow you to set the position at which the stream is accessed by the next operation. The following example uses positioning to return the character “w” from a stream over the string "hello world":

```dylan
let stream = make(<string-stream>, contents: "hello world");
position(stream, "hello world", 21);
read-element(stream);
```
let stream = make(<string-stream>, contents: "hello world");
stream-position(stream) := 6;
read-element(stream);

The following example returns a string. The first ten characters are the fill characters for the underlying sequence of the stream. The fill character for <string> is " " (the space character), so in the example below, the first ten characters are spaces.

let stream = make(<string-stream>, direction: #"output");
adjust-stream-position(stream, 10);
write(stream, "whoa!");
stream-contents(stream);

You can request a sequence containing all of the elements of a positionable stream by calling stream-contents on it. If the positionable stream is a <file-stream>, then it must be readable. Otherwise, it must be a sequence stream. The sequence returned never shares structure with any underlying sequence that might be used in the future by the stream. For instance, the string returned by calling stream-contents on an output <string-stream> will not be the same string as that being used to represent the string stream.

When making an input <string-stream>, you can cause the stream to produce elements from any subsequence of the supplied string. For example:

read-to-end(make(<string-stream>,
    contents: "hello there, world",
    start: 6,
    end: 11));

This example evaluates to "there". The interval (start, end) includes the index start but excludes the index end. This is consistent with standard Dylan functions over sequences, such as copy-sequence. The read-to-end function is one of a number of convenient utility functions for operating on streams and returns all the elements up to the end of the stream from the stream’s current position.

Streams, growing sequences, and object identity

When writing to output streams over sequences, Dylan may from time to time need to grow the underlying sequence that it is using to represent the stream data.

Consider the example of an output stream instantiated over an empty string. As soon as a write operation is performed on the stream, it is necessary to replace the string object used in the representation of the string stream. As well as incurring the cost of creating a new string, references to the string within the program after the replacement operation has occurred will still refer to the original string, and this may not be what the user intended.

To guarantee that other references to a sequence used in an output <sequence-stream> will have access to any elements written to the sequence via the stream, supply a stretchy collection (such as a <stretchy-vector>) to make. A stream over a stretchy vector will use the same stretchy vector throughout the stream’s existence.

For example:

let sv = make(<stretchy-vector>);
let stream = make(<sequence-stream>,
    contents: sv,
    direction: #"output");
write(stream, #(1, 2, 3, 4, 5, 6, 7, 8, 9));
write(stream, "ABCDEF");
values(sv, stream-contents(stream));

The example returns two values. Each value is the same (=) stretchy vector:
If a stretchy vector is not supplied, the result is different:

```dylan
def v = make(<vector>, size: 5);
def stream = make(<sequence-stream>,
    contents: v,
    direction: "output");
write(stream, #(1, 2, 3, 4, 5, 6, 7, 8, 9));
write(stream, "ABCDEF");
values(v, stream-contents(stream));
```

This example returns as its first value the original vector, whose contents are unchanged, but the second value is a new vector:

```dylan
(1, 2, 3, 4, 5, 6, 7, 8, 9, 'A', 'B', 'C', 'D', 'E', 'F')
```

This difference arises because the output stream in the second example does not use a stretchy vector to hold the stream data. A vector of at least 15 elements is necessary to accommodate the elements written to the stream, but the vector supplied, v, can hold only 5. Since the stream cannot change v’s size, it must allocate a new vector each time it grows.

### 10.4.4 Stream classes

The exported streams class heterarchy includes the classes shown in Streams module classes. Classes shown in bold are all instantiable.

- s - sealed
- o - open
- p - primary
- f - free
- c - concrete
- a - abstract
- u - uninstantiable
- i - instantiable

#### Streams module classes

- `<stream>`
- `<positionable-stream>`
- `<buffered-stream>`
- `<file-stream>`
- `<sequence-stream>`
- `<wrapper-stream>`
- `<indenting-stream>`

#### Creating streams

This section describes how to create and manage different types of file stream and sequence stream.
10.4. The streams Module
File streams

File streams are intended only for accessing the contents of files. More general file handling facilities, such as renaming, deleting, moving, and parsing directory names, are provided by the File-System module: see The file-system Module for details. The make method on `<file-stream>` does not create direct instances of `<file-stream>`, but instead an instance of a subclass determined by type-for-file-stream.

make file-stream-class

G.f method

  share-mode => file-stream-instance

Creates and opens a stream over a file, and returns a new instance of a concrete subclass of `<file-stream>` that streams over the contents of the file referenced by filename. To determine the concrete subclass to be instantiated, this method calls the generic function type-for-file-stream.

The locator: init-keyword should be a string naming a file. If the Locators library is in use, filename should be an instance of `<locator>` or a string that can be coerced to one.

The direction: init-keyword specifies the direction of the stream. This can be one of `#"input", #"output", or #"input-output"`. The default is `#"input"`.

The if-exists: and if-does-not-exist: init-keywords specify actions to take if the file named by filename does or does not already exist when the stream is created. These init-keywords are discussed in more detail in Options when creating file streams.

The buffer-size: init-keyword can be used to suggest the size of a stream’s buffer. See <buffered-stream>.

The element-type: init-keyword specifies the type of the elements in the file named by filename. See Options when creating file streams for more details.

Options when creating file streams

When creating file streams, you can supply the following init-keywords to make in addition to those described in File streams:

- if-exists: An action to take if the file already exists.
- if-does-not-exist: An action to take if the file does not already exist.
- element-type: How the elements of the underlying file are accessed.
- asynchronous?: Allows asynchronous writing of stream data to disk.
- share-mode: How the file can be accessed while the stream is operating on it.

The if-exists: init-keyword allows you to specify an action to take if the file named by filename already exists. The options are:

- #f The file is opened with the stream position at the beginning. This is the default when the stream’s direction is `#"input"` or `#"input-output"`.
- #"new-version" If the underlying file system supports file versioning, a new version of the file is created. This is the default when the stream’s direction is `#"output"`. If the file system does not support file versioning, the default is `#"replace"` when the direction of the stream is `#"output"`.
- #"overwrite" Set the stream’s position to the beginning of the file, but preserve the current contents of the file. This is useful when the direction is `#"input-output"` or `#"output"` and you want to overwrite an existing file.
- #"replace" Delete the existing file and create a new file.
• "append" Set the stream’s initial position to the end of the existing file so that all new output occurs at the end of the file. This option is only useful if the file is writeable.

• "truncate" If the file exists, it is truncated, setting the size of the file to 0. If the file does not exist, create a new file.

• "signal" Signal a <file-exists-error> condition.

The if-does-not-exist: init-keyword allows you to specify an action to take if the file named by filename does not exist. The options are:

• #f No action.

• "signal" Signal a <file-does-not-exist-error> condition. This is the default when the stream’s direction is "input".

• "create" Create a new zero-length file. This is the default when the stream’s direction is "output" or "input-output".

Because creating a file stream always involves an attempt to open the underlying file, the aforementioned error conditions will occur during file stream instance initialization.

File permissions are checked when creating and opening file streams, and if the user attempts to open a file for input, and has no read permission, or to open a file for output, and has no write permission, then an <invalid-file-permissions-error> condition is signalled at the time the file stream is created.

The element-type: init-keyword controls how the elements of the underlying file are accessed. The three possible element types are:

• <byte-character> The file is accessed as a sequence of 8-bit characters.

• <unicode-character> The file is accessed as a sequence of 16-bit Unicode characters.

• <byte> The file is accessed as a sequence of unsigned 8-bit integers.

The asynchronous?: init-keyword allows asynchronous writing of stream data to disk. If #f, whenever the stream has to write a buffer to disk, the thread which triggered the write must wait for the write to complete. If asynchronous? is #t, the write proceeds in parallel with the subsequent actions of the thread.

Note that asynchronous writes complicate error handling a bit. Any write error which occurs most likely occurs after the call which triggered the write. If this happens, the error is stored in a queue, and the next operation on that stream signals the error. If you close the stream with the wait? flag #f, the close happens asynchronously (after all queued writes complete) and errors may occur after close has returned. A method wait-for-io-completion is provided to catch any errors that may occur after close is called.

The share-mode: keyword determines how a file can be accessed by other streams while the stream has it open. The possible values are:

• "share-read" Allow other streams to be opened to the file for reading but not for writing.

• "share-write" Allow other streams to be opened for writing but not for reading.

• "share-read-write" Allow other streams to be opened for writing or reading.

• "exclusive" Do not allow other streams to be opened to this file.

### Sequence streams

There are make methods on the following stream classes:

• <sequence-stream>

• <string-stream>
Rather than creating direct instances of `<sequence-stream>` or `<string-stream>`, the `make` methods for those classes might create an instance of a subclass determined by `type-for-sequence-stream`.

- `make(<sequence-stream>)`
- `make(<string-stream>)`
- `make(<byte-string-stream>)`
- `make(<unicode-string-stream>)`

Closing streams

It is important to call `close` on streams when you have finished with them. Typically, external streams such as `<file-stream>` and `<console-stream>` allocate underlying system resources when they are created, and these resources are not recovered until the stream is closed. The total number of such streams that can be open at one time may be system dependent. It may be possible to add reasonable finalization methods to close streams when they are no longer referenced but these are not added by default. See the *The finalization Module* for full details about finalization.

Locking streams

In an application where more than one control thread may access a common stream, it is important to match the granularity of locking to the transaction model of the application. Ideally, an application should lock a stream which is potentially accessed by multiple threads, only once per transaction. Repeated and unnecessary locking and unlocking can seriously degrade the performance of the Streams module. Thus an application which wishes to write a complex message to a stream that needs to be thread safe should lock the stream, write the message and then unlock the stream after the entire message is written. Locking and unlocking the stream for each character in the message would be a poor match of locking to transaction model. The time required for the lock manipulation would dominate the time required for the stream transactions. Unfortunately this means that there is no way for the Streams module to choose a default locking scheme without the likelihood of seriously degrading streams performance for all applications whose transaction models are different from the model implied by the chosen default locking scheme. Instead, the Streams module provides the user with a single, per instance slot, `stream-lock`, which is inherited by all subclasses of `<stream>`. You should use the generic functions `lock-stream` and `unlock-stream` or the macro `with-stream-locked`, together with other appropriate functions and macros from the Threads library, to implement a locking strategy appropriate to your application and its stream transaction model. The functions in the Streams module are not of themselves thread safe, and make no guarantees about the atomicity of read and write operations.

10.4.5 Reading from and writing to streams

This section describes how you can read from or write to a stream. Note that it is an error to call any of these functions on a buffered stream while its buffer is held by another thread; see *Using buffered streams* for details about buffered streams.

Reading from streams

The following are the basic functions for reading from streams.

- `read-element`
- `read`
A number of other functions are available for reading from streams. See `peek`, `read-into!`, `discard-input`, and `stream-input-available?`.

**Convenience functions for reading from streams**

The following is a small set of reading functions that search for particular elements in a stream. These functions behave as though they were implemented in terms of the more primitive functions described in `Reading from streams`.

- `read-to`
- `read-to-end`
- `skip-through`

**Writing to streams**

This section describes the basic functions for writing to streams.

- `write-element`
- `write`

See `force-output`, `synchronize-output`, and `discard-output`.

**Reading and writing by lines**

The following functions provide line-based input and output operations.

The newline sequence for string streams is a sequence comprising the single newline character `\n`. For character file streams, the newline sequence is whatever sequence of characters the underlying platform uses to represent a newline. For example, on MS-DOS platforms, the sequence comprises two characters: a carriage return followed by a linefeed.

**Note:** No other functions in the Streams module do anything to manage the encoding of newlines; calling `write-element` on the character `\n` does not cause the `\n` character to be written as the native newline sequence, unless `\n` happens to be the native newline sequence.

- `read-line`
- `write-line`
- `new-line`

See also `read-line-into!`.

**Querying streams**

The following functions can be used to determine various properties of a stream.

- `stream-open?`
- `stream-element-type`
- `stream-at-end?`

For output streams, note that you can determine if a stream is one place past the last written element by comparing `stream-position` to `stream-size`. 
Using file streams

The following operations can be performed on file streams.

- `close(<file-stream>)`
- `stream-console?`
- `wait-for-io-completion`
- `with-open-file`

10.4.6 Using buffered streams

The Streams module provides efficient support for general use of buffered I/O. Most ordinary programmers using the module do not need to be concerned with buffering in most cases. When using buffered streams, the buffering is transparent, but programs requiring more control can access buffering functionality when appropriate. This section describes the available buffering functionality.

Overview

A buffered stream maintains some sort of buffer. All buffered streams use the sealed class `<buffer>` for their buffers. You can suggest a buffer size when creating buffered streams, but normally you do not need to do so, because a buffer size that is appropriate for the stream’s source or destination is chosen for you.

Instances of the class `<buffer>` also contain some state information. This state information includes an index where reading or writing should begin, and an index that is the end of input to be read, or the end of space available for writing.

Buffered streams also maintain a held state, indicating whether the application has taken the buffer for a stream and has not released it yet. When a thread already holds the buffer for a stream, it is an error to get the buffer again (or any other buffer for the same stream).

Useful types when using buffers

The following types are used in operations that involve buffers.

- `<byte>` A type representing limited integers in the range 0 to 255 inclusive.
- `<byte-character>` A type representing 8-bit characters that instances of `<byte-string>` can contain.
- `<unicode-character>` A type representing Unicode characters that instances of `<unicode-string>` can contain.
- `<byte-vector>` A subtype of `<vector>` whose element-type is `<byte>`.

10.4.7 Wrapper streams

Sometimes stream data requires conversion before an application can use it: you might have a stream over a file of UTF-8 bytes which you would prefer to handle as UTF-32 equivalents, or you might need to encrypt or decrypt file data.

Wrapper streams provide a mechanism for working with streams which require such conversion. Wrapper streams hold on to an underlying stream, delegating to it most streams operations. The wrapper stream carries out appropriate processing in its own implementations of the streaming protocol.
The Streams module includes a base class called `<wrapper-stream>` upon which other wrapping streams can be implemented. A subclass of `<wrapper-stream>` can “pass on” functions such as `read-element` and `write-element` by simply delegating these operations to the inner stream, as shown below:

```dylan
define method read-element
  (ws :: <io-wrapper-stream>, #key on-end-of-stream) => (element)
  read-element(ws.inner-stream, on-end-of-stream: on-end-of-stream)
end method;
define method write-element
  (ws :: <io-wrapper-stream>, element) => ()
  write-element(ws.inner-stream, element)
end method;
```

Assuming that `<io-wrapper-stream>` delegates all other operations to its inner stream, the following would suffice to implement a 16-bit Unicode character stream wrapping an 8-bit character stream.

```dylan
define class <unicode-stream> (<io-wrapper-stream>) end class;
define method read-element (s :: <unicode-stream>,
  #key on-end-of-stream) => (ch :: <unicode-character>)
  let first-char = read-element(s.inner-stream, on-end-of-stream);
  let second-char = read-element(s.inner-stream, on-end-of-stream)
  convert-byte-pair-to-unicode(first-char, second-char)
end method;
define method write-element (s :: <unicode-stream>,
  c :: <character>) => ()
  let (first-char, second-char) = 
    convert-unicode-to-byte-pair(c);
  write-element(s.inner-stream, first-char);
  write-element(s.inner-stream, second-char)
  c
end method;
define method stream-position (s :: <unicode-stream>) => p :: <integer>;
  truncate/(stream-position(s.inner-stream), 2)
end method;
define method stream-position-setter {p :: <integer>,
  s :: <unicode-stream>};
  stream-position(s.inner-stream) := p * 2
end method;
```

**Wrapper streams and delegation**

One problem with wrapper streams is the need for a wrapper stream to intercept methods invoked by its inner stream. For example, consider two hypothetical streams, `<interactihe-stream>` and `<dialog-stream>,
the latter a subclass of `<wrapper-stream>`. Both of these classes have a method called `prompt`. The `<interactive-stream>` class specializes `read` thus:

```dylan
define method read (s :: <interactive-stream>,
   n :: <integer>,
   #key on-end-of-stream);
   prompt(s);
   next-method()
end method;
```

If a `<dialog-stream>` is used to wrap an `<interactive-stream>` then an invocation of `read` on the `<dialog-stream>` will call `prompt` on the inner `<interactive-stream>`, not on the `<dialog-stream>`, as desired. The problem is that the `<dialog-stream>` delegates some tasks to its inner stream, but handles some other tasks itself.

Delegation by inner-streams to outer-streams is implemented by the use of the `outer-stream` function. The `outer-stream` function is used instead of the stream itself whenever a stream invokes one of its other protocol methods.

A correct implementation of the `read` method in the example above would be as follows:

```dylan
define method read (stream :: <interactive-stream>,
   n :: <integer>,
   #key on-end-of-stream)
   prompt(s.outer-stream);
   next-method()
end method;
```

The `initialize` method on `<stream>` is defined to set the `outer-stream` slot to be the stream itself. The `initialize` method on `<wrapper-stream>` is specialized to set the `outer-stream` slot to be the “parent” stream:

```dylan
define method initialize (stream :: <wrapper-stream>,
   #key on, #rest all-keys);
   an-inner-stream.outer-stream := stream;
   next-method()
end method;
```

### 10.4.8 Conditions

The following classes are available for error conditions on streams.

- `<end-of-stream-error>`
- `<incomplete-read-error>`
- `<file-error>`
- `<file-exists-error>`
- `<file-does-not-exist-error>`
- `<invalid-file-permissions-error>`

There is no recovery protocol defined for any of these errors. Every condition that takes an init-keyword has a slot accessor for the value supplied. The name of this accessor function takes the form `class - key`, where `class` is the name of the condition class (without the angle brackets) and `key` is the name of the init-keyword. For example, the accessor function for the `locator` init-keyword for `<file-error>` is `file-error-locator`.

For more information, please refer to the reference entry for the individual conditions.
10.4.9 Indenting streams

The Streams module provides an `<indenting-stream>` which supports managing indentation when printing text to a stream. Indenting streams are implemented as wrapper streams, so the destination stream must be provided at instantiation.

```dylan
let is = make(<indenting-stream>, inner-stream: *standard-output*);
with-indentation(is, 4)
   // Write normally to the indenting stream.
   format(is, "Hello %=!\n" name);
end with-indentation;
```

Indenting streams analyze everything written to them so that indentation can be maintained, without having to call `new-line` directly.

Using indenting streams

All operations available to `<wrapper-stream>` are available, as well as:

- `indent`
- `with-indentation`

10.4.10 Streams protocols

This section describes the protocols for different classes of stream.

Positionable stream protocol

This section describes the protocol for positionable streams.

A stream position can be thought of as a natural number that indicates how many elements into the stream the stream’s current location is. However, it is not always the case that a single integer contains enough information to reposition a stream. Consider the case of an “uncompressing” file stream that requires additional state beyond simply the file position to be able to get the next input character from the compressed file.

The Streams module addresses this problem by introducing the class `<stream-position>`, which is subclassed by various kinds of stream implementations that need to maintain additional state. A stream can be repositioned as efficiently as possible when `stream-position-setter` is given a value previously returned by `stream-position` on that stream.

It is also legal to set the position of a stream to an integer position. However, for some types of streams, to do so might be slow, perhaps requiring the entire contents of the stream up to that point to be read.

- `<position-type>`
- `<stream-position>`
- `stream-position`
- `stream-position-setter`
- `adjust-stream-position`
- `as(<integer>, <stream-position>)`
- `stream-size`
- `stream-contents`
Wrapper stream protocol

This section describes the protocol for implementing wrapper streams. For information on using wrapper streams, see Wrapper streams.

- `<wrapper-stream>`
- `inner-stream`
- `inner-stream-setter`
- `outer-stream`
- `outer-stream-setter`

10.4.11 The streams Module Reference

This section includes complete reference entries for all interfaces that are exported from the `streams` module.

`adjust-stream-position` Open Generic function

Moves the position of a positionable stream by a specified amount.

**Signature**

```dylan
adjust-stream-position positionable-stream delta #key from => new-position
```

**Parameters**

- `positionable-stream` – An instance of `<positionable-stream>`.
- `delta` – An instance of `<integer>`.
- `from` (#key) – One of `"current"`, `"start"`, or `"end"`. Default value: `"current"`.

**Values**

- `new-position` – An instance of `<stream-position>`.

**Discussion**

Moves the position of `positionable-stream` to be offset `delta` elements from the position indicated by `from`. The new position is returned.

- When `from` is `"start"`, the stream is positioned relative to the beginning of the stream. When `from` is `"end"`, the stream is positioned relative to its end. When `from` is `"current"`, the current position is used.

- Using `adjust-stream-position` to set the position of a stream to be beyond its current last element causes the underlying aggregate to be grown to a new size. When extending the underlying aggregate for a stream, the contents of the unwritten elements are the fill character for the underlying sequence.

**Example**

The following example returns a string, the first ten characters of which are the space character, which is the fill character for the sequence `<string>`.

```dylan
let stream = make(<string-stream>,
    direction: "output");
adjust-stream-position(stream, 10);
write(stream, "whoa!");
stream-contents(stream);
```
as(<integer>, <stream-position>) Method
Coerces a <stream-position> to an integer.

Signature as integer-class stream-position => integer

Parameters
- integer-class – The class <integer>.
- stream-position – An instance of <stream-position>.

Values
- integer – An instance of <integer>.

Discussion Coerces a <stream-position> to an integer. The integer-class argument is the class <integer>.

See also
- as

<buffer> Instantiable Sealed Class
A subclass of <vector> whose element-type is <byte>.

Superclasses <vector>

Init-Keywords
- size – An instance of <integer> specifying the size of the buffer. Default value: 0.
- next – An instance of <integer>. For an input buffer, this is where the next input byte can be found. For an output buffer, this is where the next output byte should be written to. Default value: 0.
- end – An instance of <integer>. The value of this is one more than the last valid index in a buffer. For an input buffer, this represents the number of bytes read.

Discussion
A subclass of <vector> whose element-type is <byte>.
Instances of <buffer> contain a data vector and two indices: the inclusive start and the exclusive end of valid data in the buffer. The accessors for these indexes are called buffer-next and buffer-end.

Note that size: is not taken as a suggestion of the size the user would like, as with the value passed with buffer-size: to make on <buffered-stream>; if you supply a value with the size: init-keyword, that size is allocated, or, if that is not possible, an error is signalled, as with making any vector.

<buffered-stream> Open Abstract Class
A subclass of <stream> supporting the Stream Extension and Buffer Access protocols.

Superclasses <stream>

Init-Keywords
- buffer-size – An instance of <integer>. This is the size of the buffer in bytes.

Discussion

Streams of this class support the `buffer-size: init-keyword`, which can be used to suggest the size of the stream’s buffer. However, the instantiated stream might not use this value: it is taken purely as a suggested value. For example, a stream that uses a specific device’s hardware buffer might use a fixed buffer size regardless of the value passed with the `buffer-size: init-keyword`.

In general, it should not be necessary to supply a value for the `buffer-size: init-keyword`.

### `<byte>` Type

**Type** A type representing limited integers in the range 0 to 255 inclusive.

**Supertypes** `<integer>`

**Discussion** A type representing limited integers in the range 0 to 255 inclusive.

**Operations**

- `type-for-file-stream`

### `<byte-character>` Type

**Type** A type representing 8-bit characters that instances of `<byte-string>` can contain.

**Supertypes** `<character>`

**Discussion** A type representing 8-bit characters that instances of `<byte-string>` can contain.

**Operations**

- `type-for-file-stream`

#### byte-storage-address(<buffer>) Sealed Method

Returns the address of the raw byte storage of a `<buffer>`.

**See also**

- `byte-storage-address`

#### byte-storage-offset-address(<buffer>) Sealed Method

Returns the address of the raw byte storage of a `<buffer>`, with an offset.

**See also**

- `byte-storage-offset-address`

### `<byte-string-stream>` Open Instantiable Class

The class of streams over byte strings.

**Superclasses** `<string-stream>`

**Init-Keywords**

- `contents` – A general instance of `<sequence>`.
- `direction` – Specifies the direction of the stream. It must be one of `"input"`, `"output"`, or `"input-output"`. Default value: `"input"`.
- `start` – An instance of `<integer>`. This specifies the start position of the sequence to be streamed over. Only valid when `direction: is "input"`. Default value: 0.
- `end` – An instance of `<integer>`. This specifies the sequence position immediately after the portion of the sequence to stream over. Only valid when `direction: is "input"`. Default value: `contents.size`. 
Discussion

The class of streams over byte strings. It is a subclass of `<string-stream>`.

The class supports the same init-keywords as `<sequence-stream>`.

The `contents:` init-keyword is used as the input for an input stream, and as the initial storage for an output stream.

The `start:` and `end:` init-keywords specify the portion of the byte string to create the stream over: `start:` is inclusive and `end:` is exclusive. The default is to stream over the entire byte string.

Operations

- `make(<byte-string-stream>)`

See also

- `make(<byte-string-stream>)`
- `<sequence-stream>`

`<byte-vector>` Sealed Class

A subtype of `<vector>` whose element-type is `<byte>`.

Superclasses `<vector>`

Keyword See Superclasses.

Discussion A subclass of `<vector>` whose element-type is `<byte>`.

See also

- `<byte>`

`close` Open Generic function

Closes a stream.

Signature `close stream #key #all-keys => ()`

Parameters

- `stream` – An instance of `<stream>`.

Discussion Closes stream, an instance of `<stream>`.

`close(<file-stream>)` Method

Closes a file stream.

Signature `close file-stream #key abort? wait? => ()`

Parameters

- `file-stream` – An instance of `<file-stream>`.
- `abort? (#key)` – An instance of `<boolean>`. Default value: `#f`.
- `wait? (#key)` – An instance of `<boolean>`.

Discussion

Closes a file stream. This method frees whatever it can of any underlying system resources held on behalf of the stream.

If `abort` is false, any pending data is forced out and synchronized with the file’s destination. If `abort` is true, then any errors caused by closing the file are ignored.

See also
• close

discard-input Open Generic function
Discards input from an input stream.

Signature  discard-input input-stream => ()

Parameters

• input-stream – An instance of <stream>.

Discussion
Discards any pending input from input-stream, both buffered input and, if possible, any input that might be at the stream’s source.

This operation is principally useful for “interactive” streams, such as TTY streams, to discard unwanted input after an error condition arises. There is a default method on <stream> so that applications can call this function on any kind of stream. The default method does nothing.

See also

• discard-output

discard-output Open Generic function
Discards output to an output stream.

Signature  discard-output output-stream => ()

Parameters

• output-stream – An instance of <stream>.

Discussion
Attempts to abort any pending output for output-stream.

A default method on <stream> is defined, so that applications can call this function on any sort of stream. The default method does nothing.

See also

• discard-input

<end-of-stream-error> Class
Error type signaled on reaching the end of an input stream.

Superclasses  <error>

Init-Keywords

• stream – An instance of <stream>.

Discussion
Signalled when one of the read functions reaches the end of an input stream. It is a subclass of <error>.

The stream: init-keyword has the value of the stream that caused the error to be signaled. Its accessor is end-of-stream-error-stream.

See also

• <file-does-not-exist-error>
• <file-error>
• <file-exists-error>
- `<incomplete-read-error>`
- `<invalid-file-permissions-error>`

### `<file-does-not-exist-error>` Class
Error type signaled when attempting to read a file that does not exist.

**Superclasses** `<file-error>`

**Keyword**  See Superclasses.

**Discussion** Signaled when an input file stream creation function tries to read a file that does not exist. It is a subclass of `<file-error>`.

See also
- `<end-of-stream-error>`
- `<file-error>`
- `<file-exists-error>`
- `<incomplete-read-error>`
- `<invalid-file-permissions-error>`

### `<file-error>` Class
The base class for all errors related to file I/O.

**Superclasses** `<error>`

**Init-Keywords**
- `locator`  – An instance of `<locator>`.

**Discussion**
The base class for all errors related to file I/O. It is a subclass of `<error>`.

The `locator` init-keyword indicates the locator of the file that caused the error to be signalled. Its accessor is `file-error-locator`.

See also
- `<end-of-stream-error>`
- `<file-does-not-exist-error>`
- `<file-exists-error>`
- `<incomplete-read-error>`
- `<invalid-file-permissions-error>`

### `<file-exists-error>` Class
Error type signaled when trying to create a file that already exists.

**Superclasses** `<file-error>`

**Keyword**  See Superclasses.

**Discussion** Signalled when an output file stream creation function tries to create a file that already exists. It is a subclass of `<file-error>`.

See also
- `<end-of-stream-error>`
- `<file-does-not-exist-error>`
<file-stream> Open Abstract Instantiable Class

The class of single-buffered streams over disk files.

Superclasses  <buffered-stream>, <positionable-stream>

Init-Keywords

• locator – An instance of <string> or <locator>. This specifies the file over which to stream.

• direction – Specifies the direction of the stream. It must be one of #"input", #"output", or #"input-output". Default value: #"input".

• if-exists – One of #f, #"new-version", #"overwrite", #"replace", #"append", #"truncate", #"signal". Default value: #f.

• if-does-not-exist – One of #f, #"signal", or #"create". Default value: depends on the value of direction:

• asynchronous? – If #t, all writes on this stream are performed asynchronously. Default value: #f.

Discussion

The class of single-buffered streams over disk files. It is a subclass of <positionable-stream> and <buffered-stream>.

When you instantiate this class, an indirect instance of it is created. The file being streamed over is opened immediately upon creating the stream.

The class supports several init-keywords: locator, direction, if-exists, and if-does-not-exist:

Operations

• close(<file-stream>)

• make(<file-stream>)

See also

• make(<file-stream>)

force-output Open Generic function

Forces pending output from an output stream buffer to its destination.

Signature  force-output output-stream #key synchronize? => ()

Parameters

• output-stream – An instance of <stream>.

• synchronize? (#key) – An instance of <boolean>. Default value: #f.

Discussion  Forces any pending output from output-stream’s buffers to its destination. Even if the stream is asynchronous, this call waits for all writes to complete. If synchronize? is true, also flushes the operating system’s write cache for the file so that all data is physically written to disk. This should only be needed if you’re concerned about system failure causing loss of data.

See also

• synchronize-output
<incomplete-read-error> Class
Error type signaled on encountering the end of a stream before reading the required number of elements.

Superclasses  <end-of-stream-error>

Init-Keywords
• sequence – An instance of <sequence>.
• count – An instance of <integer>.

Discussion
This error is signaled when input functions are reading a required number of elements, but the end of the stream is read before completing the required read.

The sequence: init-keyword contains the input that was read before reaching the end of the stream. Its accessor is incomplete-read-error-sequence.

The count: init-keyword contains the number of elements that were requested to be read. Its accessor is incomplete-read-error-count.

See also
• <end-of-stream-error>
• <file-does-not-exist-error>
• <file-error>
• <file-exists-error>
• <invalid-file-permissions-error>

<indenting-stream> Instantiable Sealed Class
A wrapper stream which outputs indented text.

Superclasses  <wrapper-stream>

Init-Keywords
• inner-stream – An instance of <stream>. Inherited from <wrapper-stream>.
• indentation – An instance of <integer>. Default value is 0.
• input-tab-width – An instance of <integer>. Default value is 8.
• output-tab-width – An instance of #f or <integer>. Default value is #f.

Discussion
A wrapper stream which outputs indented text.

The initial indentation is controlled by indentation:.

When output-tab-width: is not false, then the indenting stream converts sequences of spaces used for indentation to tabs.

Operations
• indent

  with-indentation

indent Generic function
Alters the indentation level of an <indenting-stream>.

Signature  indent stream delta => ()
Parameters

- **stream** – An instance of `<indenting-stream>`.
- **delta** – An instance of `<integer>`.

Example

```dylan
let is = make(<indenting-stream>, inner-stream: *standard-output*);
indent(is, 4);
format(is, "Hello, %n", name);
indent(is, -4);
```

See also

- `<indenting-stream>`
- `with-indentation`

**inner-stream**

Open Generic function

Returns the stream being wrapped.

**Signature**

inner-stream wrapper-stream => wrapped-stream

**Parameters**

- **wrapper-stream** – An instance of `<wrapper-stream>`.

**Values**

- **wrapped-stream** – An instance of `<stream>`.

**Discussion**

Returns the stream wrapped by `wrapper-stream`.

See also

- `inner-stream-setter`
- `outer-stream`
- `<wrapper-stream>`

**inner-stream-setter**

Open Generic function

Wraps a stream with a wrapper stream.

**Signature**

inner-stream-setter stream wrapper-stream => stream

**Parameters**

- **stream** – An instance of `<stream>`.
- **wrapper-stream** – An instance of `<wrapper-stream>`.

**Values**

- **stream** – An instance of `<stream>`.

**Discussion**

Wraps `stream` with `wrapper-stream`. It does so by setting the `inner-stream` slot of `wrapper-stream` to `stream`, and the `outer-stream` slot of `stream` to `wrapper-stream`.

**Note:** Applications should not set `inner-stream` and `outer-stream` slots directly. The `inner-stream-setter` function is for use only when implementing stream classes.

See also
• inner-stream
• outer-stream-setter

<invalid-file-permissions-error> Class
Error type signalled when accessing a file in a way that conflicts with the permissions of the file.

Superclasses <file-error>
Keyword See Superclasses.

Discussion Signalled when one of the file stream creation functions tries to access a file in a manner for which the user does not have permission. It is a subclass of <file-error>.

See also
• <end-of-stream-error>
• <file-does-not-exist-error>
• <file-error>
• <file-exists-error>
• <incomplete-read-error>

lock-stream Open Generic function
Locks a stream.

Signature lock-stream stream
Parameters
• stream – An instance of <stream>.

Discussion
Locks a stream. It is suggested that with-stream-locked be used instead of direct usages of lock-stream and unlock-stream.

See Locking streams for more detail and discussion on using streams from multiple threads.

See also
• stream-lock
• stream-lock-setter
• unlock-stream
• with-stream-locked

make(<byte-string-stream>) Method
Creates and opens a stream over a byte string.

Signature make byte-string-stream-class #key contents direction start end => byte-string-stream-instance
Parameters

• byte-string-stream-class – The class <byte-string-stream>.
• contents (#key) – An instance of <string>.
• direction (#key) – One of "input", "output", or "input-output". Default value: "input".
• start (#key) – An instance of <integer>. Default value: 0.
**end** (#key) – An instance of `<integer>`. Default value: `contents.size`.

**Values**

- **byte-string-stream-instance** – An instance of `<byte-string-stream>`.

**Discussion**

Creates and opens a stream over a byte string.

This method returns an instance of `<byte-string-stream>`. If supplied, `contents` describes the contents of the stream. The `direction`, `start`, and `end` init-keywords are as for `make` on `<sequence-stream>`.

**Example**

```
let stream = make(<byte-string-stream>,
    direction: #"output");
```

**See also**

- `<byte-string-stream>`
- `make(<sequence-stream>)`

### `make(<file-stream>)` Method

Creates and opens a stream over a file.

**Signature**

```
make file-stream-class #key filename direction if-exists if-does-not-exist buffer-size element-type => file-stream-instance
```

**Parameters**

- **file-stream-class** – The class `<file-stream>`.
- **filename (#key)** – An instance of `<object>`.
- **direction (#key)** – One of #"input", #"output", or #"input-output". The default is #"input".
- **if-exists (#key)** – One of #f, #"new-version", #"overwrite", #"replace", #"append", #"truncate", #"signal". Default value: #f.
- **if-does-not-exist (#key)** – One of #f, #"signal", or #"create". Default value: depends on the value of `direction`.
- **buffer-size (#key)** – An instance of `<integer>`.
- **element-type (#key)** – One of `<byte-character>`, `<unicode-character>`, or `<byte>`, or #f.

**Values**

- **file-stream-instance** – An instance of `<file-stream>`.

**Discussion**

Creates and opens a stream over a file.

Returns a new instance of a concrete subclass of `<file-stream>` that streams over the contents of the file referenced by `filename`. To determine the concrete subclass to be instantiated, this method calls the generic function `type-for-file-stream`.

The `filename` init-keyword should be a string naming a file. If the `Locators` library is in use, `filename` should be an instance of `<locator>` or a string that can be coerced to one.

The `direction` init-keyword specifies the direction of the stream.
The if-exists and if-does-not-exist init-keywords specify actions to take if the file named by filename does or does not already exist when the stream is created. These init-keywords are discussed in more detail in Options when creating file streams.

The buffer-size init-keyword is explained in <buffered-stream>.

The element-type init-keyword specifies the type of the elements in the file named by filename. This allows file elements to be represented abstractly; for instance, contiguous elements could be treated as a single database record. This init-keyword defaults to something useful, potentially based on the properties of the file; byte-character and unicode-character are likely choices. See Options when creating file streams.

See also

- <buffered-stream>
- <file-stream>
- type-for-file-stream

make(<sequence-stream>) Method

Creates and opens a stream over a sequence.

**Signature**

\[\text{make \ sequence-stream-class \ #key \ contents \ direction \ start \ end} \Rightarrow \text{sequence-stream-instance} \]

**Parameters**

- sequence-stream-class – The class <sequence-stream>.
- contents (#key) – An instance of <string>.
- direction (#key) – One of #"input", #"output", or #"input-output". Default value: #"input".
- start (#key) – An instance of <integer>. Default value: 0.
- end (#key) – An instance of <integer>. Default value: contents.size.

**Values**

- sequence-stream-instance – An instance of <sequence-stream>.

**Discussion**

Creates and opens a stream over a sequence.

This method returns a general instance of <sequence-stream>. To determine the concrete subclass to be instantiated, this method calls the generic function type-for-sequence-stream.

The contents init-keyword is a general instance of <sequence> which is used as the input for input streams, and as the initial storage for an output stream. If contents is a stretchy vector, then it is the only storage used by the stream.

The direction init-keyword specifies the direction of the stream.

The start and end init-keywords are only valid when direction is #"input". They specify the portion of the sequence to create the stream over: start is inclusive and end is exclusive. The default is to stream over the entire sequence.

**Example**
let sv = make(<stretchy-vector>);
let stream = make(<sequence-stream>,
    contents: sv,
    direction: #"output");
write(stream, #(1, 2, 3, 4, 5, 6, 7, 8, 9));
write(stream, "ABCDEF");
values(sv, stream-contents(stream));

See also

• <sequence-stream>
• type-for-sequence-stream

make(<string-stream>) Method
Creates and opens a stream over a string.

Signature  make string-stream-class #key contents direction start end => string-stream-instance

Parameters

• string-stream-class – The class <string-stream>.
• contents (#key) – An instance of <string>.
• direction (#key) – One of #"input", #"output", or #"input-output". Default value: #"input".
• start (#key) – An instance of <integer>. Default value: 0.
• end (#key) – An instance of <integer>. Default value: contents.size.

Values

• string-stream-instance – An instance of <string-stream>.

Discussion
Creates and opens a stream over a string.

This method returns an instance of <string-stream>. If supplied, contents describes the contents of the stream. The direction, start, and end init-keywords are as for make on <sequence-stream>.

Example

let stream = make(<string-stream>,
    contents: "here is a sequence");

See also

• make(<sequence-stream>)
• <string-stream>

make(<unicode-string-stream>) Method
Creates and opens a stream over a Unicode string.

Signature  make unicode-string-stream-class #key contents direction start end => unicode-string-stream-instance

Parameters

• unicode-string-stream-class – The class <unicode-string-stream>.
• contents (#key) – An instance of <unicode-string>.
• **direction** (#key) – One of #"input", #"output", or #"input-output". Default value: #"input".

• **start** (#key) – An instance of <integer>. Default value: 0.

• **end** (#key) – An instance of <integer>. Default value: contents.size.

**Values**

• **unicode-string-stream-instance** – An instance of <unicode-string-stream>.

**Discussion**

Creates and opens a stream over a Unicode string.

This method returns a new instance of <unicode-string-stream>. If supplied, contents describes the contents of the stream, and must be an instance of <unicode-string>. The direction, start, and end init-keywords are as for make on <sequence-stream>.

See also

• make(<sequence-stream>)

• <unicode-string-stream>

**new-line** Open Generic function

Writes a newline sequence to an output stream.

**Signature**

new-line output-stream => ()

**Parameters**

• **output-stream** – An instance of <stream>.

**Discussion**

Writes a newline sequence to output-stream.

A method for new-line is defined on <string-stream> that writes the character \n to the string stream.

**outer-stream** Open Generic function

Returns a stream’s wrapper stream.

**Signature**

outer-stream stream => wrapping-stream

**Parameters**

• **stream** – An instance of <stream>.

**Values**

• **wrapping-stream** – An instance of <wrapper-stream>.

**Discussion**

Returns the stream that is wrapping stream.

See also

• inner-stream

• outer-stream-setter

• <wrapper-stream>

**outer-stream-setter** Open Generic function

Sets a stream’s wrapper stream.

**Signature**

outer-stream-setter wrapper-stream stream => wrapper-stream
Parameters

- **wrapper-stream** – An instance of `<wrapper-stream>`.
- **stream** – An instance of `<stream>`.

Values

- **wrapper-stream** – An instance of `<wrapper-stream>`.

Discussion

Sets the `outer-stream` slot of `stream` to `wrapper-stream`.

Note: Applications should not set `inner-stream` and `outer-stream` slots directly. The `outer-stream-setter` function is for use only when implementing stream classes.

See also

- `inner-stream-setter`
- `outer-stream`

**peek** Open Generic function

Returns the next element of a stream without advancing the stream position.

Signature

```
peek input-stream #key on-end-of-stream => element-or-eof
```

Parameters

- **input-stream** – An instance of `<stream>`.
- **on-end-of-stream (#key)** – An instance of `<object>`.

Values

- **element-or-eof** – An instance of `<object>`, or `#f`.

Discussion

This function behaves as `read-element` does, but the stream position is not advanced.

See also

- `read-element`

**<positionable-stream>** Open Abstract Class

The class of positionable streams.

Superclasses `<stream>`

Keyword See Superclasses.

Discussion

A subclass of `<stream>` supporting the Positionable Stream Protocol. It is not instan-
tiable.

Operations

- `adjust-stream-position`
- `stream-contents`
- `stream-position`
- `stream-position-setter`
- `unread-element`

**<position-type>** Type
Type  A type representing positions in a stream.

Equivalent  type-union(<stream-position>, <integer>)

Supertypes  None.

Discussion  A type used to represent a position in a stream. In practice, positions within a stream are defined as instances of <integer>, but this type, together with the <stream-position> class, allows for cases where this might not be possible.

See also
• <stream-position>

read
Open Generic function
Reads a number of elements from an input stream.

Signature  read input-stream n #key on-end-of-stream => sequence-or-eof

Parameters
• input-stream – An instance of <stream>.
• n – An instance of <integer>.
• on-end-of-stream(#key) – An instance of <object>.

Values
• sequence-or-eof – An instance of <sequence>, or an instance of <object> if the end of stream is reached.

Discussion

Returns a sequence of the next n elements from input-stream.

The type of the sequence returned depends on the type of the stream’s underlying aggregate. For instances of <sequence-stream>, the type of the result is given by type-for-copy of the underlying aggregate. For instances of <file-stream>, the result is a vector that can contain elements of the type returned by calling stream-element-type on the stream.

The stream position is advanced so that subsequent reads start after the n elements.

If the stream is not at its end, read waits until input becomes available.

If the end of the stream is reached before all n elements have been read, the behavior is as follows.

• If a value for the on-end-of-stream argument was supplied, it is returned as the value of read.

• If a value for the on-end-of-stream argument was not supplied, and at least one element was read from the stream, then an <incomplete-read-error> condition is signaled. When signaling this condition, read supplies two values: a sequence of the elements that were read successfully, and n.

• If the on-end-of-stream argument was not supplied, and no elements were read from the stream, an <end-of-stream-error> condition is signalled.

See also
• <end-of-stream-error>
• <incomplete-read-error>
• stream-element-type
read-element Open Generic function
Reads the next element in a stream.

Signature  read-element input-stream #key on-end-of-stream => element-or-eof

Parameters

• input-stream – An instance of <stream>.
• on-end-of-stream(#key) – An instance of <object>.

Values

• element-or-eof – An instance of <object>.

Discussion
Returns the next element in the stream. If the stream is not at its end, the stream is advanced so that the next call to read-element returns the next element along in input-stream.

The on-end-of-stream keyword allows you to specify a value to be returned if the stream is at its end. If the stream is at its end and no value was supplied for on-end-of-stream, read-element signals an <end-of-stream-error> condition.

If no input is available and the stream is not at its end, read-element waits until input becomes available.

Example The following piece of code applies function to all the elements of a sequence:

```
let stream = make(<sequence-stream>, contents: seq);
while (~stream-at-end?(stream))
   function(read-element(stream));
end;
```

See also

• <end-of-stream-error>
• peek
• unread-element

read-into! Open Generic function
Reads a number of elements from a stream into a sequence.

Signature  read-into! input-stream n sequence #key start on-end-of-stream => count-or-eof

Parameters

• input-stream – An instance of <stream>.
• n – An instance of <integer>.
• sequence – An instance of <mutable-sequence>.
• start(#key) – An instance of <integer>.
• on-end-of-stream(#key) – An instance of <object>.

Values

• count-or-eof – An instance of <integer>, or an instance of <object> if the end of stream is reached..
Reads the next \( n \) elements from \( \text{input-stream} \), and inserts them into a mutable sequence starting at the position \( \text{start} \). Returns the number of elements actually inserted into \( \text{sequence} \) unless the end of the stream is reached, in which case the \( \text{on-end-of-stream} \) behavior is as for \( \text{read} \).

If the sum of \( \text{start} \) and \( n \) is greater than the size of \( \text{sequence} \), \( \text{read-into!} \) reads only enough elements to fill \( \text{sequence} \) up to the end. If \( \text{sequence} \) is a stretchy vector, no attempt is made to grow it.

If the stream is not at its end, \( \text{read-into!} \) waits until input becomes available.

See also

- \( \text{read} \)

\textbf{read-line} Open Generic function

Reads a stream up to the next newline.

\textbf{Signature} \( \text{read-line} \ \text{input-stream} \ #\text{key} \ \text{on-end-of-stream} \Rightarrow \text{string-or-eof} \ \text{newline}? \)

\textbf{Parameters}

- \textit{input-stream} – An instance of \(<\text{stream}>\).
- \textit{on-end-of-stream} (#key) – An instance of \(<\text{object}>\).

\textbf{Values}

- \textit{string-or-eof} – An instance of \(<\text{string}>\), or an instance of \(<\text{object}>\) if the end of the stream is reached.
- \textit{newline?} – An instance of \(<\text{boolean}>\).

\textbf{Discussion}

Returns a new string containing all the input in \( \text{input-stream} \) up to the next newline sequence.

The resulting string does not contain the newline sequence. The second value returned is \#t if the read terminated with a newline or \#f if the read terminated because it came to the end of the stream.

The type of the result string is chosen so that the string can contain characters of \( \text{input-stream} \) ’s element type. For example, if the element type is \text{byte-character}, the string will be a \text{byte-string}.

If \( \text{input-stream} \) is at its end immediately upon calling \( \text{read-line} \) (that is, the end of stream appears to be at the end of an empty line), then the end-of-stream behavior and the interpretation of \( \text{on-end-of-stream} \) is as for \( \text{read-element} \).

See also

- \( \text{read-element} \)

\textbf{read-line-into!} Open Generic function

Reads a stream up to the next newline into a string.

\textbf{Signature} \( \text{read-line-into!} \ \text{input-stream} \ \text{string} \ #\text{key} \ \text{start} \ \text{on-end-of-stream} \ \text{grow} \ ? \Rightarrow \text{string-or-eof} \ \text{newline}? \)

\textbf{Parameters}

- \textit{input-stream} – An instance of \(<\text{stream}>\).
- \textit{string} – An instance of \(<\text{string}>\).
- \textit{start} (#key) – An instance of \(<\text{integer}>\). Default value: 0.
- \textit{on-end-of-stream} (#key) – An instance of \(<\text{object}>\).
• **grow?** (#key) – An instance of `<boolean>`. Default value: `#f`.

**Values**

• **string-or-eof** – An instance of `<string>`, or an instance of `<object>` if the end of the stream is reached.

• **newline?** – An instance of `<boolean>`.

**Discussion**

Fills `string` with all the input from `input-stream` up to the next newline sequence. The `string` must be a general instance of `<string>` that can hold elements of the stream’s element type.

The input is written into `string` starting at the position `start`. By default, `start` is the start of the stream.

The second return value is `#t` if the read terminated with a newline, or `#f` if the read completed by getting to the end of the input stream.

If `grow?` is `#t`, and `string` is not large enough to hold all of the input, `read-line-into!` creates a new string which it writes to and returns instead. The resulting string holds all the original elements of `string`, except where `read-line-into!` overwrites them with input from `input-stream`.

In a manner consistent with the intended semantics of `grow?`, when `grow?` is `#t` and `start` is greater than or equal to `string.size`, `read-line-into!` grows `string` to accommodate the `start` index and the new input.

If `grow?` is `#f` and `string` is not large enough to hold the input, the function signals an error.

The end-of-stream behavior and the interpretation of `on-end-of-stream` is the same as that of `read-line`.

**See also**

• `read-line`

**read-through** Generic function

Returns a sequence containing the elements of the stream up to, and including, the first occurrence of a given element.

**Signature**

```
read-through input-stream element #key on-end-of-stream test => sequence-or-eof found?
```

**Parameters**

• **input-stream** – An instance of `<stream>`.

• **element** – An instance of `<object>`.

• **on-end-of-stream** (#key) – An instance of `<object>`.

• **test** (#key) – An instance of `<function>`. Default value: `==`.

**Values**

• **sequence-or-eof** – An instance of `<sequence>`, or an instance of `<object>` if the end of the stream is reached.

• **found?** – An instance of `<boolean>`.

**Discussion**

This function is the same as `read-to`, except that `element` is included in the resulting sequence.
If the element is not found, the result does not contain it. The stream is left positioned after element.

See also
• read-to

read-to Generic function
Returns a sequence containing the elements of the stream up to, but not including, the first occurrence of a given element.

Signature  read-to input-stream element #key on-end-of-stream test => sequence-or-eof found?

Parameters
• input-stream – An instance of <stream>.
• element – An instance of <object>.
• on-end-of-stream (#key) – An instance of <object>.
• test (#key) – An instance of <function>. Default value: ==.

Values
• sequence-or-eof – An instance of <sequence>, or an instance of <object> if the end of the stream is reached.
• found? – An instance of <boolean>.

Discussion
Returns a new sequence containing the elements of input-stream from the stream’s current position to the first occurrence of element. The result does not contain element.

The second return value is #t if the read terminated with element, or #f if the read terminated by reaching the end of the stream’s source. The “boundary” element is consumed, that is, the stream is left positioned after element.

The read-to function determines whether the element occurred by calling the function test. This function must accept two arguments, the first of which is the element retrieved from the stream first and the second of which is element.

The type of the sequence returned is the same that returned by read. The end-of-stream behavior is the same as that of read-element.

See also
• read-element

read-to-end Generic function
Returns a sequence containing all the elements up to, and including, the last element of the stream.

Signature  read-to-end input-stream => sequence

Parameters
• input-stream – An instance of <stream>.

Values
• sequence – An instance of <sequence>.

Discussion
Returns a sequence of all the elements up to, and including, the last element of input-stream, starting from the stream’s current position.
The type of the result sequence is as described for \texttt{read}. There is no special end-of-stream behavior; if the stream is already at its end, an empty collection is returned.

\textbf{Example}

\begin{verbatim}
read-to-end(make(\texttt{<string-stream>},
    contents: "hello there, world",
    start: 6,
    end: 11));
\end{verbatim}

\textbf{See also}

\begin{itemize}
  \item \texttt{read}
\end{itemize}

\textbf{\texttt{<sequence-stream> Open Class}}

The class of streams over sequences.

\textbf{Superclasses} \texttt{<positionable-stream>}

\textbf{Init-Keywords}

\begin{itemize}
  \item \texttt{contents} – A general instance of \texttt{<sequence>} which is used as the input for an input stream, and as the initial storage for an output stream.
  \item \texttt{direction} – Specifies the direction of the stream. It must be one of \texttt{"input"}, \texttt{"output"}, or \texttt{"input-output"}. Default value: \texttt{"input"}.
  \item \texttt{start} – An instance of \texttt{<integer>}.
    This specifies the start position of the sequence to be streamed over.
    Only valid when \texttt{direction:} is \texttt{"input"}.
    Default value: 0.
  \item \texttt{end} – An instance of \texttt{<integer>}.
    This specifies the sequence position immediately after the portion of the sequence to stream over.
    Only valid when \texttt{direction:} is \texttt{"input"}.
    Default value: \texttt{contents.size}.
\end{itemize}

\textbf{Discussion}

The class of streams over sequences. It is a subclass of \texttt{<positionable-stream>}.

If \texttt{contents:} is a stretchy vector, then it is the only storage used by the stream.

The \texttt{<sequence-stream>} class can be used for streaming over all sequences, but there are also subclasses \texttt{<string-stream>}, \texttt{<byte-string-stream>}, and \texttt{<unicode-string-stream>}, which are specialized for streaming over strings.

The \texttt{start:} and \texttt{end:} init-keywords specify the portion of the sequence to create the stream over: \texttt{start:} is inclusive and \texttt{end:} is exclusive. The default is to stream over the entire sequence.

\textbf{Operations}

\begin{itemize}
  \item \texttt{make(<sequence-stream>)}
\end{itemize}

\textbf{See also}

\begin{itemize}
  \item \texttt{<byte-string-stream>}
  \item \texttt{make(<sequence-stream>)}
  \item \texttt{<string-stream>}
  \item \texttt{<unicode-string-stream>}
\end{itemize}

\textbf{skip-through Generic function}

Skips through an input stream past the first occurrence of a given element.

\textbf{Signature} \texttt{skip-through input-stream element \#key test => found?}
Parameters

- **input-stream** – An instance of `<stream>`.
- **element** – An instance of `<object>`.
- **test (#key)** – An instance of `<function>`. Default value: `==`.

Values

- **found?** – An instance of `<boolean>`.

Discussion

Positions `input-stream` after the first occurrence of `element`, starting from the stream’s current position. Returns `#t` if the element was found, or `#f` if the end of the stream was encountered. When `skip-through` does not find `element`, it leaves `input-stream` positioned at the end.

The `skip-through` function determines whether the element occurred by calling the test function `test`. The test function must accept two arguments. The order of the arguments is the element retrieved from the stream first and element second.

**<stream> Open Abstract Class**

The superclass of all stream classes.

**Superclasses** `<object>`

**Init-Keywords**

- **outer-stream** – The name of the stream wrapping the stream. Default value: the stream itself (that is, the stream is not wrapped).

Discussion

The superclass of all stream classes and a direct subclass of `<object>`. It is not instantiable.

The `outer-stream`: init-keyword should be used to delegate a task to its wrapper stream. See `Wrapper streams and delegation` for more information.

**Operations**

- `close`
- `discard-input`
- `discard-output`
- `force-output`
- `lock-stream`
- `new-line`
- `outer-stream`
- `outer-stream-setter`
- `peek`
- `read`
- `read-element`
- `read-into!`
- `read-line`
- `read-line-into!`
• `read-through`
• `read-to`
• `read-to-end`
• `skip-through`
• `stream-at-end?`
• `stream-element-type`
• `stream-input-available?`
• `stream-lock`
• `stream-lock-setter`
• `stream-open?`
• `synchronize-output`
• `unlock-stream`
• `with-stream-locked`
• `write`
• `write-element`

**stream-at-end?** Open Generic function
Tests whether a stream is at its end.

**Signature** `stream-at-end? stream => at-end?`

**Parameters**
- `stream` – An instance of `<stream>`.

**Values**
- `at-end?` – An instance of `<boolean>`.

**Discussion**
Returns `#t` if the stream is at its end and `#f` if it is not. For input streams, it returns `#t` if a call to `read-element` with no supplied keyword arguments would signal an `<end-of-stream-error>`.

This function differs from `stream-input-available?`, which tests whether the stream can be read.

For output-only streams, this function always returns `#f`.

For output streams, note that you can determine if a stream is one place past the last written element by comparing `stream-position` to `stream-size`.

**Example** The following piece of code applies `function` to all the elements of a sequence:

```dylan
let stream = make(<sequence-stream>, contents: seq);
while (~stream-at-end?(stream))
    function(read-element(stream));
end;
```

**See also**
- `<end-of-stream-error>`
- **read-element**
- **stream-input-available?**

**stream-contents** Open Generic function
Returns a sequence containing all the elements of a positionable stream.

**Signature**  
stream-contents positionable-stream #key clear-contents? => sequence

**Parameters**
- **positionable-stream** – An instance of `<positionable-stream>`.
- **clear-contents? (#key)** – An instance of `<boolean>`. Default value: `#t`.

**Values**
- **sequence** – An instance of `<sequence>`.

**Discussion**

Returns a sequence that contains all of `positionable-stream` ‘s elements from its start to its end, regardless of its current position. The type of the returned sequence is as for `read`.

The `clear-contents?` argument is only applicable to writeable sequence streams, and is not defined for file-streams or any other external stream. It returns an error if applied to an input only stream. If clear-contents? is `#t` (the default for cases where the argument is defined), this function sets the size of the stream to zero, and the position to the stream’s start. Thus the next call to `stream-contents` will return only the elements written after the previous call to `stream-contents`.

Note that the sequence returned never shares structure with any underlying sequence that might be used in the future by the stream. For instance, the string returned by calling `stream-contents` on an output `<string-stream>` will not be the same string as that being used to represent the string stream.

**Example** The following forms bind `stream` to an output stream over an empty string and create the string “I see!”, using the function `stream-contents` to access all of the stream’s elements.

```dylan
let stream = make(<byte-string-stream>,
    direction: "output");
write-element(stream, 'I');
write-element(stream, ' '); 
write(stream, "see");
write-element(stream, '!');
stream-contents(stream);
```

**See also**
- **read-to-end**

**stream-element-type** Open Generic function
Returns the element-type of a stream.

**Signature**  
stream-element-type stream => element-type

**Parameters**
- **stream** – An instance of `<stream>`.

**Values**
- **element-type** – An instance of `<type>`.

**Discussion** Returns the element type of `stream` as a Dylan `<type>`.

10.4. The streams Module  289
stream-input-available? Open Generic function
Tests if an input stream can be read.

Signature  stream-input-available? input-stream => available?

Parameters
- • input-stream – An instance of <stream>.

Values
- • available? – An instance of <boolean>.

Discussion
Returns #t if input-stream would not block on read-element, otherwise it returns #f.

This function differs from stream-at-end?. When stream-input-available? returns #t, read-element will not block, but it may detect that it is at the end of the stream’s source, and consequently inspect the on-end-of-stream argument to determine how to handle the end of stream.

See also
- • read-element
- • stream-at-end?

stream-console? Open Generic function
Tests whether a stream is directed to the console.

Signature  stream-console? stream => console?

Parameters
- • stream – An instance of <file-stream>.

Values
- • console? – An instance of <boolean>.

Discussion
Returns #t if the stream is directed to the console and #f if it is not.

Example  The following piece of code tests whether stdout has been directed to the console (/example), or to a file (/example > file):

```dylan
if (stream-console?(*standard-output*))
    format-out("Output is directed to the console\n")
else
    format-out("Output is not directed to the console\n")
end if;
```

stream-lock Open Generic function
Returns the lock for a stream.

Signature  stream-lock stream => lock

Parameters
- • stream – An instance of <stream>.

Values
- • lock – An instance of <lock>, or #f.

Discussion
Returns lock for the specified stream. You can use this function, in conjunction with stream-lock-setter to implement a basic stream locking facility.
See also
  • stream-lock-setter

stream-lock-setter Open Generic function
Sets a lock on a stream.

Signature stream-lock-setter stream lock => lock

Parameters
  • stream – An instance of <stream>.
  • lock – An instance of <lock>, or #f.

Values
  • lock – An instance of <lock>, or #f.

Discussion Sets lock for the specified stream. If lock is #f, then the lock on stream is freed. You can use this function in conjunction with stream-lock to implement a basic stream locking facility.

See also
  • stream-lock

stream-open? Open Generic function
Generic function for testing whether a stream is open.

Signature stream-open? stream => open?

Parameters
  • stream – An instance of <stream>.

Values
  • open? – An instance of <boolean>.

Discussion Returns #t if stream is open and #f if it is not.

See also
  • close

stream-position Open Generic function
Finds the current position of a positionable stream.

Signature stream-position positionable-stream => position

Parameters
  • positionable-stream – An instance of <positionable-stream>.

Values
  • position – An instance of <position-type>.

Discussion
Returns the current position of positionable-stream for reading or writing.
The value returned can be either an instance of <stream-position> or an integer. When the value is an integer, it is an offset from position zero, and is in terms of the stream’s element type. For instance, in a Unicode stream, a position of four means that four Unicode characters have been read.
Example The following example uses positioning to return the character “w” from a stream over the string "hello world":

```dylan
let stream = make(<string-stream>,
    contents: "hello world");
stream-position(stream) := 6;
read-element(stream);
```

See also

&lt;position-type&gt;

&lt;stream-position&gt; Abstract Class
The class representing non-integer stream positions.

Superclasses &lt;object&gt;

Discussion
A direct subclass of &lt;object&gt;. It is used in rare cases to represent positions within streams that cannot be represented by instances of &lt;integer&gt;. For example, a stream that supports compression will have some state associated with each position in the stream that a single integer is not sufficient to represent.

The &lt;stream-position&gt; class is disjoint from the class &lt;integer&gt;.

Operations

- `as`
- `stream-position-setter`
- `stream-size`

See also

- &lt;position-type&gt;

`stream-position-setter` Open Generic function
Sets the position of a stream.

Signature `stream-position-setter position positionable-stream => new-position`

Parameters

- `position` – An instance of &lt;position-type&gt;.
- `positionable-stream` – An instance of &lt;positionable-stream&gt;.

Values

- `new-position` – An instance of &lt;stream-position&gt;, or an instance of &lt;integer&gt;.

Discussion
Changes the stream’s position for reading or writing to `position`.

When it is an integer, if it is less than zero or greater than `positionable-stream.stream-size` this function signals an error. For file streams, a `stream-position-error` is signalled. For other types of stream, the error signalled is `simple-error`.

When `position` is a &lt;stream-position&gt;, if it is invalid for some reason, this function signals an error. Streams are permitted to restrict the `position` to being a member of the set of values previously returned by calls to `stream-position` on the same stream.
The *position* may also be "start", meaning that the stream should be positioned at its start, or "end", meaning that the stream should be positioned at its end.

---

**Note:** You cannot use stream-position-setter to set the position past the current last element of the stream: use adjust-stream-position instead.

---

See also

- adjust-stream-position
- <stream-position>

---

**stream-size**

Open Generic function

Finds the number of elements in a stream.

**Signature**

stream-size positionable-stream => size

**Parameters**

- positionable-stream – An instance of <positionable-stream>.

**Values**

- size – An instance of <integer>, or #f.

**Discussion**

Returns the number of elements in positionable-stream.

For input streams, this is the number of elements that were available when the stream was created. It is unaffected by any read operations that might have been performed on the stream.

For output and input-output streams, this is the number of elements that were available when the stream was created (just as with input streams), added to the number of elements written past the end of the stream (regardless of any repositioning operations).

It is assumed that:

- There is no more than one stream open on the same source or destination at a time.
- There are no shared references to files by other processes.
- There are no alias references to underlying sequences, or any other such situations.

In such situations, the behavior of stream-size is undefined.

**<string-stream>**

Open Instantiable Class

The class of streams over strings.

**Superclasses**

<sequence-stream>

**Init-Keywords**

- contents – A general instance of <sequence>.
- direction – Specifies the direction of the stream. It must be one of "input", "output", or "input-output"; Default value: "input".
- start – An instance of <integer>. Only valid when direction: is "input". Default value: 0.
- end – An instance of <integer>. This specifies the string position immediately after the portion of the string to stream over. Only valid when direction: is "input". Default value: contents.size.
Discussion

The class of streams over strings.

The contents: init-keyword is used as the input for an input stream, and as the initial storage for an output stream.

The start: init-keyword specifies the start position of the string to be streamed over.

The class supports the same init-keywords as <sequence-stream>.

The start: and end: init-keywords specify the portion of the string to create the stream over: start: is inclusive and end: is exclusive. The default is to stream over the entire string.

Operations

• make(<string-stream>)

See also

• make(<string-stream>)

• <sequence-stream>

synchronize-output Open Generic function
Synchronizes an output stream with the application state.

Signature synchronize-output output-stream => ()

Parameters

• output-stream – An instance of <stream>.

Discussion

Forces any pending output from output-stream’s buffers to its destination. Before returning to its caller, synchronize-output also attempts to ensure that the output reaches the stream’s destination before, thereby synchronizing the output destination with the application state.

When creating new stream classes it may be necessary to add a method to the synchronize-output function, even though it is not part of the Stream Extension Protocol.

See also

• force-output

type-for-file-stream Open Generic function
Finds the type of file-stream class that needs to be instantiated for a given file.

Signature type-for-file-stream filename element-type #rest #all-keys => file-stream-type

Parameters

• filename – An instance of <object>.

• element-type – One of <byte-character>, <unicode-character>, or <byte>, or #f.

Values

• file-stream-type – An instance of <type>.

Discussion Returns the kind of file-stream class to instantiate for a given file. The method for make(<file-stream>) calls this function to determine the class of which it should create an instance.

See also
• <file-stream>
• make(<file-stream>)

**type-for-sequence-stream** Open Generic function

Finds the type of sequence-stream class that needs to be instantiated for a given sequence.

**Signature**

\[ \text{type-for-sequence-stream} \text{ sequence } \Rightarrow \text{sequence-stream-type} \]

**Parameters**

• **sequence** – An instance of `<sequence>`.

**Values**

• **sequence-stream-type** – An instance of `<type>`.

**Discussion**

Returns the sequence-stream class to instantiate over a given sequence object. The method for `make(<sequence-stream>)` calls this function to determine the concrete subclass of `<sequence-stream>` that it should instantiate.

There are `type-for-sequence-stream` methods for each of the string object classes. These methods return a stream class object that the Streams module considers appropriate.

See also

• `make(<sequence-stream>)`
• `<sequence-stream>`

**<unicode-character>** Type

**Type** The type that represents Unicode characters.

**Supertypes** `<character>`

**Discussion** A type representing Unicode characters that instances of `<unicode-string>` can contain.

**Operations**

• `type-for-file-stream`

**<unicode-string-stream>** Open Instantiable Class

The class of streams over Unicode strings.

**Superclasses** `<string-stream>`

**Init-Keywords**

• **contents** – A general instance of `<sequence>`.

• **direction** – Specifies the direction of the stream. It must be one of "input", "output", or "input-output". Default value: "input".

• **start** – An instance of `<integer>`. This specifies the start position of the sequence to be streamed over. Only valid when `direction` is "input". Default value: 0.

• **end** – An instance of `<integer>`. This specifies the sequence position immediately after the portion of the sequence to stream over. Only valid when `direction` is "input". Default value: `contents.size`.

**Discussion**

The class of streams over Unicode strings. It is a subclass of `<string-stream>`.
The contents: init-keyword is used as the input for an input stream, and as the initial storage for an output stream. If it is a stretchy vector, then it is the only storage used by the stream.

The class supports the same init-keywords as `<sequence-stream>`.

The start: and end: init-keywords specify the portion of the Unicode string to create the stream over: start: is inclusive and end: is exclusive. The default is to stream over the entire Unicode string.

Operations

- `make(<unicode-string-stream>)`

See also

- `make(<unicode-string-stream>)`
- `<sequence-stream>`

**unlock-stream** Open Generic function

Unlocks a stream.

**Signature** unlock-stream stream

**Parameters**

- `stream` – An instance of `<stream>`.

**Discussion**

Unlocks a stream. It is suggested that `with-stream-locked` be used instead of direct usages of `lock-stream` and `unlock-stream`.

See *Locking streams* for more detail and discussion on using streams from multiple threads.

See also

- `lock-stream`
- `stream-lock`
- `stream-lock-setter`
- `with-stream-locked`

**unread-element** Open Generic function

Returns an element that has been read back to a positionable stream.

**Signature** unread-element positionable-stream element => element

**Parameters**

- `positionable-stream` – An instance of `<positionable-stream>`.
- `element` – An instance of `<object>`.

**Values**

- `element` – An instance of `<object>`.

**Discussion**

“Unreads” the last element from `positionable-stream`. That is, it returns `element` to the stream so that the next call to `read-element` will return `element`. The stream must be a `<positionable-stream>`.

It is an error to do any of the following:
• To apply unread-element to an element that is not the element most recently read from the stream.
• To call unread-element twice in succession.
• To unread an element if the stream is at its initial position.
• To unread an element after explicitly setting the stream’s position.

See also
• read-element

wait-for-io-completion Generic function
Waits for all pending operations on a stream to complete.

Signature wait-for-io-completion file-stream => ()

Parameters
• file-stream – An instance of <stream>.

Discussion If file-stream is asynchronous, waits for all pending write or close operations to complete and signals any queued errors. If file-stream is not asynchronous, returns immediately.

with-indentation Statement Macro

Macro Call

with-indentation (*stream*)
  *body*
end;

with-indentation (*stream*, 4)
  *body*
end;

Parameters
• stream – A Dylan expression bnf. An instance of <indenting-stream>.
• indentation – A Dylan expression bnf. An instance of <integer>. Optional.
• body – A Dylan body bnf.

Discussion Runs a body of code with an indentation level managed automatically. This avoids the need to call indent to indent and then unindent around the body of code.

Example

with-indentation(is, 4)
  format(is, "Hello %="
  end with-indentation;

See also
• <indenting-stream>
• indent

with-open-file Statement Macro
Runs a body of code within the context of a file stream.

Macro Call
Parameters

- **stream-var** – An Dylan variable-name bnf.
- **filename** – An instance of `<string>`.
- **keys** – Instances of `<object>`.
- **body** – A Dylan body bnf.

Values

- **values** – Instances of `<object>`.

Discussion

Provides a safe mechanism for working with file streams. The macro creates a file stream and binds it to `stream-var`, evaluates a `body` of code within the context of this binding, and then closes the stream. The macro calls `close` upon exiting `body`.

The values of the last expression in `body` are returned.

Any `keys` are passed to the `make` method on `<file-stream>`.

Example  The following expression yields the contents of file `foo.text` as a `<byte-vector>`:

```dylan
with-open-file (fs = "foo.text", element-type: <byte>)
  read-to-end(fs)
end;
```

It is roughly equivalent to:

```dylan
begin
  let hidden-fs = #f; // In case the user bashes fs variable
  block ()
    hidden-fs := make(<file-stream>,
                     locator: "foo.text", element-type: <byte>);
    let fs = hidden-fs;
    read-to-end(fs);
  cleanup
    if (hidden-fs) close(hidden-fs) end;
  end block;
end;
```

See also

- `close(<file-stream>)`
- `<file-stream>`
- `make(<file-stream>)`

**with-stream-locked** Statement Macro

Run a body of code while the stream is locked.

Macro Call

```dylan
with-stream-locked (*stream-var*)
  *body* end => *values*
```
Parameters

- **stream-var** – An Dylan variable-name `bnf`.
- **body** – A Dylan body `bnf`.

Values

- **values** – Instances of `<object>`.

Discussion

Provides a safe mechanism for locking streams for use from multiple threads. The macro evaluates a `body` of code after locking the stream, and then unlocks the stream. The macro calls `unlock-stream` upon exiting `body`.

The values of the last expression in `body` are returned.

See also

- `lock-stream`
- `stream-lock`
- `stream-lock-setter`
- `unlock-stream`

 `<wrapper-stream>` Open Instantiable Class

The class of wrapper-streams.

**Superclasses** `<stream>`

**Init-Keywords**

- **inner-stream** – An instance of `<stream>`.

Discussion

The class that implements the basic wrapper-stream functionality.

It takes a required init-keyword `inner-stream`, which is used to specify the wrapped stream.

The `<wrapper-stream>` class implements default methods for all of the stream protocol functions described in this document. Each default method on `<wrapper-stream>` simply “trampolines” to its inner stream.

Operations

- `inner-stream`
- `inner-stream-setter`
- `outer-stream-setter`

Example

In the example below, `<io-wrapper-stream>`, a subclass of `<wrapper-stream>`, "passes on" functions such as `read-element` and `write-element` by simply delegating these operations to the inner stream:

```dylan
define method read-element (ws :: <io-wrapper-stream>,
  #key on-end-of-stream)
=> (element)
  read-element(ws.inner-stream)
end method;

define method write-element (ws :: <io-wrapper-stream>,
  (continues on next page))
```
write-element(ws.inner-stream, element)
end method;

Assuming that <io-wrapper-stream> delegates all other operations to its inner stream, the following is sufficient to implement a 16-bit Unicode character stream wrapping an 8-bit character stream.

define class <unicode-stream> (<io-wrapper-stream>)
end class;

define method read-element (s :: <unicode-stream>,
    #key on-end-of-stream)
=> (ch :: <unicode-character>)
    let first-char = read-element(s.inner-stream,
        on-end-of-stream);
    let second-char = read-element(s.inner-stream,
        on-end-of-stream);
    convert-byte-pair-to-unicode(first-char, second-char)
end method;

define method write-element (s :: <unicode-stream>,
    c :: <character>)
=> ()
    let (first-char, second-char)
        = convert-unicode-to-byte-pair(c);
    write-element(s.inner-stream, first-char);
    write-element(s.inner-stream, second-char)
    c
end method;

define method stream-position (s :: <unicode-stream>)
=> (p :: <integer>)
    truncate/(stream-position(s.inner-stream), 2)
end method;

define method stream-position-setter (p :: <integer>,
    s :: <unicode-stream>);
    stream-position(s.inner-stream) := p * 2
end method;

**write**

Generic function

Writes the elements of a sequence to an output stream.

**Signature**  
write output-stream sequence #key start end => ()

**Parameters**

- output-stream – An instance of <stream>.
- sequence – An instance of <sequence>.
- start (#key) – An instance of <integer>. Default value: 0.
- end (#key) – An instance of <integer>. Default value: sequence.size.

**Discussion**

Writes the elements of sequence to output-stream, starting at the stream’s current position.
The elements in sequence are accessed in the order defined by the forward iteration protocol on <sequence>. This is effectively the same as the following:

```
do (method (elt) write-element(stream, elt)
   end, sequence);
```

If supplied, start and end delimit the portion of sequence to write to the stream. The value of start is inclusive and that of end is exclusive.

If the stream is positionable, and it is not positioned at its end, write overwrites elements in the stream and then advances the stream’s position to be beyond the last element written.

**Implementation Note:** Buffered streams are intended to provide a very efficient implementation of write, particularly when sequence is an instance of <byte-string>, <unicode-string>, <byte-vector>, or <buffer>, and the stream’s element type is the same as the element type of sequence.

**Example** The following forms bind stream to an output stream over an empty string and create the string “I see!”, using the function stream-contents to access all of the stream’s elements.

```
let stream = make(<byte-string-stream>,
   direction: #"output");
write-element(stream, 'I');
write-element(stream, ' ');  
write(stream, "see");
write-element(stream, '!');
stream-contents(stream);
```

See also

- read
- write-element
- write-line

**write-element** Open Generic function

Writes an element to an output stream.

**Signature** write-element output-stream element => ()

**Parameters**

- output-stream – An instance of <stream>.
- element – An instance of <object>.

**Discussion**

Writes element to output-stream at the stream’s current position. The output-stream must be either #"output" or #"input-output". It is an error if the type of element is inappropriate for the stream’s underlying aggregate.

If the stream is positionable, and it is not positioned at its end, write-element overwrites the element at the current position and then advances the stream position.

**Example** The following forms bind stream to an output stream over an empty string and create the string “I do”, using the function stream-contents to access all of the stream’s elements.

```
let stream = make(<byte-string-stream>,
   direction: #"output");
```

(continues on next page)
write-element(stream, 'I');
write-element(stream, 'd');
write-element(stream, 'o');
stream-contents(stream);

See also

- read-element
- write
- write-line

write-line Open Generic function

Writes a string followed by a newline to an output stream.

Signature  write-line output-stream string #key start end => ()

Parameters

- output-stream – An instance of <stream>.
- string – An instance of <string>.
- start (#key) – An instance of <integer>. Default value: 0.
- end (#key) – An instance of <integer>. Default value: string.size.

Discussion

Writes string followed by a newline sequence to output-stream.

The default method behaves as though it calls write on string and then calls new-line.

If supplied, start and end delimit the portion of string to write to the stream.

See also

- new-line
- read-line
- write
- write-element

10.5 The standard-io Module

This document describes the Standard-IO module, which requires the Streams module. All interfaces described in this document are exported from the Standard-IO module. The functionality provided by this module mirrors some of the functionality provided by the Java.io package in Java.

For convenience, the Standard-IO module, together with the Format module, is repackaged by the Format-out module. See The format-out Module for details.

10.5.1 Handling standard input and output

The Standard-IO module provides a Dylan interface to the standard I/O facility of operating systems such as MS-DOS or UNIX.
The module consists of three variables, each of which is bound to a stream:

- *standard-input*
- *standard-output*
- *standard-error*

For console-based applications (i.e., applications that run in character mode), the three streams just use the console window in which the application was started.

For purely window-based applications, each variable is bound by default to a stream that lazily creates a console window as soon as any input is requested or output is performed. Only one window is created, and this is shared between all three streams. Any subsequent input or output uses the same window. The window that is created uses the standard configuration settings set by the user. For example, the window is only scrollable if all console windows are configured to be scrollable on the machine running the application.

For more information about streams, please refer to The streams Module.

10.5.2 The standard-io module

This section contains a complete reference of all the interfaces that are exported from the standard-io module.

*standard-input* Variable

The standard input stream.

Type <stream>

Value The standard input stream for the platform on which the application is running.

Discussion

This variable is bound to an input stream that reads data from the standard input location for the platform on which the application is running. It is equivalent to the Java stream java.lang.System.in.

If the platform has a notion of standard streams, such as MS-DOS, this stream maps onto the platform-specific standard input stream. If the platform has no such convention, such as a platform that is primarily window-based, then a console window is created for this stream if necessary, in order to provide users with a place to provide input.

If a console window has already been created as a result of writing to one of the other variables in the Standard-IO module, then the existing console window is used, and a new one is not created: a single console window is used for all variables in this module.

*standard-output* Variable

The standard output stream.

Type <stream>

Value The standard output stream for the platform on which the application is running.

Discussion

This variable is bound to an output stream that sends normal output to the standard output location for the platform on which the application is running. It is equivalent to the Java stream java.lang.System.out.

If the platform has a notion of standard streams, such as MS-DOS, this stream maps onto the platform-specific standard output stream. If the platform has no such convention, such as a platform that is primarily window-based, a console window is created for this stream if necessary, just to capture output to it.
If a console window has already been created as a result of writing to or reading from one of the
other variables in the Standard-IO module, then the existing console window is used, and a new
one is not created: a single console window is used for all variables in this module.

*standard-error* Variable
The standard error stream.

Type  <stream>

Value  The standard error stream for the platform on which the application is running.

Discussion
This variable is bound to an output stream that sends error messages to the standard error lo-
cation for the platform on which the application is running. It is equivalent to the Java stream
java.lang.System.err.

If the platform has a notion of standard streams, such as MS-DOS, this stream maps onto the
platform-specific standard error stream. If the platform has no such convention, such as a plat-
form that is primarily window-based, a console window is created for this stream if necessary,
just to capture output to it.

If a console window has already been created as a result of writing to or reading from one of the
other variables in the Standard-IO module, then the existing console window is used, and a new
one is not created: a single console window is used for all variables in this module.
11.1 Overview

The logging library exports a single module named “logging”.

11.2 Quick Start

Log to stdout:

```define constant $log
   = make(<log>,
          name: "my-app",
          formatter: "%{millis} %{level} [%{thread}] - %{message}";
   add-target($log, $stdout-log-target);
   log-info($log, "My-app starting with args %s", application-arguments());```

The above results in log lines like this:

```
12345 INFO [Main Thread] - My-app starting with args blah```

Make another log for debugging server requests:
Define constant $request-log
= make(log, name: "my-app.debug.request");

There are several things to notice about the above setup:

- Logs have no log targets by default. The simplest way to add a target is to add a pre-existing target such as
  $stdout-log-target or $stderr-log-target using add-target.
- Different logs are associated by name. In this example the log named "my-app" is an ancestor of the one
  named "my-app.debug.request" because the first dotted name component matches.
- No targets were added to the my-app.debug.request log. Since all log messages sent to a child are
  also sent to its ancestors (but see log-additive?-setter), anything logged to the my-app.debug.
  request log will be passed along to the my-app log.

So what's the benefit of having both logs? You can enable/disable them separately at runtime. Also, if for
example you wanted to log debug messages to a separate file you could add a target to the my-app.debug
log.

Log to a file:

add-target($log, make(rolling-file-log-target,
  pathname: "/var/log/my-app.log");

The log file will be rolled immediately if it exists and is not zero length. If you don’t want it to be rolled on startup,
pass roll: #f to make in the above call.

Logs may be disabled with log-enabled?(log) := #f. When disabled, no messages will be logged to the
log’s local targets, but the value of log-additive? will still be respected. In other words, logging to a disabled log
will still log to ancestor logs if they are themselves enabled.

### 11.3 Errors

If there is an error when parsing a log-formatter format control string or in finding a log object by name,
a logging-error will be signaled.

<logging-error> Open Class

    Superclasses <error>, <simple-condition>

### 11.4 Log Levels

There are five log levels which may be used to affect the way logs are formatted and to include/exclude logs of different
severity levels. When configuring logging, set the log level to the least severe level you want to see. “Trace” logs are
the least severe (or most verbose). “Error” logs are the most severe. The distinctions are somewhat arbitrary, but it is
hoped that five levels is enough for even the most compulsive taxonomists.

<log-level> Open Primary Abstract Class

    Each of the log level constants documented below is an instance of this class.

    Superclasses <object>

    Init-Keywords

    - name – The name used to display this log level. For example, “INFO”, “DEBUG”, etc.
$trace-level Constant
The most verbose log level. Generally use this to generate an absurd amount of debug output that you would
never want generated by (for example) a production server.

$debug-level Constant
For debug messages. Usually for messages that are expected to be temporary, while debugging a particular
problem.

$info-level Constant
For messages about relatively important events in the normal operation of a program.

$warn-level Constant
For out-of-the-ordinary events that may warrant extra attention, but don't indicate an error.

$error-level Constant
For errors.

level-name Generic function
Signature level-name (level) => (name)
Parameters
• level – An instance of <log-level>.
Values
• name – An instance of <string>.

11.5 Logging Functions

log-message Generic function
Signature log-message (level log object #rest args) => ()
This is the most basic logging function. All of the logging functions below simply call this with a specific
$log-level> object.
Parameters
• level – An instance of <log-level>.
• log – An instance of <log>.
• object – An instance of <object>. Normally this is a format control string, but it is also possible (for example) to log objects to a database back-end.
• args (#rest) – Instances of <object>. These are normally format arguments to be interpolated into the above format control string.

log-error Function
Equivalent log-message($log-error, ...)
See log-message.

log-warning Function
Equivalent log-message($log-warn, ...)
See log-message.

log-info Function
Equivalent log-message($log-info, ...)

11.5. Logging Functions
log-debug Function

Equivalent log-message($log-debug, ...)

See log-message.

log-debug-if Function

Signature log-debug-if (test log object #rest args) => ()

Equivalent

```dylan
if (test)
    log-message($log-debug, ...)
end
```

See log-message.

log-trace Function

Equivalent log-message($log-trace, ...)

See log-message.

log-level-applicable? Generic function

Signature log-level-applicable? (given-level log-level) => (applicable?)

Parameters

- `given-level` – An instance of `<log-level>`.
- `log-level` – An instance of `<log-level>`.

Values

- `applicable?` – An instance of `<boolean>`.

11.6 Logs

<abstract-log> Abstract Class

Superclasses `<object>`

Init-Keywords

- `name` – *(required)* The dotted name of this log. A `<string>`.
- `additive?` – A `<boolean>` specifying whether log messages sent to this log should be passed along to its parent log. The default is `#t`.
- `children` – A `<sequence>` of `<log>` objects.
- `enabled?` – A `<boolean>` specifying whether this log is enabled. Note that the value of `additive?` will be respected even if the log is disabled. The default is `#t`.
- `parent` – The parent of this log.

<log> Open Class

Superclasses `<abstract-log>`

Init-Keywords
- **formatter** – An instance of `<log-formatter>`.
- **level** – An instance of `<log-level>`.
- **targets** – A collection of `<log-target>` objects, each of which receives log messages sent to this log.

### get-log Generic function

**Signature**

```dylan
def get-log (name) => (abstract-log or #f)
```

**Parameters**

- **name** – An instance of `<string>`. This is normally a dotted path name like “http.server.queries”.

**Values**

- **log** – An instance of `<abstract-log>` or #f.

### get-root-log Generic function

**Signature**

```dylan
def get-root-log () => (log)
```

**Values**

- **log** – An instance of `<log>`.

### log-level Generic function

**Signature**

```dylan
def log-level (log) => (level)
```

**Parameters**

- **log** – An instance of `<log>`.

**Values**

- **level** – An instance of `<log-level>`.

### log-level-setter Generic function

**Signature**

```dylan
def log-level-setter (new-level log) => (new-level)
```

**Parameters**

- **new-value** – An instance of `<log-level>`.
- **log** – An instance of `<log>`.

**Values**

- **new-value** – An instance of `<log-level>`.

### log-targets Generic function

**Signature**

```dylan
def log-targets (log) => (targets)
```

**Parameters**

- **log** – An instance of `<log>`.

**Values**

- **targets** – An instance of `<stretchy-vector>`.

### log-additive? Generic function

**Signature**

```dylan
def log-additive? (log) => (additive?)
```

**Parameters**

- **log** – An instance of `<log>`.

---

11.6. Logs
• log – An instance of `<log>`.

Values
• additive? – An instance of `<boolean>`.

log-additive?-setter Generic function
Signature log-additive?-setter (new-value log) => (new-value)
Parameters
• new-value – An instance of `<boolean>`.
• log – An instance of `<log>`.

Values
• new-value – An instance of `<boolean>`.

log-enabled? Generic function
Signature log-enabled? (log) => (enabled?)
Parameters
• log – An instance of `<log>`.

Values
• enabled? – An instance of `<boolean>`.

log-enabled?-setter Generic function
Signature log-enabled?-setter (new-value log) => (new-value)
Parameters
• new-value – An instance of `<boolean>`.
• log – An instance of `<log>`.

Values
• new-value – An instance of `<boolean>`.

log-name Generic function
Signature log-name (log) => (name)
Parameters
• log – An instance of `<log>`.

Values
• name – An instance of `<string>`.

add-target Generic function
Signature add-target (log target) => ()
Parameters
• log – An instance of `<log>`.
• target – An instance of `<log-target>`.

remove-all-targets Generic function
Signature remove-all-targets (log) => ()
Parameters

• log – An instance of \(<log>\).

**remove-target Generic function**

**Signature** remove-target (log target) => ()

**Parameters**

• log – An instance of \(<log>\).

  • target – An instance of \(<log-target>\).

**log-formatter Generic function**

**Signature** log-formatter (log) => (formatter)

**Parameters**

• log – An instance of \(<log>\).

**Values**

• formatter – An instance of \(<log-formatter>\).

**log-formatter-setter Generic function**

**Signature** log-formatter-setter (formatter log) => (formatter)

**Parameters**

• formatter – An instance of \(<log-formatter>\).

  • log – An instance of \(<log>\).

**Values**

• formatter – An instance of \(<log-formatter>\).

### 11.7 Log Targets

**<log-target> Open Abstract Class**

**Superclasses** \(<closable-object>\)

**<null-log-target> Class**

**Superclasses** \(<log-target>\)

A log target that discards all messages.

**<file-log-target> Class**

**Superclasses** \(<log-target>\)

**Init-Keywords**

• pathname – *(required)* An instance of \(<pathname>\).

A log target that logs to a single, monolithic file. You probably want \(<rolling-file-log-target>\) instead.

**target-pathname Generic function**

**Signature** target-pathname (file-log-target) => (pathname)

**Parameters**
• **target** – An instance of `<file-log-target>`.

Values

• **pathname** – An instance of `<pathname>`.

**open-target-stream**

Open Generic function

This should not be called except by the logging library itself. Implementers of new log target classes may override it.

*Signature*  
`open-target-stream (target) => (stream)`

*Parameters*

• **target** – An instance of `<file-log-target>`.

*Values*

• **stream** – An instance of `<stream>`.

**<rolling-file-log-target> Class**

Superclasses  `<file-log-target>`

Init-Keywords

• **max-size** – An `<integer>`. The size in bytes at which to roll the file. The default size is 100MB. Note that the actual size of the file when it rolls may be slightly larger, depending on the size of the last message logged.

• **roll** – A `<boolean>` specifying whether to roll the log file at the time this log target is created, if it already exists and is not empty.

**<stream-log-target> Open Class**

A log target that sends all messages to a stream.

Superclasses  `<log-target>`

Init-Keywords

• **stream** – *(required)* An instance of `<stream>`.

**target-stream**

Generic function

*Signature*  
`target-stream (target) => (stream)`

*Parameters*

• **target** – An instance of `<stream-log-target>`.

*Values*

• **stream** – An instance of `<stream>`.

**log-to-target**

Open Generic function

This should not be called except by the logging library itself. Implementers of new log target classes may override it.

*Signature*  
`log-to-target (target level formatter object args) => ()`

*Parameters*

• **target** – An instance of `<log-target>`.

• **level** – An instance of `<log-level>`.

• **formatter** – An instance of `<log-formatter>`.

• **object** – An instance of `<object>`.
• **args** – An instance of `<sequence>`.

**write-message** Open Generic function

This should not be called except by the logging library itself. Implementers of new log target classes may override it.

**Signature**

```dylan
write-message (target object args) => ()
```

**Parameters**

• **target** – An instance of `<log-target>`.

• **object** – An instance of `<object>`.

• **args** – An instance of `<sequence>`.

**$null-log-target** Constant

An predefined instance of `<null-log-target>`.

**$stderr-log-target** Constant

An predefined instance of `<stream-log-target>` that sends log messages to `*standard-error*`.

**$stdout-log-target** Constant

An predefined instance of `<stream-log-target>` that sends log messages to `*standard-output*`.

### 11.8 Log Formatting

Each `<log>` has a `<log-formatter>` that determines how to format each log message. Make one like this:

```dylan
make(<log-formatter>, pattern: "...");
```

The log formatter pattern is similar to a format control string except it has a short and long form for each format directive. Here are the defined format directives:

<table>
<thead>
<tr>
<th>Short</th>
<th>Long</th>
<th>Description</th>
</tr>
</thead>
</table>
| %d    | %{date:fmt} | Current date. In the long form, fmt is any string acceptable as the first argument to `format-date`.
| %l    | %{level} | Log level. e.g., INFO, DEBUG, ERROR, etc.
| %m    | %{message} | Log message, as passed to log-info, log-debug etc., with format arguments already interpolated.
| %p    | %{pid} | Current process ID. (Not yet implemented.)
| %r    | %{millis} | Milliseconds since application started.
| %t    | %{thread} | Current thread name.
| %%    | None | The % character.

All format directives, in either short or long form, accept a numeric argument immediately following the % character. If provided, the numeric argument specifies the minimum width of the field. If the numeric argument is positive then the displayed value will be left justified and padded with spaces on the right if necessary. If negative, the displayed value will be right justified and padded with spaces on the left if needed.

**$default-log-formatter** Constant

Formatter used if none is specified when a `<log>` is created. Has this pattern:

```dylan
"%{date:%Y-%m-%dT%H:%M:%S.%f%fz} %-5L [%t] %m"
```

**<log-formatter>** Open Class

**Superclasses** `<object>`
Init-Keywords

- `pattern` – An instance of `<string>`. 
12.1 Overview

The Network library provides internet address protocols and TCP/IP server and client sockets. It exports a single module, called Sockets.

12.2 Utilities

This section covers the `start-sockets` function, which all libraries using the Network library must call before any other call to the Network library API. It also covers the `with-socket-thread` macro which registers the current thread as a thread that will call a socket function that blocks.

**start-sockets Function**

*Signature*  
`start-sockets () => ()`

*Discussion*

Applications must call this function before using any other function or variable from the Network library.

This function is necessary because the Win32 API library Winsock2, which the Network library calls to get native socket functionality, requires applications to call an initialization function before calling any Winsock2 API functions. The call to `start-sockets` calls this underlying Winsock2 function.

Note that you must not call `start-sockets` from a top-level form in any DLL project. The combination of this, and the restriction that you must call `start-sockets` before calling anything else, implies that no Network library function or variable can be called (directly or indirectly) from a top-level form in any DLL project. Instead, the DLL project should define a start function that calls `start-sockets` (directly or indirectly) or re-export `start-sockets` so that their clients can arrange to have it called from a top-level form in an appropriate EXE project.

Applications using the Network library must arrange for `start-sockets` to be called (directly or indirectly) before any other sockets API functions. A good place to do this is at the beginning of your start function (usually the `main` method). For example:

```define method main () => ();
    start-sockets();
    let the-server = make(<TCP-server-socket>, port: 7);
    ...
end;```

(continues on next page)
New start functions that call `start-sockets` and that are defined for DLL projects that use the Network library will inherit all of the restrictions described above for `start-sockets`.

Calling a Network library function before calling `start-sockets` results in a `<sockets-not-initialized>` error. Calling `start-sockets` from a top-level form in a DLL project will result in unpredictable failures—probably access violations during initialization.

`with-socket-thread Statement Macro`

**Macro Call**

```plaintext
begin
  with-socket-thread (#key *server?*)
  *body*
end;
```

**Discussion**

Registers the current thread as a blocking socket thread, that is, a thread which will call a socket function that blocks, such as `read-element` or `accept`.

The reason for the registration is that Network library shutdown can then synchronize with these threads. The early part of the shutdown sequence should cause the threads to unblock with an `<end-of-stream-error>` so that they can do whatever application cleanup is necessary. Once these threads have exited, the rest of the shutdown sequence can be executed.

A server socket thread (blocking on `accept` rather than `read-element`) notices that the shutdown sequence is underway slightly later, with a `<blocking-call-interrupted>` condition.

### 12.3 Internet addresses

This section covers Internet address protocols.

#### 12.3.1 Basic Internet address protocol

This section covers the class `<internet-address>` and related generic functions and constants.

**<internet-address> Open Primary Abstract Instantiable Class**

**Superclasses** `<object>`

**Init-Keywords**

- `name` – An instance of `<string>` representing a symbolic internet address.
- `address` – An instance of `<string>` representing a presentation (dotted) form Internet address or an instance of `<numeric-address>` (see below).

**Discussion**
The class of objects representing Internet addresses used as endpoints for peer-to-peer socket connections.

To construct an `<internet-address>` object you must supply either the `name:` or `address:` keyword. For example:

```
make (<internet-address>, name: "www.whatever.com")
```

or

```
make (<internet-address>, address: "9.74.122.0")
```

`make on <internet-address>` returns an instance of `<ipv4-address>`.

**host-name Open Generic function**

**Signature**  
`host-name internet-address => name`

**Discussion**

Returns an instance of `<string>` containing a symbolic host name. The `internet-address` argument must be an instance of `<internet-address>`.

Usually the name returned is the canonical host name. Note, however, that the implementation is conservative about making DNS calls. Suppose that the `<internet-address>` instance was created with the `name:` keyword and no other information. If the application has not made any other requests that would require a DNS call, such as to `host-address` or `aliases`, the name that this function returns will be the one specified with the `name:` keyword, regardless of whether that is the canonical name or not.

**host-address Open Generic function**

**Signature**  
`host-address internet-address => address`

**Discussion**

Returns an instance of `<string>` containing the presentation form of the host address. In the case of multi-homed hosts this will usually be the same as:

```
multi-homed-internet-address.all-addresses.first.host-address
```

In the case of an Internet address created using the `address:` keyword it will be either the keyword value or `all-addresses.first.host-address`.

**numeric-host-address Open Generic function**

Returns the host address as a `<numeric-address>`.

**Signature**  
`numeric-host-address internet-address => numeric-address`

**all-addresses Open Generic function**

**Signature**  
`all-addresses internet-address => sequence`

**Discussion**

Returns an instance of `<sequence>` whose elements are `<internet-address>` objects containing all known addresses for the host.

**aliases Open Generic function**

**Signature**  
`aliases internet-address => sequence`

**Discussion**

Returns an instance of `<sequence>` whose elements are instances of `<string>` representing alternative names for the host.

**local-host-name Function**

12.3. Internet addresses
Signature  local-host-name () => name
Discussion  Returns an instance of <string> containing a symbolic host name.

12.3.2 The <ipv6-address> class

This name is reserved for future development.

12.3.3 The <numeric-address> class

This section describes numeric Internet representation and associated protocols.

<nemonic-address> Primary Abstract Sealed Class

Superclasses  <object>
Discussion  The class of objects representing the numeric form of an Internet addresses.

Currently only ipv4 (32-bit) addresses are supported. Ipv6 addresses will be added when they are supported by Winsock2. In general <numeric-address> objects are accessed using the functions host-order or network-order, depending on the context in which they are employed.

network-order Sealed Generic function

Signature  network-order address => network-order-address
Discussion  Returns the value of the numeric address in network order. The argument is a general instance of <numeric-address>. The class of the object returned depends upon the particular subclass of the argument; the network-order method for <ipv4-numeric-address> returns an instance of <machine-word>.

Network order is big-endian byte order.

host-order Sealed Generic function

Signature  host-order address => host-order-address
Discussion  Like network-order but returns the value in host order.

Host order is either big-endian byte order on a big-endian host machine and little-endian on a little-endian host machine.

12.3.4 IPV4 addresses

<ipv4-numeric-address> Open Primary Abstract Instantiable Class

Superclasses  <numeric-address>
Init-Keywords

• value – An instance of <machine-word>. Required.
• order – One of "network-order" or "host-order". Required.
Discussion The single slot of this class contains a 32-bit value representing a ipv4 address. This slot is accessed by the generic functions network-order and host-order described above. <ipv4-numeric-address> has two concrete subclasses <ipv4-network-order-address> and <ipv4-host-order-address>. Make <ipv4-numeric-address> returns one or the other of these depending upon the value of the order: keyword.

host-order(<ipv4-numeric-address>) Method

- **Signature** host-order ipv4-numeric-address => machine-word
- **Discussion** Returns the numeric address in host order as an instance of <machine-word>. The argument is an instance of <ipv4-numeric-address>.

network-order(<ipv4-numeric-address>) Method

- **Signature** network-order ipv4-numeric-address => machine-word
- **Discussion** Returns the numeric address in network order as an instance of <machine-word>. The argument is an instance of <ipv4-numeric-address>.

as(<string>, <ipv4-numeric-address>) Method

Returns the presentation (dotted string) form of an instance of <ipv4-numeric-address>.

- **Signature** as string ipv4-numeric-address => string

<ipv4-network-order-address> Concrete Sealed Class

Concrete subclass for network-order numeric addresses.

- **Superclasses** <ipv4-numeric-address>
- **Discussion**

```dylan
make(<ipv4-network-order-address>)
```

is equivalent to

```dylan
make(<ipv4-numeric-address>, order: network-order)
```

<ipv4-host-order-address> Concrete Sealed Class

Concrete subclass for host order numeric addresses.

- **Superclasses** <ipv4-numeric-address>

### 12.4 Sockets

This section describes socket classes and protocols.

#### 12.4.1 The <abstract-socket> class

<abstract-socket> Open Free Abstract Uninstantiable Class

- **Superclasses** <object>
- **Init-Keywords**
  - `socket-descriptor` – A Windows handle or UNIX fd (file descriptor) for the socket. In general users of the sockets API should not need to use this keyword. Only implementors of new socket classes should be interested.
Discussion  The common superclass of all socket objects including `<socket>` (IP client socket), `<server-socket>` and `<socket-accessor>`.

Each subclass of `<abstract-socket>` must provide methods for close and for the following generic functions:

**local-port** Open Generic function

Returns the local port number.

*Signature*  local-port socket => port-number

*Parameters*

- socket – An instance of `<socket>`, `<datagram-socket>` or `<server-socket>`.

*Values*

- port-number – An instance of `<integer>`.

**socket-descriptor** Open Generic function

Returns the descriptor (handle or fd) for the socket.

*Signature*  socket-descriptor socket => descriptor

*Parameters*

- socket – An instance of `<abstract-socket>`.

*Values*

- descriptor – An instance of `<accessor-socket-descriptor>`.

**local-host** Open Generic function

Returns the address of the local host.

*Signature*  local-host socket => host-address

*Parameters*

- socket – An instance of `<abstract-socket>`.

*Values*

- host-address – An instance of `<internet-address>`.

### 12.4.2 The `<server-socket>` class

**<server-socket>** Open Primary Abstract Instantiable Class

*Superclasses* `<abstract-socket>`

*Init-Keywords*

- **service** – An instance of `<string>` containing an abstract name for a service with a "well-known" port, such as "ftp" or "daytime". Valid names depend on the configuration of the DNS. Required unless port: is supplied.

- **port** – An instance of `<integer>` identifying the port on which the `<server-socket>` should listen for connection requests. Required unless service: is supplied.

- **protocol** – An instance of `<string>` naming the protocol. Currently "tcp" is the only supported protocol. You can create instances of protocol-specific subclasses as an alternative to using the protocol: keyword. For example, make(<server-socket>,

`...`
protocol: "tcp", ...) is equivalent to make(<tcp-server-socket>, ...).

**Discussion**
Server-sockets listen on a specified port for connection requests which come in over the network. Either the `port:` or `service:` keyword must be supplied.

*make* on (<server-socket>) returns an instance of <tcp-server-socket> by default.

### accept Open Generic function

**Signature** accept server-socket #rest args #key => result

**Discussion** Blocks until a connect request is received, then it returns a connected instance of <socket>. The particular subclass of <socket> returned depends on the actual class of the argument, which must be a general instance of <server-socket>. Calling accept on <tcp-server-socket> returns a connected <tcp-socket>. The keyword arguments are passed to the creation of the <socket> instance. For UDP sockets accept returns immediately with an instance of <udp-socket>. No blocking happens for UDP sockets because they are connectionless. After reading from a UDP socket returned from accept the socket can be interrogated for the location of the sender using `remote-host` and `remote-port`.

### with-server-socket Macro

**Macro Call**

```dylan
with-server-socket (*server-var* :: *server-class* [, *keywords*)
  *body*
end;
```

**Discussion** Creates an instance of <server-socket>, using the (optional) `server-class` argument and keyword arguments to make the <server-socket>, and binds it to the local variable named by `server-var`. The *body* is evaluated in the context of the binding and the <server-socket> is closed after the body is executed.

### start-server Macro

**Macro Call**

```dylan
start-server ([*server-var* = ]<socket-server-instance>,
  *socket-var* [, *keywords*])
  *body*
end;
```

**Discussion** Enters an infinite while(#t) accept loop on the server socket. Each time accept succeeds the <socket> returned from accept is bound to `socket-var` and the *body* is evaluated in the context of the binding. When *body* exits, accept is called again producing a new binding for `socket-var`. The optional keywords are passed to the call to accept.

### 12.4.3 The <tcp-server-socket> class

**<tcp-server-socket> Class**

- **Superclasses** <server-socket>
- **Init-Keywords**
Dylan Library Reference, Release 1.0

• **element-type** – Establishes a new default for the *element-type* of `<TCP-socket>` instances returned by calling `accept` with this server socket as the argument to `accept`. This default *element-type* may be overridden for any particular call to `accept` by using the `element-type:` keyword to `accept`. If no `element-type:` is specified when the server socket is created, `<byte-character>` is used as the default *element-type*.

**Discussion** The class of TCP server sockets. A server socket is an object which listens for requests for connections from the network. When accept is called on the server socket and a request for connection is detected, accept returns a connected `<socket>`.

### accept (<tcp-server-socket>) Method

**Signature** accept server-socket #rest args #key element-type => connected-socket

**Parameters**

- **server-socket** – An instance of `<tcp-server-socket>`.
- **element-type (#key)** – Controls the element type of the `<tcp-socket>` (stream) returned. If not supplied, defaults to `#f`.

**Values**

- **connected-socket** – A connected instance of `<tcp-socket>`.

**Discussion** The other keyword arguments are passed directly to the creation of the `<tcp-socket>` instance.

#### 12.4.4 The `<socket>` class

**<socket>** Open Free Abstract Instantiable Class

The class of general client sockets. All client sockets are streams.

**Superclasses** `<abstract-socket>, <external-stream>`

**Init-Keywords**

- **direction** – Specifies the direction of the stream. It must be one of `"input", "output", and "input-output"`. This keyword is an inherited streams class keyword. See the Streams library documentation in the System and I/O library reference for a full description.
- **element-type** – An instance of `<class>`. Useful values are `<byte-character>` and `<byte>`. This keyword is an inherited streams class keyword. See the Streams library documentation in the System and I/O library reference for a full description.

#### 12.4.5 The `<buffered-socket>` class

**<buffered-socket>** Class

**Superclasses** `<socket>, <double-buffered-stream>`

**Init-Keywords**

- **force-output-before-read?** – An instance of `<boolean>`. Defaults value: `#t`. The methods which implement the stream reading protocols (read, read-line, read-element and so on) for instances of `<socket>` call `force-output` by default before blocking. This is to ensure that any pending output has been sent to the peer before the socket blocks waiting to read data sent by the peer. This corresponds to the expected,
usual behavior of single-threaded client sockets and avoids deadlock in usual cases. Multi-threaded applications, particularly applications where one thread is reading and another thread is writing to the same socket, may wish to inhibit the default force-output. If the socket is created with force-output-before-read?: as #f, force-output will not be called before the read functions block.

Discussion Socket streams whose elements are bytes or characters. These inherit buffering protocols and the implications of read, write, read-element, write-element, force-output and suchlike methods from <double-buffered-stream>.

12.4.6 The <tcp-socket> class

The class of TCP client sockets.

<tcp-socket> Class

The class of TCP client sockets.

Superclasses <buffered-socket>

Init-Keywords

• host – An instance of <internet-address> or <string>. The remote host to connect to. The <string> may be either a host name or a presentation-form Internet address. Required.

• service – An instance of <string>. A <string> containing an abstract name for a service with a “well-known” port, such as "ftp" or "daytime". Valid names depend on the configuration of the DNS. Required unless port: is supplied.

• protocol – An instance of <string> naming the protocol. Currently "tcp" and "udp" are the only supported protocols. You can create instances of protocol-specific subclasses as an alternative to using the protocol: keyword. For example make(<socket>, protocol: "tcp", ...) is equivalent to make(<TCP-socket>, ..., protocol: "tcp", ...). make on <socket> returns an instance of <tcp-socket> by default.

• port – An instance of <integer> representing the remote port to connect to. Required unless service: is supplied.

• element-type – An instance of <class>. Useful values for <tcp-streams> are <byte-character> and <byte>. This keyword is an inherited streams class keyword. See The streams Module for a full description.

remote-port Open Generic function

Returns the remote port number for a <socket>.

Signature remote-port socket => port-number

Parameters

• socket – An instance of <socket>.

Values

• port-number – An instance of <integer>.

remote-host Open Generic function

Returns the remote host for a <socket>.

Signature remote-host socket => remote-host-address

Parameters


• **socket** – An instance of `<socket>`.

**Values**

• **remote-host-address** – An instance of `<internet-address>`.

### 12.4.7 The `<udp-socket>` class

The class of UDP client sockets.

**<udp-socket> Class**

The class of UDP client sockets.

**Superclasses** `<buffered-socket>`

**Init-KeyWords**

• **host** – An instance of `<internet-address>` or `<string>`. The remote host to connect to. The `<string>` may be either a host name or a *presentation-form* Internet address. Required.

• **service** – An instance of `<string>`. A `<string>` containing an abstract name for a service with a “well-known port”, such as "ftp" or "daytime". Valid names depend on the configuration of the DNS. Required unless `port:` is supplied.

• **protocol** – An instance of `<string>` naming the protocol. Currently "tcp" and "udp" are the only supported protocols. You can create instances of protocol-specific subclasses as an alternative to using the `protocol:` keyword. For example `make(<socket>, protocol: "udp", ...)` is equivalent to `make(<UDP-socket>, ...)`. `make` on `<socket>` returns an instance of `<tcp-socket>` by default.

• **port** – An instance of `<integer>` representing the remote port to connect to. Required unless `service:` is supplied.

• **element-type** – An instance of `<class>`. Useful values for `<udp-socket>`s are `<byte-character>` and `<byte>`. This keyword is an inherited streams class keyword. See *The streams Module* for a full description.

**Discussion** Of the keywords, `host:` and one of either `service:` or `port:` are required.

### 12.4.8 The `<udp-server-socket>` class

The class of UDP server sockets.

**<udp-server-socket> Class**

**Superclasses** `<server-socket>`

**Init-KeyWords**

• **element-type** – Establishes a new default for the element-type of `<UDP-socket>` instances returned by calling `accept` with this server socket as the argument to `accept`. This default element-type may be overridden for any particular call to `accept` by using the `element-type:` keyword to `accept`. If no `element-type:` is specified when the server socket is created, `<byte-character>` is used as the default element-type.

**Discussion** The class of UDP server sockets. A server socket is an object that listens for requests from the network. When `accept` is called on the UDP server socket, `accept` returns a `<udp-socket>`. 
12.5 Socket conditions

This section lists the socket condition classes in the Network library.

**<socket-condition> Class**

All socket conditions are general instances of `<socket-condition>`. Some are recoverable and others are not.

**Superclasses** `<simple-condition>`

**Discussion**

The class of socket conditions. It inherits the `format-string:` and `format-arguments:` keywords from `<simple-condition>`.

**Slots:**

- `socket-condition-details`

  - Most socket conditions originate in error return codes from Open Dylan’s Winsock2 library, an FFI interface to the native socket library Winsock2.
  - The `socket-condition-details` slot provides information about the low-level failure which was the source for the condition. In most cases this slot will hold an instance of `<socket-accessor-condition>`, below.
  - When creating general instances of `<socket-condition>`, you can use the `details:` keyword to set the value for this slot.

**<socket-error> Class**

The class `<socket-error>` is the superclass of all unrecoverable socket conditions.

**Superclasses** `<socket-condition>`

**Discussion**

The class of socket conditions from which no recovery is possible.

**<internal-socket-error> Class**

The class `<internal-socket-error>` is the class of unexpected socket errors.

**Superclasses** `<socket-error>`

**Discussion**

The class of unexpected errors from Open Dylan’s Winsock2 library, an FFI interface to the native socket library Winsock2.

Inspect the contents of the `socket-condition-details` slot for more information.

**<recoverable-socket-condition> Class**

The `<recoverable-socket-condition>` class is the general class of socket conditions for which an application may be able to take some remedial action.

**Superclasses** `<socket-condition>`

**Discussion**

The general class of socket conditions for which an application may be able to take some remedial action.

For instance, a web browser receiving such conditions as `<connection-refused>` or `<host-not-found>` would normally expect to report those conditions to the user and continue with some other connection request from the user, while a server receiving a `<connection-closed>` condition from a connected `<socket>` would probably close the `<socket>` and continue to handle other requests for connections.
The network — probably a local network — is down. Try again later.

Superclasses <recoverable-socket-condition>

A badly formed address string has been passed to a function trying to make an <internet-address>.

Superclasses <recoverable-socket-condition>

The Domain Name Server (DNS) cannot resolve the named host or internet address. Try again with a different (correct) name or address.

Superclasses <recoverable-socket-condition>

The Domain Name Server (DNS) did not respond or returned an ambiguous result. Try again.

Superclasses <recoverable-socket-condition>

The remote host cannot be reached from this host at this time.

Superclasses <recoverable-socket-condition>

The socket or server socket has been closed.

Most operations on closed instances of <tcp-socket> return instances of <stream-closed-error> (from the Streams library) rather than instances of <socket-closed>.

The attempt to connect to the remote host was not successful. Connection failed for one of the following reasons: because the connect request timed out or because it was refused, or because the remote host could not be reached.

The connection to the remote host has been broken. The socket should be closed. To try again, open a new socket.

A process on the machine is already bound to the same fully qualified address. This condition probably occurred because you were trying to use a port with an active server already installed, or a process crashed without closing a socket.

A blocking socket call, like read, write or accept, was interrupted.

The attempt to connect to the remote host was not successful. Connection failed for one of the following reasons: because the connect request timed out or because it was refused, or because the remote host could not be reached.

The connection to the remote host has been broken. The socket should be closed. To try again, open a new socket.

A process on the machine is already bound to the same fully qualified address. This condition probably occurred because you were trying to use a port with an active server already installed, or a process crashed without closing a socket.
Superclasses  <recoverable-socket-condition>

Discussion The implementation-dependent limit on the number of open sockets has been reached. You must close some sockets before you can open any more. The limits for Windows NT (non-server machines) and Windows 95 are particularly small.

<socket-accessor-error> Class

Superclasses  <socket-error>

Discussion An implementation-specific error from the C-FFI interface to the native socket library. Usually instances of this class these appear in the socket-condition-details slot of another <socket-condition>.

<win32-socket-error> Class

Superclasses  <socket-accessor-error>

Discussion A Win32-specific error from the Winsock2 library, a C-FFI interface to the native socket library Winsock2. A function in the FFI library has returned an error return code.

Slots:

WSA-numeric-error-code  Contains the numeric error code that was returned. An instance of <integer>.

WSA-symbolic-error-code  Contains an instance of <string> giving the symbolic (human-readable) form of the error code. For example, the string might be “wsanotsock”.

explanation  An explanation if any of the error. An instance of <string>.

calling-function  The name of Winsock2 FFI interface function which returned the error code. An instance of <string>.
CHAPTER
THIRTEEN

THE PROGRESS-STREAM LIBRARY

Many text-mode programs that perform long-running, non-interactive tasks (such as compilers) provide a running indication of what fraction of the task is complete using a bar graph drawn using text characters. This library implements a wrapper stream that interleaves the progress bar graph with any status output. When output is not directed to a console then progress updates are ignored.

<progress-stream> Abstract Instantiable Class

Superclasses <wrapper-stream>

Init-Keywords

• force? – If true, forces the wrapped stream to be treated as a console, allowing textual progress bar output.

Keyword bar-width: An instance of <integer> representing the number of bar characters to be included in textual progress bar output. Defaults to 40.

Keyword line-width: An instance of <integer> representing the total line width (including both progress bar and label) in characters. Defaults to 79.

show-progress Generic function

Signature show-progress (stream position range #key label) => ()

Parameters

• stream – An instance of <progress-stream>.
• position – An instance of <integer>.
• range – An instance of <integer>.
• label (#key) – An instance of <object>.

Discussion Graphically displays the current progress, proportional to position divided by range, to the given stream. In the case where stream is a console (i.e., unredirected *standard-output*), show-progress will draw a textual progress bar with overall width configured in the <progress-stream> instance.

stream-supports-show-progress? Generic function

Signature stream-supports-show-progress? (stream) => (supports-show-progress?)

Parameters

• stream – An instance of <stream>.

Values

• supports-show-progress? – An instance of <boolean>.
**Discussion** Returns a true value if the `stream` argument can usefully support the `show-progress` operation.
14.1 Overview

The regular-expressions library exports the regular-expressions module, which contains functions that compile and search for regular expressions. The module has the same semantics as Perl (version 4) unless otherwise noted.

A regular expression that is grammatically correct may still be illegal if it contains an infinitely quantified sub-regex that matches the empty string. That is, if R is a regex that can match the empty string, then any regex containing R*, R+, and R{n,} is illegal. In this case, the regular-expressions library will signal an <invalid-regex> error when the regex is parsed.

14.2 Quick Start

The most common use of regular expressions is probably to perform a search and figure out what text matched and/or where it occurred in a string. You need to use regular-expressions; in both your library and your module, and then...

```commandline
define constant $re :: <regex> = compile-regex("^abc(.+)+123$");

let match :: false-or(<regex-match>) = regex-search($re, "abcdef123");
// match is #f if search failed.

if (match)
    let text = match-group(match, 1);
    // text = "def"

    let (text, start, _end) = match-group(match, 1);
    // text = "def", start = 3, _end = 6

    match-group(match, 2) => error: <invalid-match-group>
    // group 0 is the entire match
    ...
end;

compile-regex("*") => error: <invalid-regex>
```
14.3 Reference

<regex> Sealed Class
A compiled regular expression object. These should only be created via compile-regex.

<regex-error> Sealed Class
The superclass of all regular expression-related errors.

    Superclasses <format-string-condition>,<error>

<invalid-regex> Sealed Class
Signalled by compile-regex when the given regular expression cannot be compiled.

    Superclasses <regex-error>

    Init-Keywords
        • pattern –

regex-error-pattern Sealed Generic function
Return the pattern that caused an <invalid-regex> error.

    Signature regex-error-pattern error => pattern

    Parameters
        • error – An <invalid-regex>.

    Values
        • pattern – A <string>.

<invalid-match-group> Sealed Class
Signalled when an invalid group identifier is passed to match-group.

    Superclasses <regex-error>

<regex-match> Sealed Class
Stores the match groups and other information about a specific regex search result.

    Superclasses <object>

    Init-Keywords
        • regular-expression –

compile-regex Sealed Generic function
Compile a string into a <regex>.

    Signature compile-regex pattern #key case-sensitive verbose multi-line dot-matches-all use-cache => regex

    Parameters
        • pattern – A <string>.
        • case-sensitive(#key) – A <boolean>, default #t.
        • verbose(#key) – A <boolean>, default #f.
        • multi-line (#key) – A <boolean>, default #f.
        • dot-matches-all (#key) – A <boolean>, default #f.
        • use-cache(#key) – A <boolean>, default #t. If true, the resulting regular expression
          will be cached and re-used the next time the same string is compiled.
Values

• `regex` – A `<regex>`.

Conditions `<invalid-regex>` is signalled if `pattern` can’t be compiled.

**regex-pattern** Sealed Generic function

Return the `<string>` from which `regex` was created.

**Signature** `regex-pattern regex => pattern`

**Parameters**

• `regex` – A `<regex>`.

**Values**

• `pattern` – A `<string>`.

**regex-group-count** Sealed Generic function

Return the number of groups in a `<regex>`.

**Signature** `regex-group-count regex => num-groups`

**Parameters**

• `regex` – A `<regex>`.

**Values**

• `num-groups` – An `<integer>`.

**regex-position** Sealed Generic function

Find the position of `pattern` in `text`.

**Signature** `regex-position pattern text #key start end case-sensitive => regex-start, #rest marks`

**Parameters**

• `pattern` – A `<regex>`.

• `text` – A `<string>`.

• `start (#key)` – A `<integer>`, default 0. The index at which to start the search.

• `end (#key)` – An `<integer>`, default `*text*.size`. The index at which to end the search.

• `case-sensitive (#key)` – A `<boolean>`, default `#t`.

**Values**

• `regex-start` – An instance of `false-or(<integer>)`.

• `#rest marks` – An instance of `<object>`.

A match will only be found if it fits entirely within the range specified by `start` and `end`.

If the regular expression is not found, return `#f`, otherwise return a variable number of indices marking the start and end of groups.

This is a low-level API. Use `regex-search` if you want to get a `<regex-match>` object back.

**regex-replace** Sealed Generic function

Replace occurrences of `pattern` within `big` with `replacement`.

**Signature** `regex-replace big pattern replacement #key start end count case-sensitive => new-string`

**Parameters**
• **big** – The `<string>` within which to search.
• **pattern** – The `<regex>` to search for.
• **replacement** – The `<string>` to replace `pattern` with.
• **start (#key)** – An `<integer>`, default 0. The index in `big` at which to start searching.
• **end (#key)** – An `<integer>`, default `*big*.size`. The index at which to end the search.
• **case-sensitive (#key)** – A `<boolean>`, default `#t`.
• **count (#key)** – An instance of `false-or(<integer>)`, default `#f`. The number of matches to replace. `#f` means to replace all.

**Values**
• **new-string** – An instance of `<string>`.

A match will only be found if it fits entirely within the range specified by `start` and `end`.

**regex-search Sealed Generic function**

Search for a `pattern` within `text`.

**Signature**  
`regex-search pattern text #key anchored start end case-sensitive => match`

**Parameters**
• **pattern** – The `<regex>` to search for.
• **text** – The `<string>` in which to search.
• **anchored (#key)** – A `<boolean>`, default `#f`. Whether or not the search should be anchored at the start position. This is useful because “^...” will only match at the beginning of a string, or after n if the regex was compiled with multi-line = `#t`.
• **start (#key)** – An `<integer>`, default 0. The index in `text` at which to start searching.
• **end (#key)** – An `<integer>`, default `*text*.size`. The index at which to end the search.
• **case-sensitive (#key)** – A `<boolean>`, default `#t`.

**Values**
• **match** – An instance of `false-or(<regex-match>)`. `#f` is returned if no match was found.

A match will only be found if it fits entirely within the range specified by `start` and `end`.

**regex-search-strings Sealed Generic function**

Find all matches for a regular expression within a string.

**Signature**  
`regex-search-strings pattern text #key anchored start end case-sensitive => #rest strings`

**Parameters**
• **pattern** – An instance of `<regex>`.
• **text** – An instance of `<string>`.
• **anchored (#key)** – An instance of `<boolean>`.
• **start (#key)** – An `<integer>`, default 0. The index in `text` at which to start searching.
• **end (#key)** – An `<integer>`, default `*text*.size`. The index at which to end the search.
- **case-sensitive** (#key) – A `<boolean>`, default `#t`.

**Values**
- **#rest strings** – An instance of `<object>`.
  A match will only be found if it fits entirely within the range specified by `start` and `end`.

**match-group** Sealed Generic function
Return information about a specific match group in a `<regex-match>`.

**Signature** `match-group match group => text start-index end-index`

**Parameters**
- **match** – An instance of `<regex-match>`.
- **group** – An instance of `<string>` or `<integer>`.

**Values**
- **text** – An instance of `false-or(<string>)`.
- **start-index** – An instance of `false-or(<integer>)`.
- **end-index** – An instance of `false-or(<integer>)`.

**Conditions** `<invalid-match-group>` is signalled if `group` does not name a valid group.

The requested group may be an `<integer>` to access groups by number, or a `<string>` to access groups by name. Accessing groups by name only works if they were given names in the compiled regular expression via the `(?<foo>...)` syntax.

Group 0 is always the entire regular expression match.

It is possible for the group identifier to be valid and for `#f` to be returned. This can happen, for example, if the group was in the part of an `|` (or) expression that didn’t match.
Open Dylan’s SQL-ODBC library provides a generic Dylan protocol for interfacing applications to any database management system (DBMS) supporting Microsoft’s Open Database ConnectivityTM (ODBC) interface and the SQL database query language. The SQL-OBDC library supports the full SQL language defined in the ANSI SQL-89 and ANSI SQL-92 specifications, as well as any extensions defined by a DBMS.

A low-level interface to the Microsoft ODBC API is also available in the ODBC-FFI library. The ODBC-FFI library uses the C-FFI library and the same C-to-Dylan name mapping scheme as described in the Win32 API FFI library documentation. See the C FFI and Win32 library reference for details of that scheme. The ODBC-FFI library is otherwise undocumented.
15.1 Implementation

The SQL-ODBC library is built on top of a generic SQL library. This SQL library does not include the low-level code necessary to communicate with any particular DBMS. In itself, the SQL library simply provides a convenient high-level mechanism for integrating database operations into Dylan applications. It is designed to form the high-level part of “implementation libraries” that contain lower-level code to supporting a particular DBMS protocol, such as ODBC. The SQL-ODBC library is, then, one such implementation library.

Our intention is that the SQL library will provide a common high-level Dylan interface to any DBMS. Applications written using the SQL-ODBC library will therefore be simple to port to any future DBMSes for which implementation libraries are written.

15.2 Using the SQL-ODBC library in applications

The SQL-ODBC library is available to applications as the SQL-ODBC library, which exports the modules SQL-ODBC and SQL. (You should not need to use the SQL module, but it will be visible during debugging sessions.)

15.3 Object-oriented languages and relational databases

The SQL-ODBC library does not provide the means to “objectify” a relational database or an SQL statement. That is, it does not treat a databases or statements in an object-oriented fashion or provide support for doing so.

This is because the object-oriented programming model is very different from the relational database model. The two most significant differences follow.

First, the relational database model has only simple datatypes (string, integer, floating point number, and so on) and does not provide a means of defining new types, as object-oriented programming languages do.

Second, objects in an object-oriented program have unique identities that allow two objects of the same value to be distinguished from one another. By contrast, SQL rows (the SQL notion nearest to the notion of an object) do not have unique identities: if two rows in a given table have identical values, they are indistinguishable.

15.4 Result-retrieval protocol

The SQL-ODBC library provides an abstract Dylan protocol for handling SQL result sets, the means by which SQL SELECT statement results are retrieved. The library allows result sets to be processed as Dylan collections. The various Dylan collection protocols and functions work as you would expect on a result set.
15.5 Processing results

SQL SELECT statements return database records. You process the results of an SQL SELECT statement using a result set. Result sets are the focal point of the SQL-ODBC library’s encapsulation of the protocol for retrieving database records. Using result sets allows you to concentrate on the logic of your application instead of the logic of record retrieval.

Result sets retrieve their records from the database synchronously. As result sets retrieve their records, you can specify conversion of records to application-specific objects which are added to the result set in place of the record. Result sets retrieve their records one at a time.

15.6 Bridging the object-relational gap

Relational DBMSes do not in general deal with objects or classes. Since Dylan is an object-oriented language, this creates a gap between Dylan and the DBMS.

The SQL-ODBC library bridges this gap by allowing you to specify a liaison function for results. A liaison function acts as an interpreter for results, taking the records retrieved from the relational DBMS and converting each into suitable Dylan objects. A default liaison method exists for use in situations where your application does not know the appropriate conversion, for example when processing SQL SELECT statements typed in by the application user. The default method transforms each record retrieved into a Dylan collection, where each element of the collection corresponds to a column of the record. See Section 1.5.4 on page 36 for more on liaison functions.

15.7 Error handling

As in any application, errors at run time can occur when applications talk to databases. The SQL-ODBC library captures the errors and warnings that a DBMS generates and signals a corresponding Dylan error or warning condition. Your application can then process the condition using the Dylan condition system.

15.8 Examples used in this document

The following tables depict example database tables to which this document’s code examples refer.

<table>
<thead>
<tr>
<th>Table 1: Table 1.1 Table “Book” used in this document’s code examples.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Title</strong></td>
</tr>
<tr>
<td>An Introduction to Database Systems</td>
</tr>
<tr>
<td>Transaction Processing: Concepts and Techniques</td>
</tr>
<tr>
<td>Fundamentals of Database Systems</td>
</tr>
<tr>
<td>Relational Database Writings, 1991-1994</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 2: Table 1.2 Table “Author” used in this document’s code examples.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Author ID</strong></td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
</tbody>
</table>
Table 3: Table 1.3 Table “Book_author” used in this document’s code examples.

<table>
<thead>
<tr>
<th>Author_ID</th>
<th>ISBN</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0-201-14201-5</td>
</tr>
<tr>
<td>2</td>
<td>1-55860-190-2</td>
</tr>
<tr>
<td>3</td>
<td>1-55860-190-2</td>
</tr>
<tr>
<td>4</td>
<td>0-8053-1748-1</td>
</tr>
<tr>
<td>5</td>
<td>0-8053-1748-1</td>
</tr>
<tr>
<td>1</td>
<td>0-201-82459-0</td>
</tr>
</tbody>
</table>

15.8.1 Connecting to a database

Before it can query a database, your application must connect to it. Most DBMSes operate a form of login procedure to verify connections, using a user name and password for the purpose. The popular DBMSes each have different protocols for identifying themselves, their users, their databases, and connections to those databases.

The SQL-ODBC library provides a general-purpose connection protocol that is not specific to any DBMS, and represents DBMSes, databases, database connections, user names and passwords with generic Dylan classes, thereby hiding the idiosyncrasies of the various DBMSes from Dylan applications. The classes that the SQL-ODBC library defines are shown in Table 1.4.

Table 4: Table 1.4 Dylan DBMS classes.

<table>
<thead>
<tr>
<th>Entity</th>
<th>Abstract Dylan class</th>
<th>SQL-ODBC class</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBMS</td>
<td>&lt;dbms&gt;</td>
<td>&lt;odbc-dbms&gt;</td>
</tr>
<tr>
<td>Database</td>
<td>&lt;database&gt;</td>
<td>&lt;odbc-database&gt;</td>
</tr>
<tr>
<td>User name and password</td>
<td>&lt;user&gt;</td>
<td>&lt;odbc-user&gt;</td>
</tr>
<tr>
<td>Active connection</td>
<td>&lt;connection&gt;</td>
<td>&lt;odbc-connection&gt;</td>
</tr>
</tbody>
</table>

You should create DBMS-specific instances of these classes to connect to a database.

See also with-database.

15.9 Connection protocol functions, methods, and macros

- with-dbms
- dbms
- database
- user

15.10 Connecting and disconnecting

The SQL-ODBC library provides DBMS-independent functions to connect to and disconnect from databases. Connecting to a database establishes a context (an instance of <connection>) in which SQL statements may be executed within an application. You can make connections by calling the connect function on a DBMS-specific instance of <database> and <user>.

An application can connect to multiple databases served by a DBMS if the DBMS supports the feature. Multiple-connection support can be determined by calling the multiple-connections? function on the DBMS object.
Keeping connections open requires system resources. An application can disconnect from connections that it no longer needs in order to reduce its use of system resources. When the application terminates, the SQL-ODBC library disconnects all open connections. If a connection is not explicitly terminated using the disconnect generic function, and a client application has no references to it, the connection is terminated when the garbage collector notices that the object can be reclaimed. After a connection has been disconnected, the <connection> object cannot be reused, and so references to it should be dropped.

15.10.1 Executing SQL statements

The SQL-ODBC library provides a way of processing SQL statements: the execute function, which you must apply to instances of the <sql-statement> class.

15.11 The null value

SQL offers the null value to represent missing information, or information that is not applicable in a particular context. All columns of a table can accept the null value – unless prohibited by integrity constraints – regardless of the domain of the column. Hence, the null value is included in all domains of a relational database and can be viewed as an out-of-band value.

Relational database theory adopted a three-valued logic system – “true”, “false”, and “null” (or “unknown”) – in order to process expressions involving the null value. This system has interesting (and sometimes frustrating) consequences when evaluating arithmetic and comparison expressions. If an operand of an arithmetic expression is the null value,
the expression evaluates to the null value. If a comparand of a comparison expression is the null value, the expression may evaluate to the null/unknown truth-value.

For example:

- $a + b$, where $a$ contains the null value or $b$ contains the null value, evaluates to the null value
- $a + b$, where $a$ contains the null value and $b$ contains the null value, evaluates to the null value
- $a = b$, where $a$ contains the null value or $b$ contains the null value, evaluates to unknown
- $a = b$, where $a$ contains the null value and $b$ contains the null value, evaluates to unknown
- $a \lor b$, where $a$ is true and $b$ contains the null value, evaluates to true
- $a \land b$, where $a$ is false and $b$ contains the null value, evaluates to false

The SQL `SELECT` statements return records for which the `WHERE` clause (or `WHERE` predicate) evaluates to true (not to false and not to the null value). In order to test for the presence or absence of the null value, SQL provides a special predicate of the form:

```
column-name is [not] null
```

The null value is effectively a universal value that is difficult to use efficiently in Dylan. To identify when null values are returned from or need to be sent to a DBMS server, the SQL-ODBC library supports indicator objects. Indicator objects indicate when a column of a record retrieved from a database contains the null value, or when a client application wishes to set a column to the null value.

- `<null-value>`
- `null-value`

### 15.12 Input indicators and output indicators

It is difficult for database applications written in traditional programming languages to represent the semantics of the null value, because it is a universal value which is in the domain of all types, and the three-valued logic system which accompanies null values does not easily translate to the two-value logic system in traditional programming languages.

In Dylan, a universal value can be achieved if we ignore type specialization, but this inhibits optimization and method dispatching. Even if we were to forgo type specialization, the evaluation of arithmetic and comparison expressions is a problem since Dylan’s logic system is boolean and not three-valued. Therefore, the SQL-ODBC library has a goal of identifying null values and translating them into Dylan values that can be recognized as representing null values.

In order to identify null values during SQL statement processing, the `<sql-statement>` class supports an input indicator and output indicator. An input indicator is a marker value or values which identifies an input host variable as containing the null value. An output indicator is a substitution value which semantically identifies columns of a retrieved record as containing the null value.

If the SQL-ODBC library encounters a null value when retrieving records from a database, and there is no appropriate indicator object, it signals a `<data-exception>` condition. The condition is signaled from result-set functions (including the collection protocol) and not the execute function.

During the execution of an SQL statement to which an input indicator value was supplied, each input host variable is compared (with the function `\==`) to the input indicator and, if it holds the input indicator value, the null value is substituted for it.

The input indicator may be a single value or a sequence of values. A single value is useful when it is in the domain of all input host variables; if the host variables have not been specialized, any newly created value will do. Otherwise, a sequence of values must be used. Input indicators that are general instances of `<sequence>` use their positional occurrence within the SQL statement as the key for the sequence.
The SQL SELECT statement is the only SQL statement that returns non-status results back to the client application. During the retrieval of these results, the SQL-ODBC library substitutes the output indicator, if supplied, for null values found in the columns of each record.

The output indicator may be a single value or a sequence of values. If the output indicator is a general instance of `<sequence>`, the element of the sequence whose key corresponds to the column index is used as the substitution value. Otherwise, the output indicator value itself is used as the substitution value.

### 15.12.1 Data retrieval using result-set collection

Executing an SQL SELECT statement by invoking the execute function on the instance of `<sql-statement>` that represents the statement yields a result set.

A result set is a Dylan collection which encapsulates the protocol necessary to retrieve data from a database. The SQL-ODBC library defines two subclasses of `<result-set>` that provide different behaviors and performance characteristics. The type of the result set returned by the execute function is determined by the result-set policy supplied to the function or macro.

There are two subclasses of `<result-set>`: `<forward-only-result-set>` and `<scrollable-result-set>`.

The `<forward-only-result-set>` class provides an efficient means of accessing the elements of a result set. Efficiency is achieved by performing minimal processing on each record retrieved and by maintaining in memory only the current record. Implicit in this behavior is that records you have accessed previously are no longer available to your application; if you maintain references to previous records behavior is unpredictable. The key for each access must always be greater than or equal to the previous access’s key; otherwise, a condition is signaled.

The `<scrollable-result-set>` class allows your application to access elements of the result-set collection in any order, meaning that records you have accessed previously can be revisited. Scrollable result sets retrieve records synchronously.

Example:

This example returns a list of authors who have published two or more books.

```dylan
let query = make(<sql-statement>,
  text: "select last_name, first_name, count(*)
  from author, book_author
  where book_author.author_id = author.author_id
  group by last_name, first_name
  having count(*) > 2"
);
execute(query, result-set-policy: $scrollable-result-set-policy);
```

### 15.12.2 Result-set collections

A result-set collection, in spirit, contains the result of an SQL SELECT statement. To provide these results, result-set collections and their methods control the retrieval of elements from the database. Each element of a result set is a
record and each element of a record is a value. The SQL-ODBC library does not provide any classes to represent columns; the elements of a record are just Dylan objects.

Result-set classes, in conjunction with the methods defined on them, provide a protocol to retrieve data from a database. Result-sets do not necessarily contain the records from the database. A result set could cache a small subset of the records retrieved for performance reasons. The logic for retrieving a record from a result set (from the database) is as follows:

1. Perform an internal fetch: values are stored into bindings established during SQL statement preparation. A record object is created during the preparation of the SQL statement which represents the values of the record (collection of values).
2. Invoke the liaison method on the record object. The result of the liaison function is the result of the collection access.

The columns of a record are processed when the columns are retrieved from the record object. This includes checking for null values and performing data coercion if a coercion-policy is supplied.

15.13 Record class

An instance of the \texttt{<record>} class is a placeholder for records retrieved from the database. The record class is a collection whose elements are the columns of the records retrieved from the database. If the record object has a coercion policy (obtained through the result-set-policy), datatype coercion is performed on the elements of the record object as they are retrieved from the collection.

The elements of a record collection are ephemeral under the result-set retrieval protocol: the values for the elements of the collection can change when the next record of the result set is accessed. A result set may maintain more than one record object to improve performance.

Record collections support the forward- and backward-iteration protocols. The result of calling \texttt{type-for-copy} on the \texttt{<record>} class is \texttt{<simple-object-vector>}. Applications cannot instantiate the \texttt{<record>} class. However, the functions returned by the forward- and backward-iteration protocol methods on the result-set classes return instances of this class.

The values in a record object have a short lifespan: they are only valid until the next fetch is performed.

See also:

- \texttt{<coercion-policy>}
- \texttt{<record>}

15.14 Result-set policy class

Applications use result-set policy classes to specify the behavior and performance characteristics of a result set, as well as its type. The type of the result set is determined by the result-set policy object. The type of the record object is determined by the \texttt{coercion-policy} slot of \texttt{<sql-statement>}. If \texttt{result-set-policy.scrollable?} is \texttt{#t}, the result set will be an instance of \texttt{<scrollable-result-set>} otherwise it will be an instance of \texttt{<forward-only-result-set>}. If \texttt{statement.coercion-policy} \texttt{~=} \texttt{$no-coercion} then the record will be an instance of \texttt{<coercion-record>}; otherwise, it will be an instance of \texttt{<record>}.
Table 5: Table 1.5 Result set policies and classes.

<table>
<thead>
<tr>
<th>Scrollable?</th>
<th>Coercion policy</th>
<th>Result set class</th>
</tr>
</thead>
<tbody>
<tr>
<td>#f</td>
<td>#f</td>
<td>&lt;forward-only-result-set&gt;</td>
</tr>
<tr>
<td>#t</td>
<td>•</td>
<td>&lt;scrollable-result-set&gt;</td>
</tr>
</tbody>
</table>

See also:

- <result-set-policy>

## 15.15 Result-set classes

Result-sets are the focal point for the encapsulation of the protocol required to retrieve records from a database. The SQL-ODBC library provides three result-set classes with different performance and behavioral characteristics. These classes are <result-set>, <forward-only-result-set>, and <scrollable-result-set>.

## 15.16 Liaison functions

Liaison functions convert records retrieved from a database query to Dylan objects. These functions bridge the conceptual gap between relational databases and Dylan’s object-orientation.

To create a Dylan object from a retrieved record, the liaison function must understand the form of the records coming from the database and the mappings of records to Dylan objects. These Dylan objects make up the elements of the result set: the results of the liaison function are added to the result set each time it is called. As your application iterates over a result set, the liaison function provides the objects that the application processes.

If you do not provide a liaison function for a result set, the SQL-ODBC library supplies a default-liaison function to perform the conversion. If a coercion policy is provided, the default-liaison function is copy-sequence. The new sequence is safe in that it is a normal Dylan collection with no relationship to databases, SQL statements, or result sets. If a coercion policy is not provided, the default-liaison is the identity function.

You can specify the identity function as the liaison function to process the actual record objects. If no type coercion is performed by the functions on the record class, this function will have the lowest overhead, but there are some restrictions: the values retrieved from the record may become invalid when the state of the iteration protocol changes.

The liaison function can, potentially, cause the greatest number of problems for an application using SQL-ODBC since there is no type safety between the liaison function, the record class and the SQL SELECT statement. You must ensure that the liaison function is in sync with the SQL SELECT statement since there is no support in SQL-ODBC for this.

Example:

```dylan
define class <book> {<object>}
  slot title :: <string>, init-keyword: title:;
  slot publisher :: <string>, init-keyword: publisher:;
  slot isbn :: <string>, init-keyword: isbn:;
  slot author :: <string>, init-keyword: author:;
end class;

begin
  let booker =
    method (record :: <record>) => (book :: <book>)
      let (title, publisher, isbn, last_name, first_name) =
```
apply(values, record);

make(<book>, title: title, publisher: publisher,
     isbn: isbn, author: concatenate(last_name, "", first_name));
end method;

let query = make(<sql-statement>,
                statement: "select title, publisher, isbn,
                           last_name, first_name
                   from book, author, book_author
                   where book.isbn = book_author.isbn
                   and book_author.author_id =
                       author.author_id
                   order by author.last_name,
                           author.first_name");

execute(query, liaison: booker
          result-set-policy: make(<forward-only-result-set-policy>));
end;

15.17 Coercion policies

In the SQL-ODBC library, the element method on the record class encapsulates all coercions of data retrieved from
a database. This method can return columns with or without coercion: as low-level SQL data-types (no conversion),
as Dylan data-types, or as user-defined types. The coercion-policy: init-keyword of the <sql-statement>
class determines this behavior.

If the coercion-policy: init-keyword is $no-coercion, coercions are not performed. Hence, your appli-
cation will be processing objects with low-level SQL datatypes. This option has the lowest overhead but the most
restrictions: the values returned from the element method may not be valid (values may change as storage may be
reused) after the next call to the next-state method returned by forward-iteration-protocol.

The value of $default-coercion for the coercion-policy: init-keyword (the default value) indicates that
default coercion should be performed: the data retrieved from the database is coerced to the corresponding Dylan
objects.

A sequence for the coercion-policy: init-keyword instructs the SQL library to perform specific data coercion
on the data retrieved from the database. Essentially, each element of the limited sequence is a data coercion function
which will be invoked using the objects returned from the database as the argument.

When there is a one-to-one correspondence between an SQL datatype and a built-in or user-defined Dylan datatype,
use the <record> class to perform the conversion. When multiple columns define a Dylan object or one column
defines multiple Dylan objects, use the liaison function to perform the conversion.

15.17.1 Data types and conversions

The datatypes that relational DBMSes use are different from those Dylan uses. The SQL-ODBC library provides
classes that represent these low-level relational datatypes, along with a table that defines the mapping from these
datatypes to Dylan datatypes (Table 1.6). The methods on the record class consult this mapping when performing data
coercion.

The datatypes of host variables are limited to the Dylan datatypes that appear in Table 1.6. Host variables come in two
flavors: read and write. Host variables appearing in an into clause of an SQL SELECT statement are write parameters,
and all other host variables are read parameters.
Table 6: Table 1.6 Mapping from DBMS to Dylan datatypes

<table>
<thead>
<tr>
<th>DBMS type</th>
<th>SQL type</th>
<th>Dylan type</th>
</tr>
</thead>
<tbody>
<tr>
<td>sql_char</td>
<td>&lt;sql-char&gt;</td>
<td>&lt;character&gt;</td>
</tr>
<tr>
<td>sql_varchar</td>
<td>&lt;sql-varchar&gt;</td>
<td>&lt;string&gt;</td>
</tr>
<tr>
<td>sql_longvarchar</td>
<td>&lt;sql-longvarchar&gt;</td>
<td>&lt;string&gt;</td>
</tr>
<tr>
<td>sql_decimal</td>
<td>&lt;sql-decimal&gt;</td>
<td>&lt;string&gt;</td>
</tr>
<tr>
<td>sql_numeric</td>
<td>&lt;sql-numeric&gt;</td>
<td>&lt;string&gt;</td>
</tr>
<tr>
<td>sql_bit</td>
<td>&lt;sql-bit&gt;</td>
<td>&lt;integer&gt;</td>
</tr>
<tr>
<td>sql_tinyint</td>
<td>&lt;sql-tinyint&gt;</td>
<td>&lt;integer&gt;</td>
</tr>
<tr>
<td>sql_smallint</td>
<td>&lt;sql-smallint&gt;</td>
<td>&lt;integer&gt;</td>
</tr>
<tr>
<td>sql_integer</td>
<td>&lt;sql-integer&gt;</td>
<td>&lt;integer&gt;</td>
</tr>
<tr>
<td>sql_bigint</td>
<td>&lt;sql-bigint&gt;</td>
<td>&lt;integer&gt;</td>
</tr>
<tr>
<td>sql_real</td>
<td>&lt;sql-real&gt;</td>
<td>&lt;single-float&gt;, &lt;double-float&gt; or &lt;extended-float&gt;</td>
</tr>
<tr>
<td>sql_double</td>
<td>&lt;sql-double&gt;</td>
<td>&lt;double-float&gt;</td>
</tr>
<tr>
<td>sql_binary</td>
<td>&lt;sql-binary&gt;</td>
<td>&lt;binary&gt;</td>
</tr>
<tr>
<td>sql_varbinary</td>
<td>&lt;sql-varbinary&gt;</td>
<td>&lt;binary&gt;</td>
</tr>
<tr>
<td>sql_longvarbinary</td>
<td>&lt;sql-longvarbinary&gt;</td>
<td>&lt;binary&gt;</td>
</tr>
<tr>
<td>sql_date</td>
<td>&lt;sql-date&gt;</td>
<td>&lt;date&gt;</td>
</tr>
<tr>
<td>sql_time</td>
<td>&lt;sql-time&gt;</td>
<td>&lt;time&gt;</td>
</tr>
<tr>
<td>sql_timestamp</td>
<td>&lt;sql-timestamp&gt;</td>
<td>&lt;timestamp&gt;</td>
</tr>
</tbody>
</table>

To retrieve integer elements from databases that may contain more than 30-bit data, you must use the generic-arithmetic library or a run-time error will occur. The Dylan SQL-ODBC library must also be prepared.

Example library and module definition:

```dylan
define library sql-example
    use common-dylan;
    use generic-arithmetic;
    use sql-odbc;

    export sql-example;
end library;

define module sql-example
    use generic-arithmetic-common-dylan;
    use sql-odbc;
end module;
```

### 15.17.2 Warning and error conditions

The SQL-ODBC library defines condition classes for each category of error and warning defined in SQL-92. (SQL-92 calls them classes rather than categories.)

When an error or warning occurs, SQL-ODBC detects it, creates a condition object, and signals it. You can then handle the condition using the Dylan condition system.

Some DBMSes can detect and report multiple errors or warnings during the execution of a single SQL statement. The DBMS reports these errors and warnings to the SQL-ODBC library using SQL-92’s concept of diagnostics; the first error or warning in the diagnostic area is the same error or warning indicated by the SQLSTATE status parameter. The SQL-ODBC library signals a condition which corresponds to the error or warning indicated by SQLSTATE.
While handling the first condition, your application can process any additional errors or warnings that may have occurred by signaling the next DBMS condition; to obtain the next DBMS condition, call `next-dbms-condition` on the condition being handled.

## 15.18 Diagnostics

SQL-92 defines a diagnostics area as a DBMS-managed data structure that captures specific information about the execution of a SQL statement, with the exception of the `GET DIAGNOSTICS` statement. A diagnostics area consists of two sections, a header and a collection of diagnostic details.

The header contains information about the last SQL statement executed, while the diagnostic details contain information about each error or warning that resulted from the execution of the SQL statement.

The size of the diagnostic details section is the default value for the DBMS implementation. This size is always greater than one, since the first diagnostic detail corresponds to `sqlstate`. A DBMS may only fill in one diagnostic detail regardless of the number of errors or warnings that occur. If multiple diagnostic details are filled in, there is no presumption of precedence or importance.

The SQL-ODBC library provides wrapper classes for these constructs and accessors for the information they represent. See also:

- `row-count`
- `<diagnostic>`
- `condition-number`
- `returned-sqlstate`
- `class-origin`
- `subclass-origin`
- `connection-name`
- `message-text`

### 15.18.1 Database introspection

The SQL-ODBC library offers introspection features to allow you to determine the structure of a database at run time. A database structure is a hierarchy comprising catalogs, schemas, tables and columns. A catalog is a named collection of schemas, a schema is a named collection of tables, and a table is a named collection of columns. For security reasons, the SQL-ODBC library does not provide any means of obtaining a list of databases available from a particular DBMS; your application must provide access to a particular database via a connection object.

For DBMSes which do not support catalogs or schemas, the SQL-ODBC library uses a default value that your application can use to perform introspection.

### 15.19 Database objects and integrity constraints

You can interrogate schema and table database objects for a collection of constraints defined against them. A constraint is a data integrity rule which the DBMS enforces at all times. These constraints are unique, primary key, referential and check.
The unique constraint specifies that one or more columns within a table must have a unique value or set of values for each record in the table (however, the set of columns are not necessarily a key). The primary key constraint is similar to the unique constraint, except the set of columns must uniquely identify records within the table.

The referential constraint specifies the relationship between a column or a group of columns in one table to another table; this constraint also specifies the action to take when records within the table are updated or deleted.

Finally, the check constraint is a general constraint on a table which must never be false and, due to three-valued logic, an unknown or null value will satisfy the constraint.

An assertion is a constraint on a schema. It is similar to the check constraint but it normally involves more than one table. The significant difference between an assertion and a check is that an assertion must always be true, whereas a check must never be false.

The nullability of a column is a column constraint which can be determined by introspection on the desired column. Syntactically, SQL-92 supports table and column constraints; semantically, however, all constraints are enforced at the table level.

15.19.1 The SQL module

$default-coercion Constant
$default-result-set-policy Constant
$diagnostic-table Constant
$no-coercion Constant
$no-indicator Constant
$null-value Constant

Discussion References the canonical null value. It is an instance of <null-value>.

$read-committed Constant
$read-only Constant
$read-uncommitted Constant
$read-write Constant
$repeatable-read Constant
$scrollable-result-set-policy Constant
$serializable Constant
*all-connections* Constant
*all-connections-lock* Constant
<ambiguous-cursor-name> Open Class

  Superclasses <diagnostic>

  Init-Keywords
    • class-code –

<assertion-constraint> Abstract Class

  Superclasses <constraint>

<cardinality-violation> Open Class
Superclasses <catalog-not-found> Class
Init-Keywords
  • class-code –

<catalog> Open Abstract Class
Superclasses <database-object-not-found>,<result-set>
Init-Keywords
  • catalog-name –

<character-not-in-repertoire> Open Class
Superclasses <data-exception>
Init-Keywords
  • connection –

<check-constraint> Abstract Class
Superclasses <constraint>

<coercion-policy> Constant
  Determines what data coercion is to be performed on a result set.

<coercion-record> Open Abstract Class
Superclasses <record>
Init-Keywords
  • record-coercion-policy –

<column> Open Abstract Class
Superclasses <database-object>
Init-Keywords
  • default-value –
  • domain –
  • nullable? –

<connection-does-not-exist> Open Class
Superclasses <connection-exception>
Init-Keywords
  • subclass-code –

<connection-exception> Open Class
Superclasses <diagnostic>
Init-Keywords
• class-code –
<connection-failure> Open Class
  Superclasses <connection-exception>
  Init-Keywords
  • subclass-code –
<connection-name-in-use> Open Class
  Superclasses <connection-exception>
  Init-Keywords
  • subclass-code –
<connection-not-specified> Open Class
  Superclasses <error>
<connection> Open Abstract Class
  Superclasses <object>
  Init-Keywords
  • dbms –

Discussion The <connection> class represents a database connection. More formally, we can say that it identifies a context in which a client application can execute SQL statements. The exact composition of a connection depends on the DBMS and the client platform. Implementation libraries like SQL-ODBC define a subclass of <connection> that implements the necessary requirements to identify the execution context to the client application.

<constraint> Abstract Class
  Superclasses <database-object>
<cursor-operation-conflict> Open Class
  Superclasses <diagnostic>
  Init-Keywords
  • class-code –
<cursor-specification-cannot-be-executed> Open Class
  Superclasses <dynamic-sql-error>
  Init-Keywords
  • subclass-code –
<data-exception> Open Class
  Superclasses <diagnostic>
  Init-Keywords
  • class-code –
<data-not-available> Open Class
  Superclasses <error>
<database-collection> Open Abstract Class
Superclasses <sequence>

<database-error> Open Abstract Class
Superclasses <error>

<database-object-not-found> Abstract Class
Superclasses <diagnostic>

<database-object> Abstract Class
Superclasses <object>
Init-Keywords
  • name –

<database-statement> Open Abstract Class
Superclasses <object>

Discussion This class represents statements which can be executed by a DBMS server.

<database> Open Abstract Class
Superclasses <object>

Discussion The <database> class identifies a database to a DBMS. Exactly what a database is depends on the DBMS in use. Implementation libraries like SQL-ODBC supply an instantiable subclass of <database> to provide whatever implementation is necessary for identifying a database to a specific DBMS.

<datetime-field-overflow> Open Class
Superclasses <data-exception>
Init-Keywords
  • subclass-code –

<dbms-not-specified> Open Class
Superclasses <error>

<dbms> Open Abstract Class
Superclasses <object>

Discussion The <dbms> class identifies a database management system (DBMS) to a client application. Implementation libraries like SQL-ODBC supply an instantiable subclass of <dbms> to provide whatever implementation is necessary for identifying a DBMS to an application.

<dependent-privilege-descriptors-still-exist> Open Class
Superclasses <diagnostic>
Init-Keywords
  • class-code –

<diagnostic-table> Class
Superclasses <object>
Init-Keywords
  • general-key –

<diagnostic> Open Abstract Class
Superclasses `<condition>`
Init-Keywords
- `class-code`
- `condition-number`
- `subclass-code`

`<disconnect-error>` Open Class
Superclasses `<sql-warning>`
Init-Keywords
- `subclass-code`

`<division-by-zero>` Open Class
Superclasses `<data-exception>`
Init-Keywords
- `subclass-code`

`<dynamic-sql-error>` Open Class
Superclasses `<diagnostic>`
Init-Keywords
- `class-code`

`<empty-result-set>` Open Class
Superclasses `<result-set>`
Init-Keywords
- `liaison`

`<error-in-assignment>` Open Class
Superclasses `<data-exception>`
Init-Keywords
- `subclass-code`

`<feature-not-supported>` Open Class
Superclasses `<diagnostic>`
Init-Keywords
- `class-code`

`<forward-only-result-set>` Open Abstract Class
The class for result sets that support a one-shot forward iteration protocol.

Superclasses `<result-set>`

Discussion
Instances of this class represent the results of an SQL `SELECT` statement, and support a one-shot forward iteration protocol. By one-shot, we mean each element of the collection can be visited only once, and no previously visited element can be revisited. A condition is signaled if the application tries to revisit a record. Thus, backward iteration protocol is not supported on this collection.
This collection class is useful when the result of a query is large and each element can be processed individually.

The function type-for-copy returns <simple-object-vector> when applied to objects of this class.

See also

• <scrollable-result-set>

<implicit-zero-bit-padding> Open Class
Superclasses <sql-warning>
Init-Keywords
  • subclass-code –

<index> Open Abstract Class
Superclasses <database-object>
Init-Keywords
  • indexed-table –
  • unique-index? –

<indicator-overflow> Open Class
Superclasses <data-exception>
Init-Keywords
  • subclass-code –

<indicator-policy> Constant

<insufficient-item-descriptor-areas> Open Class
Superclasses <sql-warning>
Init-Keywords
  • subclass-code –

<integrity-constraint-violation> Open Class
Superclasses <diagnostic>
Init-Keywords
  • class-code –

<interval-field-overflow> Open Class
Superclasses <data-exception>
Init-Keywords
  • subclass-code –

<invalid-argument> Open Class
Superclasses <error>

<invalid-authorization-specification> Open Class
Superclasses <diagnostic>
Init-Keywords
• class-code –
<invalid-catalog-name> Open Class
   Superclasses <diagnostic>
   Init-Keywords
   • class-code –
<invalid-character-set-name> Open Class
   Superclasses <diagnostic>
   Init-Keywords
   • class-code –
<invalid-character-value-for-cast> Open Class
   Superclasses <data-exception>
   Init-Keywords
   • subclass-code –
<invalid-condition-number> Open Class
   Superclasses <diagnostic>
   Init-Keywords
   • class-code –
<invalid-cursor-name> Open Class
   Superclasses <diagnostic>
   Init-Keywords
   • class-code –
<invalid-datatype-hint> Open Class
   Superclasses <warning>
   Init-Keywords
   • datatype-hint –
<invalid-datetime-format> Open Class
   Superclasses <data-exception>
   Init-Keywords
   • subclass-code –
<invalid-descriptor-count> Open Class
   Superclasses <dynamic-sql-error>
   Init-Keywords
   • subclass-code –
<invalid-descriptor-index> Open Class
   Superclasses <dynamic-sql-error>
   Init-Keywords
• subclass-code –
<invalid-escape-character> Open Class
  Superclasses <data-exception>
  Init-Keywords
    • subclass-code –
<invalid-escape-sequence> Open Class
  Superclasses <data-exception>
  Init-Keywords
    • subclass-code –
<invalid-fetch-sequence> Open Class
  Superclasses <data-exception>
  Init-Keywords
    • subclass-code –
<invalid-parameter-value> Open Class
  Superclasses <data-exception>
  Init-Keywords
    • subclass-code –
<invalid-schema-name> Open Class
  Superclasses <diagnostic>
  Init-Keywords
    • class-code –
<invalid-sql-descriptor-name> Open Class
  Superclasses <diagnostic>
  Init-Keywords
    • class-code –
<invalid-sql-statement-name> Open Class
  Superclasses <diagnostic>
  Init-Keywords
    • class-code –
<invalid-time-zone-displacement-value> Open Class
  Superclasses <data-exception>
  Init-Keywords
    • subclass-code –
<invalid-transaction-state> Open Class
  Superclasses <diagnostic>
  Init-Keywords
• **class-code** –

**<invalid-transaction-termination> Open Class**

Superclasses **<diagnostic>**

Init-Keywords

• **class-code** –

**<isolation-level> Constant**

**<multiple-server-transaction> Open Class**

Superclasses **<feature-not-supported>**

Init-Keywords

• **subclass-code** –

**<no-data> Open Class**

Superclasses **<diagnostic>**

Init-Keywords

• **class-code** –

**<null-value-eliminated-in-set-function> Open Class**

Superclasses **<sql-warning>**

Init-Keywords

• **subclass-code** –

**<null-value-no-indicator-parameter> Open Class**

Superclasses **<data-exception>**

Init-Keywords

• **subclass-code** –

**<null-value> Open Class**

Superclasses **<object>**

Discussion Instances of this class represent the canonical null value. This class is the root class for all null-value classes.

**<numeric-value-out-of-range> Open Class**

Superclasses **<data-exception>**

Init-Keywords

• **subclass-code** –

**<prepared-statement-not-a-cursor-specification> Open Class**

Superclasses **<dynamic-sql-error>**

Init-Keywords

• **subclass-code** –

**<privilege-not-granted> Open Class**

Superclasses **<sql-warning>**
The class of records retrieved from a DBMS table as the result of executing an SQL SELECT statement.

Instances of this class represent a record that was retrieved from a DBMS table as the result of executing an SQL SELECT statement.

If the value passed to coercion-policy is a sequence whose size is less than the degree of the record, the extra columns are converted to their equivalent Dylan type using the default coercion. If the size of the sequence is greater than the degree of the record, the extra elements of the sequence are ignored.

The class of records retrieved from a DBMS table as the result of executing an SQL SELECT statement.

Instances of this class represent a record that was retrieved from a DBMS table as the result of executing an SQL SELECT statement.

If the value passed to coercion-policy is a sequence whose size is less than the degree of the record, the extra columns are converted to their equivalent Dylan type using the default coercion. If the size of the sequence is greater than the degree of the record, the extra elements of the sequence are ignored.

Specifies the behavior and performance characteristics of a result set.

Instances of this class represent a result set that was retrieved from a DBMS table as the result of executing an SQL SELECT statement.

If the value passed to coercion-policy is a sequence whose size is less than the degree of the record, the extra columns are converted to their equivalent Dylan type using the default coercion. If the size of the sequence is greater than the degree of the record, the extra elements of the sequence are ignored.
• **rowset-size** – An instance of type-union(<integer>, "all").

• **scroll-window** – An instance of <integer>. A cache size hint.

• **scrollable** – An instance of <boolean>. Default value: #f.

Specifies the behavior and performance characteristics of a result set.

The rowset-size slot is the number of records to retrieve each time an internal fetch is performed. If rowset-size is "all", all records are retrieved the first time a fetch is performed. Currently, rowset-size is ignored.

<result-set> Open Abstract Class

**Superclasses** <database-collection>

**Init-Keywords**

• **liaison** –

**Discussion**

Instances of this class represent the results of an SQL SELECT statement.

This class is the root class for all result-set classes. The type-for-copy function returns <simple-object-vector> for objects of this class.

See also

• <forward-only-result-set>

• <scrollable-result-set>

<schema-not-found> Class

**Superclasses** <database-object-not-found>

**Init-Keywords**

• **schema-name** –

<schema> Open Abstract Class

**Superclasses** <database-object>, <result-set>

<scrollable-result-set> Open Abstract Class

The class for result sets that support both forward and backward iteration.

**Superclasses** <result-set>

**Discussion**

Instances of this class support both the forward- and backward-iteration-protocol.

The type-for-copy function returns <simple-object-vector> for objects of this class.

See also

• <forward-only-result-set>

<search-condition-too-long-for-information-schema> Open Class

**Superclasses** <sql-warning>

**Init-Keywords**

• **subclass-code** –

<sql-bigint> Open Class
Superclasses <sql-datatype>
<sql-binary> Open Class
Superclasses <sql-datatype>
<sql-bit-varying> Open Class
Superclasses <sql-datatype>
<sql-bit> Open Class
Superclasses <sql-datatype>
<sql-character-varying> Open Class
Superclasses <sql-datatype>
<sql-character> Open Class
Superclasses <sql-datatype>
<sql-client-unable-to-establish-connection> Open Class
Superclasses <connection-exception>
Init-Keywords
  • subclass-code –
<sql-datatype> Open Abstract Class
Superclasses <object>
<sql-date> Open Class
Superclasses <sql-datatype>
<sql-day-time-interval> Open Class
Superclasses <sql-datatype>
<sql-decimal> Open Class
Superclasses <sql-datatype>
<sql-double-precision> Open Class
Superclasses <sql-datatype>
<sql-double> Open Class
Superclasses <sql-datatype>
<sql-error> Open Abstract Class
Superclasses <database-error>
<sql-float> Open Class
Superclasses <sql-datatype>
<sql-integer> Open Class
Superclasses <sql-datatype>
<sql-longvarbinary> Open Class
Superclasses <sql-datatype>
<sql-longvarchar> Open Class
Superclasses `<sql-datatype>`

```<sql-national-character-varying> Open Class
    Superclasses `<sql-character-varying>`
```

```<sql-national-character> Open Class
    Superclasses `<sql-character>`
```

```<sql-numeric> Open Class
    Superclasses `<sql-character>`
```

```<sql-real> Open Class
    Superclasses `<sql-datatype>`
```

```<sql-server-rejected-establishment-of-connection> Open Class
    Superclasses `<connection-exception>`
```

Init-Keywords

• `subclass-code` –

```<sql-smallint> Open Class
    Superclasses `<sql-datatype>`
```

```<sql-statement> Open Abstract Class
    Superclasses `<database-statement>`
```

Init-Keywords

• `coercion-policy` – An instance of `false-or(<coercion-policy>)`. The coercion policy is a sequence of functions, or the value `$default-coercion`, or the value `$no-coercion`, used to perform data coercion when the SQL statement to be executed is a SELECT statement.

• `datatype-hints` – An instance of `false-or(<sequence>)`. This is a hint for parameter binding when the SQL statement to be executed is a SELECT statement.

• `input-indicator` – An instance of `<object>`. The input indicator is a marker value used to identify null values in host variables.

• `output-indicator` – An instance of `<object>`. The output indicator is a substitution value to be used whenever the column of a retrieved record contains the null value.

• `text` – An instance of `<string>`. Required. Contains the text of the SQL statement. If you want to include host variables, place a question mark (?) at the point in the string at which you want a host variable to be substituted.

Discussion

The `<sql-statement>` class represents SQL statements and their indicator values and coercion policy. You can use this class to represent any SQL statement, be it static or dynamic. You can send SQL statements to the DBMS for execution by calling the `execute` function on an instance of `<sql-statement>`. The `execute` function returns the results of executing the SQL statement, if there are any.

In the `make` method on `<sql-statement>`, you can specify that values should be substituted into the SQL statement when it is executed. You do not specify the values until calling `execute` on the statement, when you can pass the substitution values with the `parameter:` keyword.
The values are substituted wherever a question mark (?) occurs in the SQL statement string. We call the question marks anonymous host variables because there is no Dylan variable name. Substitution occurs positionally: the first value replaces the first anonymous host variable, the second value replaces the second anonymous host variable, and so on. If the number of values is greater than the number of anonymous host variables, the extra parameters are ignored. If the number of anonymous host variables is greater than the number of parameters, a condition is signaled.

When the SQL statement is SELECT, you can also specify a result-set policy and a liaison function in the call to execute. A result-set policy describes behavioral and performance characteristics of the result-set object that the execute function returns. A liaison function creates Dylan objects from the records retrieved from the database. These objects become the elements of the result set instead of the record object.

```
<sql-table> Open Abstract Class
  Superclasses <database-object>, <result-set>
<sql-time-with-time-zone> Open Class
  Superclasses <sql-datatype>
<sql-time> Open Class
  Superclasses <sql-datatype>
<sql-timestamp-with-time-zone> Open Class
  Superclasses <sql-datatype>
<sql-timestamp> Open Class
  Superclasses <sql-datatype>
<sql-tinyint> Open Class
  Superclasses <sql-datatype>
<sql-type-timestamp> Open Class
  Superclasses <sql-datatype>
<sql-unknown-type> Open Class
  Superclasses <sql-datatype>
<sql-unsupported-type> Open Class
  Superclasses <sql-datatype>
<sql-varbinary> Open Class
  Superclasses <sql-datatype>
<sql-warning> Open Class
  Superclasses <diagnostic>
  Init-Keywords
    • class-code -
<sql-year-month-interval> Open Class
  Superclasses <sql-datatype>
<statement-completion-unknown> Open Class
```
Superclasses <transaction-rollback>
Init-Keywords
  • subclass-code –
<string-data-length-mismatch> Open Class
Superclasses <data-exception>
Init-Keywords
  • subclass-code –
<string-data-right-truncation> Open Class
Superclasses <data-exception>
Init-Keywords
  • subclass-code –
<string-error> Open Class
Superclasses <data-exception>
Init-Keywords
  • subclass-code –
<successful-completion> Open Class
Superclasses <diagnostic>
Init-Keywords
  • class-code –
.syntax-error-or-access-rule-violation-in-direct-sql-statement> Open Class
Superclasses <diagnostic>
Init-Keywords
  • class-code –
<syntax-error-or-access-rule-violation-in-dynamic-sql-statement> Open Class
Superclasses <diagnostic>
Init-Keywords
  • class-code –
<syntax-error-or-access-rule-violation> Open Class
Superclasses <diagnostic>
Init-Keywords
  • class-code –
<table-not-found> Class
Superclasses <database-object-not-found>
Init-Keywords
  • table-name –
<transaction-mode> Constant
<transaction-resolution-unknown> Open Class
  Superclasses <connection-exception>
  Init-Keywords
    • subclass-code –

<transaction-rollback-due-to-integrity-constraint-violation> Open Class
  Superclasses <transaction-rollback>
  Init-Keywords
    • subclass-code –

<transaction-rollback-due-to-serialization-failure> Open Class
  Superclasses <transaction-rollback>
  Init-Keywords
    • subclass-code –

<transaction-rollback> Open Class
  Superclasses <diagnostic>
  Init-Keywords
    • class-code –

<transaction> Open Class
  Superclasses <object>
  Init-Keywords
    • diagnostics-size –
    • isolation-level –
    • transaction-mode –

<triggered-data-change-violation> Open Class
  Superclasses <diagnostic>
  Init-Keywords
    • class-code –

<trim-error> Open Class
  Superclasses <data-exception>
  Init-Keywords
    • subclass-code –

<unhandled-diagnostic> Open Class
  Superclasses <sql-error>
  Init-Keywords
    • diagnostic –

<unique-constraint> Abstract Class
  Superclasses <constraint>
The <user> class identifies a user to a DBMS. Exactly what a “user” means depends on the DBMS. Implementation libraries like SQL-ODBC supply an instantiable subclass of <user> to provide whatever implementation is necessary for identifying a user to a specific DBMS.

When connecting to a DBMS that did not have any users per se, instances of <user> would merely satisfy the API protocol, and would not identify a specific user — any instance of <user> would identify all users to the DBMS. However, most DBMSes do require a user name and password to identify a specific user. Indeed, some DBMSes require stringent authorization information in order to identify a user, such as multiple passwords.
Superclasses <sql-warning>
Init-Keywords
  • subclass-code –
<warning-string-data-right-truncation> Open Class
Superclasses <sql-warning>
Init-Keywords
  • subclass-code –
<warning-string-data-right-truncation> Open Class
Superclasses <diagnostic>
Init-Keywords
  • class-code –
acquire-null-value Generic function
Signature acquire-null-value (indicator index) => (null-value)
Parameters
  • indicator – An instance of <object>.
  • index – An instance of <integer>.
Values
  • null-value – An instance of <object>.
asynchronous Generic function
Signature asynchronous (object) => (#rest results)
Parameters
  • object – An instance of <object>.
Values
  • #rest results – An instance of <object>.
catalog-from-name Open Generic function
Signature catalog-from-name (connection name) => (catalog)
Parameters
  • connection – An instance of <connection>.
  • name – An instance of <string>.
Values
  • catalog – An instance of <catalog>.
catalog-name Open Generic function
Signature catalog-name (diag) => (catalog-name)
Parameters
  • diag – An instance of <diagnostic>.
Values
• **catalog-name** – An instance of `<string>`.

catalogs **Open Generic function**

**Signature** catalogs (#key connection) => (result-set)

**Parameters**

- **connection (#key)** – An instance of `<connection>`.

**Values**

- **result-set** – An instance of `<result-set>`.

catalogs-assist **Open Generic function**

**Signature** catalogs-assist (connection) => (result-set)

**Parameters**

- **connection** – An instance of `<connection>`.

**Values**

- **result-set** – An instance of `<result-set>`.

class-code **Generic function**

**Signature** class-code (object) => (#rest results)

**Parameters**

- **object** – An instance of `<object>`.

**Values**

- **#rest results** – An instance of `<object>`.

class-origin **Open Generic function**

**Signature** class-origin (diag) => (class-origin)

**Parameters**

- **diag** – An instance of `<diagnostic>`.

**Values**

- **class-origin** – An instance of `<string>`.

coercion-policy **Generic function**

**Signature** coercion-policy (sql-statement) => (coercion-policy)

**Parameters**

- **sql-statement** – An instance of `<sql-statement>`.

**Values**

- **coercion-policy** – An instance of `<coercion-policy>`.

**Discussion** Returns the coercion policy for sql-statement. This method is only relevant to SQL SELECT statements.

coaercion-policy-setter **Generic function**

**Signature** coercion-policy-setter (new-coercion-policy sql-statement) => (new-coercion-policy)

**Parameters**
• new-coercion-policy – An instance of <coercion-policy>.
• sql-statement – An instance of <sql-statement>.

Values

• new-coercion-policy – An instance of <coercion-policy>.

Discussion Sets the coercion-policy slot of sql-statement to new-coercion-policy.

column-name Open Generic function

Signature column-name (diag) => (column-name)

Parameters

• diag – An instance of <diagnostic>.

Values

• column-name – An instance of <string>.

command-function Open Generic function

Signature command-function (diag) => (command-function)

Parameters

• diag – An instance of <diagnostic>.

Values

• command-function – An instance of <string>.

commit-transaction Open Generic function

Signature commit-transaction (transaction) => ()

Parameters

• transaction – An instance of <transaction>.

condition-number Generic function

Signature condition-number (object) => (#rest results)

Parameters

• object – An instance of <object>.

Values

• #rest results – An instance of <object>.

conditions-not-recorded? Open Generic function

Signature conditions-not-recorded? (diag) => (not-recorded-status)

Parameters

• diag – An instance of <diagnostic>.

Values

• not-recorded-status – An instance of <boolean>.

connect Open Generic function

Signature connect (database user) => (connection)
Parameters

- **database** – An instance of `<database>`.
- **user** – An instance of `<user>`.

Values

- **connection** – An instance of `<connection>`.

`connect-with-prompt` Open Generic function

**Signature**

`connect-with-prompt (dbms #key database user) => (connection)`

**Parameters**

- **dbms** – An instance of `<dbms>`.
- **database (#key)** – An instance of `false-or(<database>)`.
- **user (#key)** – An instance of `false-or(<user>)`.

Values

- **connection** – An instance of `<connection>`.

`connect-with-prompt?` Open Generic function

**Signature**

`connect-with-prompt? (dbms) => (connect-with-prompt-status)`

**Parameters**

- **dbms** – An instance of `<dbms>`.

Values

- **connect-with-prompt-status** – An instance of `<boolean>`.

`connection` Open Generic function

**Signature**

`connection (o) => (result)`

**Parameters**

- **o** – An instance of `<object>`.

Values

- **result** – An instance of `<connection>`.

`connection-name` Open Generic function

**Signature**

`connection-name (diag) => (connection-name)`

**Parameters**

- **diag** – An instance of `<diagnostic>`.

Values

- **connection-name** – An instance of `<string>`.

`connection-setter` Open Generic function

**Signature**

`connection-setter (c o) => (result)`

**Parameters**

- **c** – An instance of `<connection>`.
• o – An instance of <object>.

Values
• result – An instance of <connection>.

connections Open Generic function
Signature connections (#key dbms) => (connection-sequence)
Parameters
• dbms (#key) – An instance of false-or(<dbms>).

Values
• connection-sequence – An instance of <sequence>.

constraint-catalog Open Generic function
Signature constraint-catalog (diag) => (constraint-catalog)
Parameters
• diag – An instance of <diagnostic>.

Values
• constraint-catalog – An instance of <string>.

constraint-name Open Generic function
Signature constraint-name (diag) => (constraint-name)
Parameters
• diag – An instance of <diagnostic>.

Values
• constraint-name – An instance of <string>.

constraint-schema Open Generic function
Signature constraint-schema (diag) => (constraint-schema)
Parameters
• diag – An instance of <diagnostic>.

Values
• constraint-schema – An instance of <string>.

constraints Open Generic function
Signature constraints (db-object) => (result)
Parameters
• db-object – An instance of <database-object>.

Values
• result – An instance of <result-set>.

convert-value Generic function
Signature convert-value (coercion-policy value key) => (converted-value)
Parameters
• **coercion-policy** – An instance of `<coercion-policy>`.
• **value** – An instance of `<object>`.
• **key** – An instance of `<integer>`.

**Values**
• **converted-value** – An instance of `<object>`.

**cursor-name** Open Generic function

**Signature** cursor-name (diag) => (cursor-name)

**Parameters**
• **diag** – An instance of `<diagnostic>`.

**Values**
• **cursor-name** – An instance of `<string>`.

**database** Open Generic function

**Signature** database (connection) => (database)

**Parameters**
• **connection** – An instance of `<connection>`.

**Values**
• **database** – An instance of `<database>`.

**database-object-name** Generic function

**Signature** database-object-name (object) => (#rest results)

**Parameters**
• **object** – An instance of `<object>`.

**Values**
• **#rest results** – An instance of `<object>`.

**database-object-name-setter** Generic function

**Signature** database-object-name-setter (value object) => (#rest results)

**Parameters**
• **value** – An instance of `<object>`.
• **object** – An instance of `<object>`.

**Values**
• **#rest results** – An instance of `<object>`.

**datatype-hints** Generic function

**Signature** datatype-hints (object) => (#rest results)

**Parameters**
• **object** – An instance of `<object>`.

**Values**
• **#rest results** – An instance of `<object>`.

15.19. Database objects and integrity constraints
datatype-hints-setter Generic function
   Signature  datatype-hints-setter (value object) => (#rest results)
   Parameters
      • value – An instance of <object>.
      • object – An instance of <object>.
   Values
      • #rest results – An instance of <object>.

dbms Open Generic function
   Signature  dbms (connection) => (dbms)
   Parameters
      • connection – An instance of <connection>.
   Values
      • dbms – An instance of <dbms>.

dbms-name Open Generic function
   Signature  dbms-name (dbms #key connection) => (dbms-name)
   Parameters
      • dbms – An instance of <dbms>.
      • connection (#key) – An instance of <connection>.
   Values
      • dbms-name – An instance of <string>.

dbms-version Open Generic function
   Signature  dbms-version (dbms #key connection) => (dbms-version)
   Parameters
      • dbms – An instance of <dbms>.
      • connection (#key) – An instance of <connection>.
   Values
      • dbms-version – An instance of <string>.

default-connection Generic function
   Signature  default-connection () => (connection)
   Values
      • connection – An instance of <connection>.

default-conversion Open Generic function
   Signature  default-conversion (value) => (converted-value)
   Parameters
      • value – An instance of <object>.
   Values
• **converted-value** – An instance of `<object>`.

**default-dbms** Generic function

*Signature*  
default-dbms () => (dbms)

*Values*

• **dbms** – An instance of `<dbms>`.

**default-diagnostics-size** Open Generic function

*Signature*  
default-diagnostics-size (connection) => (diagnostics-size)

*Parameters*

• **connection** – An instance of `<connection>`.

*Values*

• **diagnostics-size** – An instance of `<integer>`.

**default-isolation-level** Open Generic function

*Signature*  
default-isolation-level (connection) => (level)

*Parameters*

• **connection** – An instance of `<connection>`.

*Values*

• **level** – An instance of `<isolation-level>`.

**default-transaction-mode** Open Generic function

*Signature*  
default-transaction-mode (connection) => (mode)

*Parameters*

• **connection** – An instance of `<connection>`.

*Values*

• **mode** – An instance of `<transaction-mode>`.

**default-value** Open Generic function

*Signature*  
default-value (column) => (default)

*Parameters*

• **column** – An instance of `<column>`.

*Values*

• **default** – An instance of `<object>`.

**diagnostic-to-string** Open Generic function

*Signature*  
diagnostic-to-string (diag) => (string)

*Parameters*

• **diag** – An instance of `<diagnostic>`.

*Values*

• **string** – An instance of `<string>`.
Signature diagnostics-size (object) => (#rest results)

Parameters
• object – An instance of <object>.

Values
• #rest results – An instance of <object>.

diagnostics-size-setter Generic function
Signature diagnostics-size-setter (value object) => (#rest results)
Parameters
• value – An instance of <object>.
• object – An instance of <object>.

Values
• #rest results – An instance of <object>.

disconnect Open Generic function
Signature disconnect (connection #key terminate-statements) => ()
Parameters
• connection – An instance of <connection>.
• terminate-statements (#key) – An instance of <boolean>.

disconnect-all Open Generic function
Signature disconnect-all (#key dbms) => ()
Parameters
• dbms (#key) – An instance of false-or(<dbms>).

domain Generic function
Signature domain (object) => (#rest results)
Parameters
• object – An instance of <object>.

Values
• #rest results – An instance of <object>.

dynamic-function Open Generic function
Signature dynamic-function (diag) => (dynamic-function)
Parameters
• diag – An instance of <diagnostic>.

Values
• dynamic-function – An instance of <string>.

dynamic-function
• **transaction** – An instance of `<transaction>`.

**environment-name** Open Generic function

*Signature*  
`environment-name (diag) => (env-name)`

*Parameters*

- **diag** – An instance of `<diagnostic>`.

*Values*

- **env-name** – An instance of `<string>`.

**execute** Open Generic function

Prepares an SQL statement for execution on the specified connection and then executes the statement.

*Signature*  
`execute (database-statement #key #all-keys) => (result-set)`

*Parameters*

- **database-statement** – An instance of `type-union(<database-statement>, <string>)`.
- **connection (#key)** – An instance of `<connection>`.
- **parameters (#key)** – An instance of `false-or(<sequence>)`.
- **result-set-policy (#key)** – An instance of `false-or(<result-set-policy>)`.
- **liaison (#key)** – An instance of `false-or(<function>)` whose signature is `liaison(<record>) => <object>`. Default value: `default-liaison`.

*Values*

- **result-set** – An instance of `false-or(<result-set>)`.

**Discussion**

Prepares the SQL statement represented by `sql-statement` for execution on the connection, then sends it to the DBMS for execution.

If connection is not supplied, execute uses the connection returned by `default-connection` instead.

The liaison function is invoked on each record as it is retrieved from the database. If a liaison function is not provided, a default value of `default-liaison` is used; each result-set class has its own `default-liaison`.

In the SQL-ODBC library, the `database-statement` will be an instance of `<sql-statement>`. If anonymous host variables—that is, question marks (?)—appear in `database-statement`, pass suitable substitution values in the call to this function.

**Example**

This example executes two SQL statements against the database represented by `the-connection`. The first SQL statement inserts a new book record into the book table. The second SQL statement queries for the list of titles and their ISBN published by Addison Wesley.

```dylan
with-connection(the-connection)
  let insert-stmt :: <sql-statement> =
    make(<sql-statement>),
    text: "insert into book (title, publisher, isbn)
      values (?, ?, ?)",
    input-indicator: $null-value);
  execute(insert-stmt,
  ```

(continues on next page)
let query-stmt :: <sql-statement> = make(<sql-statement>,
text: "select title, isbn from book
where publisher = ?",
output-indicator: $null-value);
execute(query-stmt, parameters: #("Addison-Wesley"));
end with-connection;

=> #(#("An Introduction to Database Systems", "0-201-14201-5"),
   #("Relational Database Writings, 1991-1994", "0-8053-1748-1"), #("Large Databases", $null-value))

fields Generic function

Signature  fields (object) => (#rest results)
Parameters
• object – An instance of <object>.
Values
• #rest results – An instance of <object>.

fields-setter Generic function

Signature  fields-setter (value object) => (#rest results)
Parameters
• value – An instance of <object>.
• object – An instance of <object>.
Values
• #rest results – An instance of <object>.

find-diagnostic Function

Signature  find-diagnostic (table diagnostic-set-key sqlstate) => (diagnostic-detail-class)
Parameters
• table – An instance of <diagnostic-table>.
• diagnostic-set-key – An instance of <object>.
• sqlstate – An instance of <string>.
Values
• diagnostic-detail-class – An instance of <object>.

indexed-table Generic function

Signature  indexed-table (object) => (#rest results)
Parameters
• object – An instance of <object>.
Values
indexed-table-setter Generic function
Signature indexed-table-setter (value object) => (#rest results)
Parameters
• value – An instance of <object>.
• object – An instance of <object>.
Values
• #rest results – An instance of <object>.
indexes Open Generic function
Signature indexes (table) => (index-collection)
Parameters
• table – An instance of <sql-table>.
Values
• index-collection – An instance of <result-set>.
indicator-policy Generic function
Signature indicator-policy (object) => (#rest results)
Parameters
• object – An instance of <object>.
Values
• #rest results – An instance of <object>.
input-indicator Open Generic function
Signature input-indicator (sql-statement) => (input-indicator)
Parameters
• sql-statement – An instance of <sql-statement>.
Values
• input-indicator – An instance of <indicator-policy>.
input-indicator-setter Open Generic function
Signature input-indicator-setter (new-input-indicator sql-statement) => (new-input-indicator)
Parameters
• new-input-indicator – An instance of <indicator-policy>.
• sql-statement – An instance of <sql-statement>.
Values
• new-input-indicator – An instance of <indicator-policy>.
install-diagnostic Function
Signature install-diagnostic (table class #key key) => ()
Parameters
• **table** – An instance of `<diagnostic-table>`.
• **class** – An instance of `subclass(<diagnostic>)`.
• **key (#key)** – An instance of `<symbol>`.

**install-diagnostic-key Function**

**Signature**
`install-diagnostic-key (key) => ()`

**Parameters**
• **key** – An instance of `<symbol>`.

**installation-functions** Generic function

**Signature**
`installation-functions (object) => (#rest results)`

**Parameters**
• **object** – An instance of `<object>`.

**Values**
• **#rest results** – An instance of `<object>`.

**is-null?** Generic function

**Signature**
`is-null? (record key) => (null-state)`

**Parameters**
• **record** – An instance of `<record>`.
• **key** – An instance of `<integer>`.

**Values**
• **null-state** – An instance of `<boolean>`.

**isolation-level** Generic function

**Signature**
`isolation-level (object) => (#rest results)`

**Parameters**
• **object** – An instance of `<object>`.

**Values**
• **#rest results** – An instance of `<object>`.

**isolation-level-setter** Generic function

**Signature**
`isolation-level-setter (value object) => (#rest results)`

**Parameters**
• **value** – An instance of `<object>`.
• **object** – An instance of `<object>`.

**Values**
• **#rest results** – An instance of `<object>`.

**liaison** Generic function

**Signature**
`liaison (object) => (#rest results)`

**Parameters**
• object – An instance of <object>.

Values

• #rest results – An instance of <object>.

liaison-setter Generic function

Signature  liaison-setter (value object) => (#rest results)

Parameters

• value – An instance of <object>.
• object – An instance of <object>.

Values

• #rest results – An instance of <object>.

make-dbms-specific Open Generic function

Signature  make-dbms-specific (type dbms #rest more-args) => (instance)

Parameters

• type – An instance of <class>.
• dbms – An instance of <dbms>.
• more-args (#rest) – An instance of <object>.

Values

• instance – An instance of <object>.

message-text Open Generic function

Signature  message-text (diag) => (message-text)

Parameters

• diag – An instance of <diagnostic>.

Values

• message-text – An instance of <string>.

multiple-connections? Open Generic function

Signature  multiple-connections? (dbms) => (multiple-connections-status)

Parameters

• dbms – An instance of <dbms>.

Values

• multiple-connections-status – An instance of <boolean>.

next-dbms-diagnostic Open Generic function

Signature  next-dbms-diagnostic (diag) => (next-diagnostic)

Parameters

• diag – An instance of <diagnostic>.

Values

• next-diagnostic – An instance of false-or(<diagnostic>).
nullable? Generic function

Signature nullable? (object) => (#rest results)

Parameters

• object – An instance of <object>.

Values

• #rest results – An instance of <object>.

output-indicator Open Generic function

Signature output-indicator (sql-statement) => (output-indicator)

Parameters

• sql-statement – An instance of <sql-statement>.

Values

• output-indicator – An instance of <indicator-policy>.

output-indicator-setter Open Generic function

Signature output-indicator-setter (new-output-indicator sql-statement) => (new-output-indicator)

Parameters

• new-output-indicator – An instance of <indicator-policy>.
• sql-statement – An instance of <sql-statement>.

Values

• new-output-indicator – An instance of <indicator-policy>.

possible-explanation Generic function

Signature possible-explanation (object) => (#rest results)

Parameters

• object – An instance of <object>.

Values

• #rest results – An instance of <object>.

record-available? Open Generic function

Signature record-available? (result-set key) => (availability)

Parameters

• result-set – An instance of <result-set>.
• key – An instance of <integer>.

Values

• availability – An instance of <boolean>.

record-coercion-policy Generic function

Signature record-coercion-policy (object) => (#rest results)

Parameters

• object – An instance of <object>.
Values

- **#rest results** – An instance of `<object>`.

register-diagnostic-installer Function

Signature: register-diagnostic-installer (function) => ()
Parameters

- **function** – An instance of `<function>`.

returned-sqlstate Open Generic function

Signature: returned-sqlstate (diag) => (sqlstate)
Parameters

- **diag** – An instance of `<diagnostic>`.

Values

- **sqlstate** – An instance of `<string>`.

rollback-transaction Open Generic function

Signature: rollback-transaction (transaction) => ()
Parameters

- **transaction** – An instance of `<transaction>`.

row-count Open Generic function

Signature: row-count (diag) => (count)
Parameters

- **diag** – An instance of `<diagnostic>`.

Values

- **count** – An instance of `<integer>`.

rowset-size Generic function

Signature: rowset-size (object) => (#rest results)
Parameters

- **object** – An instance of `<object>`.

Values

- **#rest results** – An instance of `<object>`.

schema-from-name Open Generic function

Signature: schema-from-name (connection catalog-name schema-name) => (schema)
Parameters

- **connection** – An instance of `<connection>`.
- **catalog-name** – An instance of `<string>`.
- **schema-name** – An instance of `<string>`.

Values

- **schema** – An instance of `<schema>`.
schema-name Open Generic function

Signature schema-name (diag) => (schema-name)
Parameters
• diag – An instance of <diagnostic>.
Values
• schema-name – An instance of <string>.

scroll-window Generic function

Signature scroll-window (object) => (#rest results)
Parameters
• object – An instance of <object>.
Values
• #rest results – An instance of <object>.

scrollable? Generic function

Signature scrollable? (object) => (#rest results)
Parameters
• object – An instance of <object>.
Values
• #rest results – An instance of <object>.

sql Macro

start-transaction Open Generic function

Signature start-transaction (connection transaction-mode isolation-level diagnostics-size) => (transaction)
Parameters
• connection – An instance of <connection>.
• transaction-mode – An instance of <transaction-mode>.
• isolation-level – An instance of <isolation-level>.
• diagnostics-size – An instance of <integer>.
Values
• transaction – An instance of <transaction>.

statement-column-names Open Generic function

Signature statement-column-names (statement) => (column-names)
Parameters
• statement – An instance of <sql-statement>.
Values
• column-names – An instance of <sequence>.

subclass-code Generic function
Signature subclass-code (object) => (#rest results)

Parameters

• object – An instance of <object>.

Values

• #rest results – An instance of <object>.

subclass-origin Open Generic function

Signature subclass-origin (diag) => (subclass-origin)

Parameters

• diag – An instance of <diagnostic>.

Values

• subclass-origin – An instance of <string>.

table-from-name Open Generic function

Signature table-from-name (connection catalog-name schema-name table-name) => (table)

Parameters

• connection – An instance of <connection>.
• catalog-name – An instance of <string>.
• schema-name – An instance of <string>.
• table-name – An instance of <string>.

Values

• table – An instance of <sql-table>.

table-name Open Generic function

Signature table-name (diag) => (table-name)

Parameters

• diag – An instance of <diagnostic>.

Values

• table-name – An instance of <string>.

text Open Generic function

Returns a string containing the text of an SQL statement.

Signature text (sql-statement) => (sql-statement-text)

Parameters

• sql-statement – An instance of <sql-statement>.

Values

• sql-statement-text – An instance of <string>.

Discussion Returns a string containing the text of sql-statement.

See also

• text-setter

15.19. Database objects and integrity constraints
text-setter Open Generic function
Changes the text of an SQL statement.

Signature  text-setter (new-text sql-statement) => (new-text)
Parameters
  • new-text – An instance of <string>.
  • sql-statement – An instance of <sql-statement>.
Values
  • new-text – An instance of <string>.
Discussion Changes the text of the SQL statement in sql-statement to new-text.
See also
  • text

transaction-mode Generic function
Signature  transaction-mode (object) => (#rest results)
Parameters
  • object – An instance of <object>.
Values
  • #rest results – An instance of <object>.

transaction-mode-setter Generic function
Signature  transaction-mode-setter (value object) => (#rest results)
Parameters
  • value – An instance of <object>.
  • object – An instance of <object>.
Values
  • #rest results – An instance of <object>.

unique-index? Generic function
Signature  unique-index? (object) => (#rest results)
Parameters
  • object – An instance of <object>.
Values
  • #rest results – An instance of <object>.

user Open Generic function
Signature  user (connection) => (user)
Parameters
  • connection – An instance of <connection>.
Values
  • user – An instance of <user>.
with-connection Macro
with-database Macro
with-dbms Macro
with-transaction Macro
The strings library exports definitions for basic string manipulation.

Note:

- This library does not address any higher-level operations such as text formatting or anything that requires semantic knowledge of words, such as `pluralize`.
- Where it makes sense, functions can be applied to either a single character or a string of characters. For example, `lowercase('C') => 'c'` and `lowercase("Foo") => "foo"`.
- Functions are case-sensitive by default. Versions that ignore alphabetic case are named with a trailing “-ic” or “-ic?”.
- Open Dylan doesn’t yet support Unicode. When it does, this library will be updated to support it also.

The strings library was originally defined in DEP-0004. Some additional background material can be found there.

### Contents

- **The strings Module**
  - Character Class Predicates
  - Substring Functions
  - Case Conversion Functions
  - Comparison Functions
  - Miscellaneous Functions
- **Other Useful Functions**
  - Built-In
  - common-extensions Module

## 16.1 The strings Module

### 16.1.1 Character Class Predicates

**alphabetical?** Sealed Generic function

Return `#t` if the argument is alphabetic, else `#f`. 
Signature: \textit{alphabetic? (string-or-character, \#key)} \Rightarrow (alphabetic?)

Parameters:
- \textit{string-or-character} – An instance of \textit{type-union(<string>, <character>)}.

Values:
- \textit{alphabetic?} – An instance of \textit{boolean}.

\textbf{alphabetic? (<character>)} Sealed Method
Returns \#t if the given character is a member of the set a-z or A-Z. Otherwise returns \#f.

Signature: \textit{alphabetic? (character) \Rightarrow (alphabetic?)}

Parameters:
- \textit{character} – An instance of \textit{character}.

Values:
- \textit{alphabetic?} – An instance of \textit{boolean}.

Example:
\begin{verbatim}
alphabetic?('a') \Rightarrow \#t
alphabetic?('-') \Rightarrow \#f
\end{verbatim}

\textbf{alphabetic? (<string>)} Sealed Method
Returns \#t if every character in the string is a member of the set a-z or A-Z. Otherwise returns \#f.

Signature: \textit{alphabetic? (string, \#key start, end) \Rightarrow (alphabetic?)}

Parameters:
- \textit{string} – An instance of \textit{string}.
- \textit{start (\#key)} – Index into \textit{string} at which to start the comparison. An instance of \textit{integer}, default 0.
- \textit{end (\#key)} – Index into \textit{string} at which to stop the comparison. An instance of \textit{integer}, default \textit{string.size}.

Values:
- \textit{alphabetic?} – An instance of \textit{boolean}.

Example:
\begin{verbatim}
alphabetic?("abc") \Rightarrow \#t
alphabetic?("abc123", end: 3) \Rightarrow \#t
alphabetic?("abc123") \Rightarrow \#f
\end{verbatim}

\textbf{alphanumeric? Sealed Generic function}
Returns \#t if the argument is alphanumeric, otherwise \#f.

Signature: \textit{alphanumeric? (string-or-character, \#key)} \Rightarrow (alphanumeric?)

Parameters:
- \textit{string-or-character} – An instance of \textit{type-union(<string>, <character>)}.

Values:
• **alphanumeric?** – An instance of `<boolean>`.

**alphanumeric?(<character>) Sealed Method**

Returns `#t` if the argument is a member of the set of characters a-z, A-Z, or 0-9, otherwise `#f`.

**Signature**  
alphanumeric?(character) => (alphanumeric?)

**Parameters**

• **character** – An instance of `<character>`.

**Values**

• **alphanumeric?** – An instance of `<boolean>`.

**Example**

<table>
<thead>
<tr>
<th>character</th>
<th>alphanumeric?</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>'Z'</td>
<td></td>
<td>#t</td>
</tr>
<tr>
<td>'9'</td>
<td></td>
<td>#t</td>
</tr>
<tr>
<td>'*'</td>
<td></td>
<td>#f</td>
</tr>
</tbody>
</table>

**alphanumeric?(<string>) Sealed Method**

Returns `#t` if every character in the string is a member of the set a-z, A-Z, or 0-9, otherwise `#f`.

**Signature**  
alphanumeric?(string) => (alphanumeric?)

**Parameters**

• **string** – An instance of `<string>`.

**Values**

• **alphanumeric?** – An instance of `<boolean>`.

**Example**

<table>
<thead>
<tr>
<th>string</th>
<th>alphanumeric?</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;abc123&quot;</td>
<td></td>
<td>#t</td>
</tr>
<tr>
<td>&quot;abc...&quot;</td>
<td></td>
<td>#f</td>
</tr>
<tr>
<td>&quot;abc...&quot;, end: 3</td>
<td></td>
<td>#t</td>
</tr>
</tbody>
</table>

**control? Sealed Generic function**

Returns `#t` if the argument is entirely composed of control characters, otherwise `#f`.

**Signature**  
control?(string-or-character, #key) => (control?)

**Parameters**

• **string-or-character** – An instance of type-union(<string>, <character>).

**Values**

• **control?** – An instance of `<boolean>`.

**control?(<character>) Sealed Method**

Returns `#t` if the argument is not a graphic or whitespace character, otherwise `#f`.

**Signature**  
control?(character) => (control?)

**Parameters**

• **character** – An instance of `<character>`.

**Values**
• **control?** – An instance of `<boolean>`.

Example

```
control?('a') => #f
control?('\0') => #t
```

**control? (<string>)** Sealed Method

Returns #t if the argument is entirely composed of non-graphic, non-whitespace characters.

**Signature**  
control? (string) => (control?)

**Parameters**

• **string** – An instance of `<string>`.

• **start (#key)** – Index into string at which to start the comparison. An instance of `<integer>`, default 0.

• **end (#key)** – Index into string at which to stop the comparison. An instance of `<integer>`, default string.size.

**Values**

• **control?** – An instance of `<boolean>`.

Example

```
control?("\0\a\b") => #t
control?("abc\0") => #f
control?("abc\0", start: 3) => #t
```

**graphic?** Sealed Generic function

Returns #t if the argument is entirely composed of graphic characters.

**Signature**  
graphic? (string-or-character, #key) => (graphic?)

**Parameters**

• **string-or-character** – An instance of type-union(<string>, <character>).

**Values**

• **graphic?** – An instance of `<boolean>`.

**graphic? (<character>)** Sealed Method

Returns #t if the argument is a graphic character, defined as those with character codes between 32 (Space) and 126 (~) in the US ASCII character set.

**Signature**  
graphic? (character, #key) => (graphic?)

**Parameters**

• **character** – An instance of `<character>`.

**Values**

• **graphic?** – An instance of `<boolean>`.

Example

```
graphic?('a') => #t
graphic?('') => #f
```
graphic? (<string>) Sealed Method

Returns #t if the argument is entirely composed of graphic characters, defined as those with character codes between 32 (Space) and 126 (~).

Signature  graphic? (string, #key) => (graphic?)

Parameters

- **string** – An instance of <string>.
- **start (#key)** – Index into string at which to start the comparison. An instance of <integer>, default 0.
- **end (#key)** – Index into string at which to stop the comparison. An instance of <integer>, default string.size.

Values

- **graphic?** – An instance of <boolean>.

Example

```
graphic?("ABC") => #t
graphic?("ABC\n") => #f
graphic?("ABC\n", end: 3) => #t
```

printable? Sealed Generic function

Returns #t if the argument is entirely composed of printable characters, defined as either a graphic or whitespace character.

Signature  printable? (string-or-character, #key) => (printable?)

Parameters

- **string-or-character** – An instance of type-union(<string>, <character>).

Values

- **printable?** – An instance of <boolean>.

printable? (<character>) Sealed Method

Returns #t if the argument is a printable character, defined as either a graphic or whitespace character. Otherwise #f is returned.

Signature  printable? (character, #key) => (printable?)

Parameters

- **character** – An instance of <character>.

Values

- **printable?** – An instance of <boolean>.

Example

```
printable?('x') => #t
printable?('	') => #t
printable?('\0') => #f
```

printable? (<string>) Sealed Method

Returns #t if the argument is entirely composed of printable characters, defined as either a graphic or whitespace character. Otherwise #f is returned.
Signature printable? (string, #key) => (printable?)

Parameters

• **string** – An instance of `<string>`.
• **start (#key)** – Index into string at which to start the comparison. An instance of `<integer>`, default 0.
• **end (#key)** – Index into string at which to stop the comparison. An instance of `<integer>`, default string.size.

Values

• **printable?** – An instance of `<boolean>`.

Example

```
printable?("a b c") => #t
printable?("abc\0") => #f
printable?("abc\0", end: 3) => #t
```

---

**whitespace?** Sealed Generic function

Returns #t if the argument is entirely composed of whitespace characters.

Signature whitespace? (string-or-character, #key) => (whitespace?)

Parameters

• **string-or-character** – An instance of type-union(<string>, <character>).

Values

• **whitespace?** – An instance of `<boolean>`.

**whitespace?** (<character>) Sealed Method

Returns #t if the argument is ‘ ‘ (Space), ‘\t’ (Tab), ‘\n’ (Newline), ‘\f’ (Formfeed), or ‘\r’ (Return). Otherwise #f is returned.

Signature whitespace? (character, #key) => (whitespace?)

Parameters

• **character** – An instance of `<character>`.

Values

• **whitespace?** – An instance of `<boolean>`.

Example

```
whitespace?(‘ ‘) => #t
whitespace?(‘\r’) => #t
whitespace?(‘x’) => #f
```

**whitespace?** (<string>) Sealed Method

Returns #t if the argument is entirely composed of whitespace characters, defined as ‘ ‘ (Space), ‘\t’ (Tab), ‘\n’ (Newline), ‘\f’ (Formfeed), or ‘\r’ (Return). Otherwise #f is returned.

Signature whitespace? (string, #key) => (whitespace?)

Parameters
• **string** – An instance of `<string>`.

• **start (#key)** – Index into string at which to start the comparison. An instance of `<integer>`, default 0.

• **end (#key)** – Index into string at which to stop the comparison. An instance of `<integer>`, default string.size.

**Values**

• **whitespace?** – An instance of `<boolean>`.

**Example**

```dylan
whitespace?("x\t x") => #f
whitespace?("x\t x", start: 1, end: 3) => #t
```

decimal-digit? Sealed Generic function

Returns #t if the argument is a decimal digit, otherwise #f.

**Signature**

decimal-digit? (string-or-character, #key) => (decimal-digit?)

**Parameters**

• **string-or-character** – An instance of type-union(<string>, <character>).

**Values**

• **decimal-digit?** – An instance of `<boolean>`.

decimal-digit?(<character>) Sealed Method

Returns #t if the character is a member of the set [0-9], otherwise #f is returned.

**Signature**

decimal-digit? (character, #key) => (decimal-digit?)

**Parameters**

• **character** – An instance of <character>.

**Values**

• **decimal-digit?** – An instance of `<boolean>`.

**Example**

```dylan
decimal-digit?('a') => #f
decimal-digit?('4') => #t
```

decimal-digit?(<string>) Sealed Method

Returns #t if every character in the string is a member of the set [0-9], otherwise #f is returned.

**Signature**

decimal-digit? (string, #key) => (decimal-digit?)

**Parameters**

• **string** – An instance of `<string>`.

• **start (#key)** – Index into string at which to start the comparison. An instance of `<integer>`, default 0.

• **end (#key)** – Index into string at which to stop the comparison. An instance of `<integer>`, default string.size.

**Values**
• **decimal-digit?** – An instance of `<boolean>`.

**Example**

```
decimal-digit?("123") => #t
decimal-digit?("x123y") => #f
decimal-digit?("x123y", start: 1, end: 4) => #t
```

---

**hexadecimal-digit?** Sealed Generic function

Returns #t if the argument is entirely composed of hexadecimal digits, otherwise #f is returned.

**Signature**  
`hexadecimal-digit? (string-or-character, #key) => (hexadecimal-digit?)`

**Parameters**

• **string-or-character** – An instance of `type-union(<string>, <character>)`.

**Values**

• **hexadecimal-digit?** – An instance of `<boolean>`.

**hexadecimal-digit? (<character>)** Sealed Method

Returns #t if the character is a member of the set [0-9a-fA-F], otherwise #f is returned.

**Signature**  
`hexadecimal-digit? (character, #key) => (hexadecimal-digit?)`

**Parameters**

• **character** – An instance of `<character>`.

**Values**

• **hexadecimal-digit?** – An instance of `<boolean>`.

**Example**

```
hexadecimal-digit?(\'a\') => #t
hexadecimal-digit?(\'g\') => #f
hexadecimal-digit?(\'0\') => #t
```

**hexadecimal-digit? (<string>)** Sealed Method

Returns #t if every character in the string is a member of the set [0-9a-fA-F], otherwise #f is returned.

**Signature**  
`hexadecimal-digit? (string, #key) => (hexadecimal-digit?)`

**Parameters**

• **string** – An instance of `<string>`.

• **start (#key)** – Index into `string` at which to start the comparison. An instance of `<integer>`, default 0.

• **end (#key)** – Index into `string` at which to stop the comparison. An instance of `<integer>`, default `string.size`.

**Values**

• **hexadecimal-digit?** – An instance of `<boolean>`.

**Example**
octal-digit? Sealed Generic function
Returns #t if the argument is entirely composed of octal digits, otherwise #f is returned.

Signature  octal-digit? (string-or-character, #key) => (octal-digit?)

Parameters

• string-or-character – An instance of type-union(<string>, <character>).

Values

• octal-digit? – An instance of <boolean>.

octal-digit?(<character>) Sealed Method
Returns #t if the character is a member of the set [0-9a-fA-F], otherwise #f is returned.

Signature  octal-digit? (character, #key) => (octal-digit?)

Parameters

• character – An instance of <character>.

Values

• octal-digit? – An instance of <boolean>.

Example

octal-digit?('7') => #t
octal-digit?('0') => #t
octal-digit?('8') => #f

octal-digit?(<string>) Sealed Method
Returns #t if every character in the string is a member of the set [0-9a-fA-F], otherwise #f is returned.

Signature  octal-digit? (string, #key) => (octal-digit?)

Parameters

• string – An instance of <string>.
• start (#key) – Index into string at which to start the comparison. An instance of <integer>, default 0.
• end (#key) – Index into string at which to stop the comparison. An instance of <integer>, default string.size.

Values

• octal-digit? – An instance of <boolean>.

Example

octal-digit?("700") => #t
octal-digit?("7008") => #f
octal-digit?("7008", end: 3) => #t
16.1.2 Substring Functions

count-substrings Sealed Generic function
Count how many times a substring pattern occurs in a larger string.

Signature  count-substrings (big pattern #key start end ignore-case?) => (count)

Parameters

• big – An instance of <string>. The string in which to search.
• pattern – An instance of <string>. The substring to search for.
• start (#key) – An instance of <integer>, default 0. Where to start searching.
• end (#key) – An instance of <integer>, default big.size. Where to stop searching.
  Note that if pattern is not completely between the bounds of start (inclusive) and end (exclusive) it will not be counted.
• ignore-case? (#key) – An instance of <boolean>, default #f.

Values

• count – An instance of <integer>.

Example

| count-substrings("", ") => 1 |
| count-substrings("xxxxxx", "xx", end: 5) => 2 // no overlap |
| count-substrings("xx", ", ignore-case?: #t) => 3 |

find-substring Sealed Generic function
Find the index of a substring pattern in a larger string. Returns #f if not found.

Signature  find-substring (big pattern #key start end ignore-case?) => (index)

Parameters

• big – An instance of <string>. The string in which to search.
• pattern – An instance of <string>. The substring to search for.
• start (#key) – An instance of <integer>, default 0. Where to start searching.
• end (#key) – An instance of <integer>, default big.size. Where to stop searching.
  Note that if pattern is not completely between the bounds of start (inclusive) and end (exclusive) it will not match.
• ignore-case? (#key) – An instance of <boolean>, default #f.

Values

• index – An instance of false-or(<integer>).

Example

| find-substring("My dog has fleas.", "dog") => 3 |

replace-substrings Sealed Generic function
Replace a substring pattern in a larger string. Allocates a new string for the return value if any replacements are done. If there are no replacements the implementation may return big unmodified.

Signature  replace-substrings (big pattern replacement #key count start end ignore-case?) => (new-string)

Parameters
• **big** – An instance of `<string>`. The string in which to search.

• **pattern** – An instance of `<string>`. The substring pattern to search for.

• **replacement** – An instance of `<string>`. The string with which to replace `pattern`.

• **count (#key)** – An instance of `false-or(<integer>)`. The number of occurrences to replace. The default is #f, meaning to replace all. Replacements are performed from left to right within `big` until `count` has been reached.

• **start (#key)** – An instance of `<integer>`, default 0. Where to start searching.

• **end (#key)** – An instance of `<integer>`, default `big.size`. Where to stop searching. Note that if `pattern` is not completely between the bounds of `start` (inclusive) and `end` (exclusive) it will not be replaced.

• **ignore-case? (#key)** – An instance of `<boolean>`, default #f.

Values

• **new-string** – An instance of `<string>`.  

Example

```dylan
replace-substrings("My cat and your cat", "cat", "dog")
=> "My dog and your dog"
```

### 16.1.3 Case Conversion Functions

#### lowercase Sealed Generic function

Returns a lowercased version of its argument.

**Signature**

`lowercase (string-or-character) => (new-string-or-character)`

**Parameters**

- **string-or-character** – An instance of `type-union(<string>, <character>)`.

**Values**

- **new-string-or-character** – An instance of `type-union(<string>, <character>)`.

#### lowercase(<character>) Sealed Method

If the given character is alphabetic, its lowercase equivalent is returned. Otherwise the character itself is returned.

**Signature**

`lowercase (character) => (new-character)`

**Parameters**

- **character** – An instance of `<character>`.

**Values**

- **lowercase-character** – An instance of `<character>`.

Example

```dylan
lowercase('A') => 'a'
lowercase('#') => '#'
```
**lowercase(<string>) Sealed Method**

Returns a newly allocated string with all uppercase characters converted to lowercase. The implementation may return the given string unchanged if it contains no uppercase characters.

**Signature**

class lowercase (string) => (lowercase-string)

**Parameters**

- **string** – An instance of <string>.
- **start (#key)** – An instance of <integer>, default 0. The index at which to start lowercasing.
- **end (#key)** – An instance of <integer>, default string.size. The index before which to stop lowercasing.

**Values**

- **lowercase-string** – An instance of <string>.

**Example**

```
lowercase("Hack Dylan!") => "hack dylan!"
lowercase("Hack Dylan!", end: 4) => "hack"
```

**lowercase! Sealed Generic function**

**Signature**

class lowercase! (string-or-character) => (new-string-or-character)

**Parameters**

- **string-or-character** – An instance of type-union(<string>, <character>).

**Values**

- **new-string-or-character** – An instance of type-union(<string>, <character>).

**lowercase!(<character>) Sealed Method**

If the given character is alphabetic, its lowercase equivalent is returned. Otherwise the character is returned unchanged. This operation is not a mutation; this method is provided for symmetry with lowercase!(<character>).

**Signature**

class lowercase! (character) => (new-character)

**Parameters**

- **character** – An instance of <character>.

**Values**

- **lowercase-character** – An instance of <character>.

**Example**

```
lowercase!('A') => 'a'
lowercase!('#') => '#'
```

**lowercase!(<string>) Sealed Method**

Mutates the given string such that all uppercase characters are converted to lowercase.

**Signature**

class lowercase! (string) => (string)
Parameters

- **string** – An instance of `<string>`.
- **start (#key)** – An instance of `<integer>`, default 0. The index at which to start lowercasing.
- **end (#key)** – An instance of `<integer>`, default `string.size`. The index before which to stop lowercasing.

Values

- **lowercase-string** – An instance of `<string>`.

Example

```dylan
let text = concatenate("Hack", "Dylan!");
lowercase!(text);
=> "hackdylan!"

text;
=> "hackdylan!"
lowercase!("Hack Dylan!")
=> error, attempt to modify a string constant
```

### lowercase? Sealed Generic function

Returns `#t` if the argument is entirely composed of non-uppercase characters.

**Signature**  
`lowercase? (string-or-character) => (is-lowercase?)`

**Parameters**

- **string-or-character** – An instance of `type-union(<string>, <character>)`.

**Values**

- **is-lowercase?** – An instance of `<boolean>`.

### lowercase?(<character>) Sealed Method

Returns `#t` if the given character is not an uppercase alphabetic. Otherwise `#f` is returned.

**Signature**  
`lowercase? (character) => (is-lowercase?)`

**Parameters**

- **character** – An instance of `<character>`.

**Values**

- **is-lowercase?** – An instance of `<boolean>`.

Example

```dylan
lowercase?('n') => #t
lowercase?('N') => #f
lowercase?('*') => #t
```

### lowercase?(<string>) Sealed Method

Returns `#t` if the argument does not contain any uppercase alphabetic characters. Otherwise `#f` is returned.

**Signature**  
`lowercase? (string) => (is-lowercase?)`

**Parameters**
• **string** – An instance of `<string>`.

• **start (#key)** – An instance of `<integer>`, default 0. The index at which to start checking.

• **end (#key)** – An instance of `<integer>`, default `string.size`. The index before which to stop checking.

**Values**

• **is-lowercase?** – An instance of `<boolean>`.

**Example**

```dylan
lowercase?("Why me?") => #f
lowercase?("Why me?", start: 1) => #t
lowercase?("e.e. cummings") => #t
```

---

**uppercase** Sealed Generic function

Returns an uppercased version of its argument.

**Signature**  `uppercase (string-or-character) => (new-string-or-character)`

**Parameters**

• **string-or-character** – An instance of `type-union(<string>, <character>)`.

**Values**

• **new-string-or-character** – An instance of `type-union(<string>, <character>)`.

**uppercase(<character>)** Sealed Method

If the given character is alphabetic, its uppercase equivalent is returned. Otherwise the character itself is returned.

**Signature**  `uppercase (character) => (new-character)`

**Parameters**

• **character** – An instance of `<character>`.

**Values**

• **uppercase-character** – An instance of `<character>`.

**Example**

```dylan
uppercase('x') => 'X'
uppercase('*') => '*'
```

**uppercase(<string>)** Sealed Method

Returns a newly allocated string with all lowercase alphabetic characters converted to uppercase. The implementation may return the original string unchanged if it contains no lowercase characters.

**Signature**  `uppercase (string) => (uppercase-string)`

**Parameters**

• **string** – An instance of `<string>`.

• **start (#key)** – An instance of `<integer>`, default 0. The index at which to start uppercasing.
• **end** (#key) – An instance of `<integer>`, default `string.size`. The index before which to stop uppercasing.

**Values**

• **uppercase-string** – An instance of `<string>`.

**Example**

<table>
<thead>
<tr>
<th>uppercase(&quot;Hack Dylan!&quot;) =&gt; &quot;HACK DYLAN!&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>uppercase(&quot;Hack Dylan!&quot;, <strong>end</strong>: 4) =&gt; &quot;HACK Dylan!&quot;</td>
</tr>
</tbody>
</table>

### uppercase! Sealed Generic function

**Signature**  uppercase! (string-or-character) => (new-string-or-character)

**Parameters**

• **string-or-character** – An instance of `type-union(<string>, <character>)`.

**Values**

• **new-string-or-character** – An instance of `type-union(<string>, <character>)`.

### uppercase! (<character>) Sealed Method

If the given character is alphabetic, its uppercase equivalent is returned. Otherwise the character is returned unchanged. This operation is not a mutation, but the method is provided for symmetry with `uppercase! (<character>)`.

**Signature**  uppercase! (character) => (uppercase-character)

**Parameters**

• **character** – An instance of `<character>`.

**Values**

• **uppercase-character** – An instance of `<character>`.

**Example**

| uppercase!('t') => 'T' |

### uppercase! (<string>) Sealed Method

Mutates the given string such that all lowercase characters are converted to uppercase.

**Signature**  uppercase! (string) => (uppercase-string)

**Parameters**

• **string** – An instance of `<string>`.

• **start** (#key) – An instance of `<integer>`, default 0. The index at which to start uppercasing.

• **end** (#key) – An instance of `<integer>`, default `string.size`. The index before which to stop uppercasing.

**Values**

• **uppercase-string** – An instance of `<string>`.

**Example**
let text = concatenate("Hack", "Dylan");
upppercase!(text);
=> "HACKDYLAN!"
text;
=> "HACKDYLAN!"
upppercase!("Hack Dylan!")
=> error, attempt to modify a string constant

uppercase? Sealed Generic function
Returns #t if the argument is entirely composed of non-lowercase characters.

Signature uppercase? (string-or-character) => (is-uppercase?)
Parameters
• string-or-character – An instance of type-union(<string>,
  <character>).
Values
• is-uppercase? – An instance of <boolean>.

uppercase?(<character>) Sealed Method
Returns #t if the given character is not a lowercase alphabetic. Otherwise #f is returned.

Signature uppercase? (character) => (is-uppercase?)
Parameters
• character – An instance of <character>.
Values
• is-uppercase? – An instance of <boolean>.
Example

 uppercase?('T') => #t
 uppercase?('t') => #f
 uppercase?('^') => #t

uppercase?(<string>) Sealed Method
Returns #t if the argument does not contain any lowercase alphabetic characters. Otherwise #f is returned.

Signature uppercase? (string) => (is-uppercase?)
Parameters
• string – An instance of <string>.
• start (#key) – An instance of <integer>, default 0. The index at which to start checking.
• end (#key) – An instance of <integer>, default string.size. The index before which to stop checking.
Values
• is-uppercase? – An instance of <boolean>.
Example
### 16.1.4 Comparison Functions

Case insensitive character comparison functions are provided for convenience. (See DEP-0004 for discussion.)

**char-compare Function**

Returns -1 if char1 < char2, 0 if char1 = char2, and 1 if char1 > char2, using *case sensitive* comparison.

**Signature**

```
char-compare (char1 char2) => (result)
```

**Parameters**

- `char1` – An instance of `<character>`.
- `char2` – An instance of `<character>`.

**Values**

- `result` – An instance of `one-of(-1, 0, 1)`.

**Example**

```
char-compare('a', 'b') => -1
char-compare('a', 'a') => 0
char-compare('b', 'a') => 1
char-compare('a', 'B') => 1
```

**char-compare-ic Function**

Returns -1 if char1 < char2, 0 if char1 = char2, and 1 if char1 > char2, using *case insensitive* comparison.

**Signature**

```
char-compare-ic (char1 char2) => (result)
```

**Parameters**

- `char1` – An instance of `<character>`.
- `char2` – An instance of `<character>`.

**Values**

- `result` – An instance of `one-of(-1, 0, 1)`.

**Example**

```
char-compare-ic('a', 'b') => -1
char-compare-ic('a', 'a') => 0
char-compare-ic('b', 'a') => 1
char-compare-ic('a', 'B') => -1
```

**char-equal-ic? Function**

Returns `#t` if char1 and char2 are the same, *ignoring case*. Otherwise `#f` is returned.

**Signature**

```
char-equal-ic? (char1 char2) => (equal?)
```

**Parameters**

- `char1` – An instance of `<character>`.
- `char2` – An instance of `<character>`.

**Values**

- `result` – An instance of `one-of(-1, 0, 1)`.

**Example**

```
char-equal-ic?('a', 'b') => #f
char-equal-ic?('a', 'a') => #t
char-equal-ic?('b', 'a') => #f
char-equal-ic?('a', 'B') => #t
```
- `equal?` – An instance of `<boolean>`.

  **Example**
  ```dylan
  char-equal-ic?('a', 'A') => #t
  ```

**string-compare** Sealed Generic function

Returns -1 if `string1 < string2`, 0 if `string1` and `string2` are the same, and 1 if `string1 > string2`, using *case sensitive* comparison.

**Signature**

\[
\text{string-compare (string1 string2 #key start1 end1 start2 end2 test) => (result)}
\]

**Parameters**

- `string1` – An instance of `<string>`.
- `string2` – An instance of `<string>`.
- `start1 (#key)` – An instance of `<integer>`, default 0. The index in `string1` at which to start the comparison.
- `end1 (#key)` – An instance of `<integer>`, default `string1.size`. The index in `string1` before which to stop the comparison.
- `start2 (#key)` – An instance of `<integer>`, default 0. The index in `string2` at which to start the comparison.
- `end2 (#key)` – An instance of `<integer>`, default `string2.size`. The index in `string2` before which to stop the comparison.
- `test (#key)` – An instance of `<function>`, default `char-compare`.

**Values**

- `result` – An instance of one-of(-1, 0, 1).

  **Example**
  ```dylan
  string-compare("abc", "abc") => 0
  string-compare("the", "them") => -1
  string-compare("beer", "bee") => 1
  ```

**string-equal?** Sealed Generic function

Returns `#t` if `string1` and `string2` are of equal length and contain the same sequence of characters. Otherwise returns `#f`.

**Signature**

\[
\text{string-equal? (string1 string2 #key start1 end1 start2 end2 test) => (equal?)}
\]

**Parameters**

- `string1` – An instance of `<string>`.
- `string2` – An instance of `<string>`.
- `start1 (#key)` – An instance of `<integer>`, default 0. The index in `string1` at which to start the comparison.
- `end1 (#key)` – An instance of `<integer>`, default `string1.size`. The index in `string1` before which to stop the comparison.
- `start2 (#key)` – An instance of `<integer>`, default 0. The index in `string2` at which to start the comparison.
- `end2 (#key)` – An instance of `<integer>`, default `string2.size`. The index in `string2` before which to stop the comparison.
• **test** (#key) – An instance of `<function>`, default char-compare.

Values

• **equal?** – An instance of `<boolean>`.

Example

```
string-equal?("abc", "abc") => #t
string-equal?("ABC", "abc") => #f
string-equal?("the", "them") => #f
string-equal?("the", "them", end2: 3) => #t
```

**string-equal-ic?** Sealed Generic function

Returns #t if string1 and string2 are of equal length and contain the same sequence of characters, ignoring case. Otherwise returns #f.

**Signature**  
string-equal-ic? (string1 string2 #key start1 end1 start2 end2) => (equal?)

**Parameters**

• **string1** – An instance of `<string>`.

• **string2** – An instance of `<string>`.

• **start1 (#key)** – An instance of `<integer>`, default 0. The index in string1 at which to start the comparison.

• **end1 (#key)** – An instance of `<integer>`, default string1.size. The index in string1 before which to stop the comparison.

• **start2 (#key)** – An instance of `<integer>`, default 0. The index in string2 at which to start the comparison.

• **end2 (#key)** – An instance of `<integer>`, default string2.size. The index in string2 before which to stop the comparison.

Values

• **equal?** – An instance of `<boolean>`.

Example

```
string-equal-ic?("ABC", "abc") => #t
string-equal-ic?("the", "them") => #f
string-equal-ic?("The", "them", end2: 3) => #t
```

**string-greater?** Sealed Generic function

Return #t if string1 is greater than string2, using case sensitive comparison.

**Signature**  
string-greater? (string1 string2 #key start1 end1 start2 end2 test) => (greater?)

**Parameters**

• **string1** – An instance of `<string>`.

• **string2** – An instance of `<string>`.

• **start1 (#key)** – An instance of `<integer>`, default 0. The index in string1 at which to start the comparison.

• **end1 (#key)** – An instance of `<integer>`, default string1.size. The index in string1 before which to stop the comparison.

16.1. The strings Module
• **start2 (#key)** – An instance of `<integer>`, default 0. The index in `string2` at which to start the comparison.

• **end2 (#key)** – An instance of `<integer>`, default `string2.size`. The index in `string2` before which to stop the comparison.

• **test (#key)** – An instance of `<function>`, default `char-compare`.

**Values**

• **greater?** – An instance of `<boolean>`.

**Example**

```dylan
string-greater?("dog", "cat") => #t
string-greater?("Dog", "cat") => #f
string-greater?("dogs", "dog") => #t
```

---

**string-greater-ic?** Sealed Generic function

Return `#t` if `string1` is greater than `string2`, using case insensitive comparison.

**Signature**

```
string-greater-ic? (string1 string2 #key start1 end1 start2 end2) => (greater?)
```

**Parameters**

• **string1** – An instance of `<string>.

• **string2** – An instance of `<string>`.

• **start1 (#key)** – An instance of `<integer>`, default 0. The index in `string1` at which to start the comparison.

• **end1 (#key)** – An instance of `<integer>`, default `string1.size`. The index in `string1` before which to stop the comparison.

• **start2 (#key)** – An instance of `<integer>`, default 0. The index in `string2` at which to start the comparison.

• **end2 (#key)** – An instance of `<integer>`, default `string2.size`. The index in `string2` before which to stop the comparison.

**Values**

• **greater?** – An instance of `<boolean>`.

**Example**

```dylan
string-greater-ic?("dog", "cat") => #t
string-greater-ic?("Dog", "cat") => #t
string-greater-ic?("DOGS", "dog") => #t
```

---

**string-less?** Sealed Generic function

Return `#t` if `string1` is less than `string2`, using case sensitive comparison.

**Signature**

```
string-less? (string1 string2 #key start1 end1 start2 end2 test) => (less?)
```

**Parameters**

• **string1** – An instance of `<string>`.

• **string2** – An instance of `<string>`.

• **start1 (#key)** – An instance of `<integer>`, default 0. The index in `string1` at which to start the comparison.
• **end1** (#key) – An instance of `<integer>`, default string1.size. The index in string1 before which to stop the comparison.

• **start2** (#key) – An instance of `<integer>`, default 0. The index in string2 at which to start the comparison.

• **end2** (#key) – An instance of `<integer>`, default string2.size. The index in string2 before which to stop the comparison.

• **test** (#key) – An instance of `<function>`, default char-compare.

Values

• **less?** – An instance of `<boolean>`.

Example

```
(string-less?"dog", "cat") => #f
(string-less?"Dog", "cat") => #t
(string-less?"dogs", "dog") => #f
```

**string-less-ic?** Sealed Generic function

Return #t if string1 is less than string2, using case insensitive comparison.

Signature **string-less-ic?** (string1 string2 #key start1 end1 start2 end2) => (less?)

Parameters

• **string1** – An instance of `<string>`.

• **string2** – An instance of `<string>`.

• **start1** (#key) – An instance of `<integer>`, default 0. The index in string1 at which to start the comparison.

• **end1** (#key) – An instance of `<integer>`, default string1.size. The index in string1 before which to stop the comparison.

• **start2** (#key) – An instance of `<integer>`, default 0. The index in string2 at which to start the comparison.

• **end2** (#key) – An instance of `<integer>`, default string2.size. The index in string2 before which to stop the comparison.

Values

• **less?** – An instance of `<boolean>`.

Example

```
(string-less-ic?"cat", "dog") => #t
(string-less-ic?"cat", "Dog") => #t
(string-less-ic?"dogs", "dog") => #t
```

**starts-with?** Sealed Generic function

Return #t if string1 is starts with string2, using case sensitive comparison.

Signature **starts-with?** (string pattern #key test) => (starts-with?)

Parameters

• **string** – An instance of `<string>`.

• **pattern** – An instance of `<string>`.

16.1. The strings Module
Dylan Library Reference, Release 1.0

• test (#key) – An instance of <function>, default char-compare. For case insensitive comparison pass char-compare-ic here.
Values
• starts-with? – An instance of <boolean>.
Example
starts-with?("Watermelon", "water") => #f
starts-with?("Watermelon", "water", test: char-compare-ic) => #t

ends-with? Sealed Generic function
Return #t if string1 is ends with string2, using case sensitive comparison.
Signature ends-with? (string pattern #key test) => (ends-with?)
Parameters
• string – An instance of <string>.
• pattern – An instance of <string>.
• test (#key) – An instance of <function>, default char-compare. For case insensitive comparison pass char-compare-ic here.
Values
• ends-with? – An instance of <boolean>.
Example
ends-with?("Open Dylan", "dylan") => #f
ends-with?("Open Dylan", "dylan", test: char-compare-ic) => #t

16.1.5 Miscellaneous Functions
pad Sealed Generic function
Add a character to both sides of a string until it reaches the given width.
Signature pad (string width #key fill) => (padded-string)
Parameters
• string – An instance of <string>. The string to pad.
• width – An instance of <integer>. The final width of the result string.
• fill (#key) – An instance of <character>. The character to pad with.
Values
• padded-string – An instance of <string>.
Example
pad("foo", 5) => " foo "
pad("foo", 5, fill: '*') => "*foo*"

pad-left Sealed Generic function
Add a character to the left side of a string until it reaches the given width.
Signature pad-left (string width #key fill) => (padded-string)

408

Chapter 16. The strings Library


Parameters

- **string** – An instance of `<string>`. The string to pad.
- **width** – An instance of `<integer>`. The final width of the result string.
- **fill (#key)** – An instance of `<character>`. The character to pad with.

Values

- **padded-string** – An instance of `<string>`.

Example

```
pad-left("foo", 5) => " foo"
pad-left("foo", 5, fill: 'x') => "xfoo"
```

**pad-right** Sealed Generic function
Add a character to the right side of a string until it reaches the given width.

Signature  
`pad-right (string width #key fill) => (padded-string)`

Parameters

- **string** – An instance of `<string>`. The string to pad.
- **width** – An instance of `<integer>`. The final width of the result string.
- **fill (#key)** – An instance of `<character>`. The character to pad with.

Values

- **padded-string** – An instance of `<string>`.

Example

```
pad-right("foo", 5) => "foo "
pad-right("foo", 5, fill: 'x') => "foox"
```

**split-lines** Function
Split a string on line boundaries, which may be CR alone, CRLF, or LF alone.

Signature  
`split-lines (string #key remove-if-empty?) => (lines)`

Parameters

- **string** – An instance of `<string>`.
- **remove-if-empty? (#key)** – An instance of `<boolean>`. If true, the result will not contain any empty strings.

Values

- **lines** – An instance of `<sequence>`.

Example

```
// Lines are separated by CR, CRLF, or LF, but not LFCR
split-lines("aa\nbb\ncc\rdd\n\nree") => 
  # ["aa", "bb", "cc", "dd", ",", "ree"]

// The end-of-line marker (CR, CRLF, or LF) is considered part
// of the line and is stripped.
split-lines("\nXYZ\n") => 
  # [
    "", "XYZ"
  ]
```

(continues on next page)
See also
- `split`

**strip** Sealed Generic function
Remove characters (whitespace by default) from both sides of a string.

**Signature**
\[
\text{strip (string #key test start end) => (new-string)}
\]

**Parameters**
- `string` – An instance of `<string>`. The string to strip.
- `test` (#`key`) – An instance of `<function>`. A function that accepts a character and returns #t if the character should be removed and #f otherwise.
- `start` (#`key`) – An instance of `<integer>`, default 0. The index in `string` at which to start stripping.
- `end` (#`key`) – An instance of `<integer>`, default `string.size`. The index in `string` before which to stop stripping.

**Values**
- `new-string` – An instance of `<string>`.

**Example**

\[
\begin{align*}
\text{strip(" \	abc\n")} & \Rightarrow \text{"abc"} \\
\text{strip("foo", test: curry(\=, \text{'\*'})} & \Rightarrow \text{"foo"}
\end{align*}
\]

**strip-left** Sealed Generic function
Remove characters (whitespace by default) from the beginning of a string.

**Signature**
\[
\text{strip-left (string #key test start end) => (new-string)}
\]

**Parameters**
- `string` – An instance of `<string>`. The string to strip.
- `test` (#`key`) – An instance of `<function>`. A function that accepts a character and returns #t if the character should be removed and #f otherwise.
- `start` (#`key`) – An instance of `<integer>`, default 0. The index in `string` at which to start stripping.
- `end` (#`key`) – An instance of `<integer>`, default `string.size`. The index in `string` before which to stop stripping.

**Values**
- `new-string` – An instance of `<string>`.

**Example**

\[
\begin{align*}
\text{strip-left(" \	abc\n")} & \Rightarrow \text{"abc\n"} \\
\text{strip-left("foo", test: curry(\=, \text{'\*'})} & \Rightarrow \text{"foo"}
\end{align*}
\]

**strip-right** Sealed Generic function
Remove characters (whitespace by default) from the end of a string.

**Signature**
\[
\text{strip-right (string #key test start end) => (new-string)}
\]

**Parameters**
- `string` – An instance of `<string>`. The string to strip.
- `test` (#`key`) – An instance of `<function>`. A function that accepts a character and returns #t if the character should be removed and #f otherwise.
- `start` (#`key`) – An instance of `<integer>`, default 0. The index in `string` at which to start stripping.
- `end` (#`key`) – An instance of `<integer>`, default `string.size`. The index in `string` before which to stop stripping.

**Values**
- `new-string` – An instance of `<string>`.

**Example**

\[
\begin{align*}
\text{strip-right(" \	abc\n")} & \Rightarrow \text{"abc\n"} \\
\text{strip-right("foo", test: curry(\=, \text{'\*'})} & \Rightarrow \text{"foo"}
\end{align*}
\]
Signature: strip-right (string #key test start end) => (new-string)

Parameters:

- **string** – An instance of `<string>`. The string to strip.
- **test (#key)** – An instance of `<function>`. A function that accepts a character and returns #t if the character should be removed and #f otherwise.
- **start (#key)** – An instance of `<integer>`, default 0. The index in string at which to start stripping.
- **end (#key)** – An instance of `<integer>`, default string.size. The index in string before which to stop stripping.

Values:

- **new-string** – An instance of `<string>`.

Example:

```
strip-right(" \	abc\n") => " \tabc"
strip-right("*foo*", test: curry(\=, '*')) => "*foo"
```

16.2 Other Useful Functions

There are a number of functions outside the strings library itself that can be used with strings.

16.2.1 Built-In

- copy-sequence
- concatenate
- replace-subsequence!
- subsequence-position
- member?
- size
- empty?
- reverse
- reverse!
- as-lowercase
- as-lowercase!
- as-uppercase
- as-uppercase!

16.2.2 common-extensions Module

- concatenate!
- float-to-string
- integer-to-string
- string-to-integer
The System library exports the following modules:

17.1 The date Module

The `date` module is part of the System library and provides a machine-independent facility for representing and manipulating dates and date/time intervals.

**Contents**

- The date Module
  - Representing dates and times
  - Representing durations
  - Performing operations on dates and durations
    - Comparing dates
    - Comparing durations
    - Performing arithmetic operations
    - Dealing with time-zones
  - Extracting information from dates
  - Formatting Dates
  - Parsing Dates
  - The date module

17.1.1 Representing dates and times

The `date` module contains a single class, `<date>`, an instance of which can represent any date between 1 Jan 1800 00:00:00 and 31 Dec 2199 23:59:59, Greenwich Mean Time. You can create a date object by calling the function `encode-date` or using the `make` method for `<date>`. 

- `<date>`
- `make`
- `encode-date`
Each of the arguments to `encode-date` and to the `make` method on `<date>` is an instance of `<integer>` (except for the `iso8601-string` keyword for the `make` method, which is a string) that is passed as an init-keyword value to the `<date>` object. Each init-keyword takes an instance of `<integer>`, limited to the natural range for that attribute. For example, `month:` can only take values between 1 and 12.

You must specify values, via `encode-date`, for at least the `year:`, `month:`, and `day:` init-keywords. In addition, you can also specify values for `hours:`, `minutes:`, `seconds:`, `microseconds:`, and `time-zone-offset:`. If not supplied, the default value for each of these init-keywords is 0.

The `time-zone-offset:` init-keyword is used to represent time zones in the `date` module as `<integer>` values representing the offset in minutes from Greenwich Mean Time (GMT). Positive values are used for time zones East of Greenwich; negative values represent time zones to the west of Greenwich.

For example, the value -300 (-5 hours) is U.S. Eastern Standard Time and the value -240 (-4 hours) is U.S. Eastern Daylight Savings Time.

If you wish, a `<date>` can be specified completely by using the `iso8601-string:` init-keyword. This init-keyword takes an instance of `<string>`, which should be a valid ISO8601 format date. If you use the `iso8601-string:` init-keyword, there is no need to specify any other init-keywords to a call to `make` on `<date>`.

### 17.1.2 Representing durations

Date/time intervals, called durations, are modeled in a style quite similar to that of SQL. There are two, effectively disjoint, classes of duration: one with a resolution of months (for example, 3 years, 2 months) and the other with a resolution of microseconds (for example, 50 days, 6 hours, 23 minutes). The first is `<year/month-duration>` and the second `<day/time-duration>`.

An important distinction between `<day/time-duration>` and `<year/month-duration>` is that a given instance of `<day/time-duration>` is always a fixed unit of a fixed length, whereas a `<year/month-duration>` follows the vagaries of the calendar. So if you have a `<date>` that represents, for example, the 5th of some month, adding a `<year/month-duration>` of 1 month to that will always take you to the 5th of the following month, whether that is an interval of 28, 29, 30, or 31 days.

### 17.1.3 Performing operations on dates and durations

A number of interfaces are exported from the `date` module that let you perform other operations on dates and durations, and extract date-specific information from your local machine.

#### Comparing dates

The following operations are exported from the `date` module.
These methods let you perform arithmetic-like operations on dates to test for equality, or to test whether one date occurred before another.

**Comparing durations**

The following operations are exported from the `date` module.

- `=`
- `<`

As with dates, you can perform arithmetic-like operations on durations to test for equality, or to test whether one duration is shorter than another.

**Performing arithmetic operations**

You can add, subtract, multiply, and divide dates and durations in a number of ways to produce a variety of date or duration information. Methods are defined for any combination of date and duration, with any operation that makes sense, and the return value is of the appropriate type.

For example, a method is defined that subtracts one date from another, and returns a duration, but there is no method for adding two dates together, since dates cannot be summed in any sensible way. However, there are methods for adding dates and durations which return dates.

Note that some addition and subtraction operations involving dates and instances of `<year/month-duration>` can cause errors where the result is a date that does not exist in the calendar. For example, adding one month to January 30th.

The table below summarizes the methods defined for each arithmetic operation, for different combinations of date and duration arguments, together with their return values.

**Methods defined for arithmetic operations on dates and durations**

<table>
<thead>
<tr>
<th>Op</th>
<th>Argument 1</th>
<th>Argument 2</th>
<th>Return value</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td><code>&lt;duration&gt;</code></td>
<td><code>&lt;duration&gt;</code></td>
<td><code>&lt;duration&gt;</code></td>
</tr>
<tr>
<td>+</td>
<td><code>&lt;year/month-duration&gt;</code></td>
<td><code>&lt;year/month-duration&gt;</code></td>
<td><code>&lt;year/month-duration&gt;</code></td>
</tr>
<tr>
<td>+</td>
<td><code>&lt;day/time-duration&gt;</code></td>
<td><code>&lt;day/time-duration&gt;</code></td>
<td><code>&lt;day/time-duration&gt;</code></td>
</tr>
<tr>
<td>+</td>
<td><code>&lt;date&gt;</code></td>
<td><code>&lt;duration&gt;</code></td>
<td><code>&lt;date&gt;</code></td>
</tr>
<tr>
<td>+</td>
<td><code>&lt;duration&gt;</code></td>
<td><code>&lt;date&gt;</code></td>
<td><code>&lt;date&gt;</code></td>
</tr>
<tr>
<td>-</td>
<td><code>&lt;duration&gt;</code></td>
<td><code>&lt;duration&gt;</code></td>
<td><code>&lt;duration&gt;</code></td>
</tr>
<tr>
<td>-</td>
<td><code>&lt;year/month-duration&gt;</code></td>
<td><code>&lt;year/month-duration&gt;</code></td>
<td><code>&lt;year/month-duration&gt;</code></td>
</tr>
<tr>
<td>-</td>
<td><code>&lt;day/time-duration&gt;</code></td>
<td><code>&lt;day/time-duration&gt;</code></td>
<td><code>&lt;day/time-duration&gt;</code></td>
</tr>
<tr>
<td>-</td>
<td><code>&lt;date&gt;</code></td>
<td><code>&lt;duration&gt;</code></td>
<td><code>&lt;date&gt;</code></td>
</tr>
<tr>
<td>-</td>
<td><code>&lt;date&gt;</code></td>
<td><code>&lt;date&gt;</code></td>
<td><code>&lt;day/time-duration&gt;</code></td>
</tr>
<tr>
<td>*</td>
<td><code>&lt;duration&gt;</code></td>
<td><code>&lt;real&gt;</code></td>
<td><code>&lt;duration&gt;</code></td>
</tr>
<tr>
<td>*</td>
<td><code>&lt;real&gt;</code></td>
<td><code>&lt;duration&gt;</code></td>
<td><code>&lt;duration&gt;</code></td>
</tr>
<tr>
<td>/</td>
<td><code>&lt;duration&gt;</code></td>
<td><code>&lt;real&gt;</code></td>
<td><code>&lt;duration&gt;</code></td>
</tr>
</tbody>
</table>

**Dealing with time-zones**

The following functions return information about the time-zone that the host machine is in.
• local-time-zone-name
• local-time-zone-offset
• local-daylight-savings-time?

17.1.4 Extracting information from dates

A number of functions are available to return discrete pieces of information from a specified <date> object. These are useful to allow you to deconstruct a given date in order to retrieve useful information from it.

The most basic way to extract information from a date is to use the function decode-date.

• decode-date

Decodes a <date> into its constituent parts. This function is the companion of encode-date, in that it takes a <date> object and returns all of its constituent parts. Note, however, that in contrast to encode-date, it does not return any millisecond component to the date, but it does return the day of the week of the specified date.

A number of other functions exist to extract individual components from a <date> object. Each of these functions is listed below. Each function takes a single argument, a <date> object, and returns the component of the date referred to in the function name. For example, date-month takes a <date> object as an argument, and returns the month that the date refers to.

• date-year
• date-month
• date-day
• date-day-of-week
• date-hours
• date-minutes
• date-seconds
• date-microseconds
• date-time-zone-offset

For each function except date-day-of-week, the value returned is an instance of <integer>. The date-day-of-week function returns an object of type <day-of-week>. For more information, please refer to the reference entries of each function. Also see the function date-time-zone-offset-setter, which allows you to set the time-zone offset of a <date> explicitly.

17.1.5 Formatting Dates

To return an ISO 8601 format date from a <date> object, use the function as-iso8601-string.

Dates can also be returned in RFC-822 and RFC-1123 formats with the as-rfc822-string and as-rfc1123-string functions.

More flexible date formatting is available with format-date.

17.1.6 Parsing Dates

Dates can be parsed with parse-date-string. ISO-8601 formatted date strings can be parsed with parse-iso8601-string.
17.1.7 The date module

This section contains a reference entry for each item exported from the Date module.

=(<date>) Sealed Method

Compares two dates for equality.

Signature  \( date1 = date2 \Rightarrow equal? \)

Parameters

• \( date1 \) – An instance of \(<date>\).

• \( date2 \) – An instance of \(<date>\).

Discussion This method lets you compare two dates to see if they are equal. Any differences in microseconds between \( date1 \) and \( date2 \) are ignored.

See also

• <

=(<duration>) Sealed Method

Compares two durations for equality.

Signature  \( duration1 = duration2 \Rightarrow equal? \)

Parameters

• \( duration1 \) – An instance of \(<duration>\).

• \( duration2 \) – An instance of \(<duration>\).

Discussion This method lets you compare two durations to see if they are equal. If the durations are actually instances of \(<day/time-duration>\), any differences in microseconds between \( duration1 \) and \( duration2 \) are ignored.

See also

• <

<(<date>) Sealed Method

Determines whether one date is earlier than another.

Signature  \( date1 < date2 \Rightarrow before? \)

Parameters

• \( date1 \) – An instance of \(<date>\).

• \( date2 \) – An instance of \(<date>\).

Discussion This method determines if \( date1 \) is earlier than \( date2 \). Any differences in microseconds between \( date1 \) and \( date2 \) are ignored.

See also
• =

<(<duration>) Sealed Method
Determines whether one duration is less than another.

Signature  duration1 < duration2 => less-than?

Parameters
• duration1 – An instance of <duration>.
• duration2 – An instance of <duration>.

Values
• less-than? – An instance of <boolean>.

Discussion This method determines if duration1 is less than duration2. If the durations are actually
instances of <day/time-duration>, any differences in microseconds between duration1
and duration2 are ignored.

See also
• =

+(<date>) Sealed Method
Performs addition on specific combinations of dates and durations.

Signature
• arg1 arg2 => sum

Parameters
• arg1 – An instance of <date> or <duration>. See description for details.
• arg2 – An instance of <date> or <duration>. See description for details.

Values
• sum – An instance of <date>.

Discussion
A number of methods are defined for the + generic function to allow summing of various com-
binations of dates and durations. Note that there is not a method defined for every possible
combination of date and duration. Specifically, you cannot sum two dates. The table below lists
the methods that are defined on + and <date>.

Methods defined for addition of dates and durations

<table>
<thead>
<tr>
<th>arg1</th>
<th>arg2</th>
<th>sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;date&gt;</td>
<td>&lt;duration&gt;</td>
<td>&lt;date&gt;</td>
</tr>
<tr>
<td>&lt;duration&gt;</td>
<td>&lt;date&gt;</td>
<td>&lt;date&gt;</td>
</tr>
</tbody>
</table>

See also
• +(<duration>)
• -(<date>)
• -(<duration>)
• *(<duration>)
• /(<duration>)
+(<duration>) Sealed Method
Performs addition on durations.

Signature

- \( arg1 \ arg2 \rightarrow sum \)

Parameters

- \( arg1 \) – An instance of \(<duration>\).
- \( arg2 \) – An instance of \(<duration>\).

Values

- \( sum \) – An instance of \(<duration>\).

Discussion

A number of methods are defined for the + generic function to allow summing of durations. Note that there is not a method defined for every possible combination of duration. Specifically, you cannot sum different types of duration. The table below lists the methods that are defined on +.

Methods defined for addition of durations

<table>
<thead>
<tr>
<th>arg1</th>
<th>arg2</th>
<th>sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>(&lt;duration&gt;)</td>
<td>(&lt;duration&gt;)</td>
<td>(&lt;duration&gt;)</td>
</tr>
<tr>
<td>(&lt;year/month-duration&gt;)</td>
<td>(&lt;year/month-duration&gt;)</td>
<td>(&lt;year/month-duration&gt;)</td>
</tr>
<tr>
<td>(&lt;day/time-duration&gt;)</td>
<td>(&lt;day/time-duration&gt;)</td>
<td>(&lt;day/time-duration&gt;)</td>
</tr>
</tbody>
</table>

See also

- \(+(<date>)\)
- \(-(<date>)\)
- \-(<duration>)
- \(+(<duration>)\)
- \(/(<duration>)\)

-(<date>) Sealed Method
Performs subtraction on specific combinations of dates and durations.

Signature

- \( arg1 \ arg2 \rightarrow diff \)

Parameters

- \( arg1 \) – An instance of \(<date>\) or \(<duration>\). See description for details.
- \( arg2 \) – An instance of \(<duration>\), or an instance of \(<date>\) if \( arg1 \) is a \(<date>\). See description for details.

Values

- \( diff \) – An instance of \(<date>\) or \(<duration>\). See description for details.

Discussion

A number of methods are defined for the – generic function to allow subtraction of various combinations of dates and durations. Note that there is not a method defined for every possible
combination of date and duration. Specifically, you cannot subtract a date from a duration, and you cannot subtract different types of duration. The return value can be either a date or a duration, depending on the arguments supplied. The table below lists the methods that are defined on –.

Methods defined for subtraction of dates and durations

<table>
<thead>
<tr>
<th>arg1</th>
<th>arg2</th>
<th>difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;date&gt;</td>
<td>&lt;duration&gt;</td>
<td>&lt;date&gt;</td>
</tr>
<tr>
<td>&lt;date&gt;</td>
<td>&lt;date&gt;</td>
<td>&lt;day/time-duration&gt;</td>
</tr>
</tbody>
</table>

See also

- +(<date>)
- +(<duration>)
- −(<duration>)
- *(<duration>)
- /(<duration>)

-(<duration>) Sealed Method

Performs subtraction on specific combinations of durations.

Signature

• arg1 arg2 => diff

Parameters

• arg1 – An instance of <date> or <duration>. See description for details.
• arg2 – An instance of <duration>, or an instance of <date> if arg1 is a <date>. See description for details.

Values

• diff – An instance of <date> or <duration>. See description for details.

Discussion

A number of methods are defined for the – generic function to allow subtraction of durations. Note that there is not a method defined for every possible combination of duration. Specifically, you cannot subtract different types of duration. The table below lists the methods that are defined on –.

Methods defined for subtraction of dates and durations

<table>
<thead>
<tr>
<th>arg1</th>
<th>arg2</th>
<th>difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;year/month-duration&gt;</td>
<td>&lt;year/month-duration&gt;</td>
<td>&lt;year/month-duration&gt;</td>
</tr>
<tr>
<td>&lt;day/time-duration&gt;</td>
<td>&lt;day/time-duration&gt;</td>
<td>&lt;day/time-duration&gt;</td>
</tr>
</tbody>
</table>

See also

- +(date)
- +(duration)
- -(date)
• *(<duration>)
• /(<duration>)

*<duration>* Sealed Method
Multiplies a duration by a scale factor.

Signature  * duration scale => new-duration
Signature  * scale duration => new-duration

Parameters
• duration – An instance of <duration>.
• scale – An instance of <real>.

Values
• new-duration – An instance of <duration>.

Discussion

Note: These arguments can be expressed in any order.

Multiples a duration by a scale factor and returns the result. Note that the arguments can be expressed in any order: methods are defined such that the duration can be placed first or second in the list of arguments.

See also
• +(<date>)
• +(<duration>)
• -(<date>)
• -(<duration>)
• *(<duration>)

/(<duration>) Sealed Method
Divides a duration by a scale factor

Signature  / duration scale => new-duration

Parameters
• duration – An instance of <duration>.
• scale – An instance of <real>.

Values
• new-duration – An instance of <duration>.

Discussion

See also
• +(<date>)
• +(<duration>)
• -(<date>)
• -(<duration>)
• *(<duration>)
as-iso8601-string Function
Returns a string representation of a date, conforming to the ISO 8601 standard.

Signature  as-iso8601-string date #key precision => iso8601-string

Parameters
- date – An instance of <date>.
- precision (#key) – An instance of <integer>. Default value: 0.

Values
- iso8601-string – An instance of <string>.

Discussion
Returns a string representation of date using the format identified by International Standard ISO 8601 (for example, "19960418T210634Z"). If precision is non-zero, the specified number of digits of a fraction of a second are included in the string (for example, "19960418T210634.0034Z").

The returned string always expresses the time in Greenwich Mean Time. The iso8601-string init-keyword for <date>, however, accepts ISO 8601 strings with other time zone specifications.

This is the same as calling:

```
format-date("%Y-%m-%dT%H:%M:%S%:z", date);
```

See also
- <date>
- as-rfc822-string
- as-rfc1123-string
- format-date
- parse-date-string
- parse-iso8601-string

as-rfc822-string Function
Returns a string representation of a date, conforming to RFC 822.

Signature  as-rfc822-string date => rfc822-string

Parameters
- date – An instance of <date>.

Values
- rfc822-string – An instance of <string>.

Discussion
An example in this format is:

```
Sun, 01 Sep 13 17:00:00 GMT
```

This is the same as calling:

```
format-date("%a, %d %b %y %H:%M:%S %z", date);
```

See also
as-rfc1123-string Function

Returns a string representation of a date, conforming to RFC 1123.

Signature  
as-rfc1123-string  
date  
=>  
rfc1123-string

Parameters

•  
date  – An instance of <date>.

Values

•  
rfc1123-string  – An instance of <string>.

Discussion

The date format for RFC-1123 is the same as for RFC-822 except that the year must be 4 digits rather than 2:

Sun, 01 Sep 2013 17:00:00 GMT

This is the same as calling:

format-date("%a, %d %b %Y %H:%M:%S %z", date);

This format is commonly used in email, HTTP headers, RSS feeds and other protocols where date representations are used.

See also

•  
<date>

•  
as-rfc822-string

•  
as-iso8601-string

•  
format-date

•  
parse-date-string

current-date Function

Returns a date object representing the current date and time.

Signature  
current-date ()  
=>  
date

Values

•  
date  – An instance of <date>.

Discussion  
Returns date for the current date and time.

<date> Sealed Class

The class of objects representing dates.

Superclasses  
<number>

Init-Keywords

•  
iso8601-string  – An instance of false-or(<string>). Default value: #f.
• **year** – An instance of `limited(<integer>, min: 1800, max: 2199)`.
• **month** – An instance of `limited(<integer>, min: 1, max: 12)`.
• **day** – An instance of `limited(<integer>, min: 1, max: 31)`.
• **hours** – An instance of `limited(<integer>, min: 0, max: 23)`. Default value: 0.
• **minutes** – An instance of `limited(<integer>, min: 0, max: 59)`. Default value: 0.
• **seconds** – An instance of `limited(<integer>, min: 0, max: 59)`. Default value: 0.
• **microseconds** – An instance of `limited(<integer>, min: 0, max: 999999)`. Default value: 0.
• **time-zone-offset** – An instance of `<integer>`. Default value: 0.

**Discussion**

Represents a date and time between 1 Jan 1800 00:00:00 and 31 Dec 2199 23:59:59, Greenwich Mean Time (GMT).

A `<date>` can be specified to microsecond precision and includes a time zone indication.

If supplied, the `iso8601-string` init-keyword completely specifies the value of the `<date>`. Otherwise, the `year`, `month`, and `day` init-keywords must be supplied. Note that, although you can supply ISO 8601 strings that represent any time zone specification, the related function `as-iso8601-string` always returns an ISO 8601 string representing a time in Greenwich Mean Time.

For the `time-zone-offset` init-keyword, a positive number represents an offset ahead of GMT, in minutes, and a negative number represents an offset behind GMT. The value returned is an instance of `<integer>` (for example, -300 represents the offset for EST, which is 5 hours behind GMT).

**Operations**

- `=`
- `<`
- `+`
- `-`
- `as-iso8601-string`
- `as-rfc822-string`
- `as-rfc1123-string`
- `current-date`
- `date-day`
- `date-day-of-week`
- `date-hours`
- `date-microseconds`
- `date-minutes`
- `date-month`
• date-seconds
• date-time-zone-offset
• date-time-zone-offset-setter
• date-year
• decode-date

See also
• as-iso8601-string
• as-rfc822-string
• as-rfc1123-string
• <day-of-week>

date-day Function
Returns the day of the month component of a specified date.

Signature  date-day date => day

Parameters
• date – An instance of <date>.

Values
• day – An instance of <integer>.

Discussion  Returns the day of the month component of the specified date. For example, if passed a <date> that represented 16:36 on the 20th June, 1997, date-day returns the value 20.

See also
• decode-date
• date-month
• date-year
• date-hours
• date-minutes
• date-seconds
• date-microseconds
• date-time-zone-offset
• date-day-of-week

date-day-of-week Function
Returns the day of the week of a specified date.

Signature  date-day-of-week date => day-of-week

Parameters
• date – An instance of <date>.

Values
• day-of-week – An object of type <day-of-week>.

Discussion  Returns the day of the week of the specified date.
See also

- `decode-date`
- `date-month`
- `date-year`
- `date-hours`
- `date-minutes`
- `date-seconds`
- `date-microseconds`
- `date-time-zone-offset`
- `date-day`
- `<day-of-week>`

**date-hours Function**

Returns the hour component of a specified date.

**Signature**

```
date-hours date => hour
```

**Parameters**

- `date` – An instance of `<date>`.

**Values**

- `hour` – An instance of `<integer>`.

**Discussion**

Returns the hour component of the specified `date`. This component is always expressed in 24 hour format.

See also

- `decode-date`
- `date-month`
- `date-day`
- `date-year`
- `date-minutes`
- `date-seconds`
- `date-microseconds`
- `date-time-zone-offset`
- `date-day-of-week`

**date-microseconds Function**

Returns the microseconds component of a specified date.

**Signature**

```
date-microseconds date => microseconds
```

**Parameters**

- `date` – An instance of `<date>`.

**Values**

- `microseconds` – An instance of `<integer>`.
Dylan Library Reference, Release 1.0

**Discussion**  Returns the microseconds component of the specified `date`. Note that this does *not* return the entire date object, represented as a number of microseconds; it returns any value assigned to the *microseconds*: init-keyword when the `<date>` object was created.

See also
- `decode-date`
- `date-month`
- `date-day`
- `date-hours`
- `date-minutes`
- `date-seconds`
- `date-year`
- `date-time-zone-offset`
- `date-day-of-week`

**`date-minutes Function`**

Returns the minutes component of a specified date.

**Signature**  `date-minutes date => minutes`

**Parameters**
- `date` – An instance of `<date>`.

**Values**
- `minutes` – An instance of `<integer>`.

**Discussion**  Returns the minutes component of the specified `date`.

See also
- `decode-date`
- `date-month`
- `date-day`
- `date-hours`
- `date-year`
- `date-seconds`
- `date-microseconds`
- `date-time-zone-offset`
- `date-day-of-week`

**`date-month Function`**

Returns the month of a specified date.

**Signature**  `date-month date => month`

**Parameters**
- `date` – An instance of `<date>`.

**Values**
• **month** – An instance of `<integer>`.

**Discussion** Returns the month of the specified `date`.

**See also**
- `decode-date`
- `date-year`
- `date-day`
- `date-hours`
- `date-minutes`
- `date-seconds`
- `date-microseconds`
- `date-time-zone-offset`
- `date-day-of-week`

### `date-seconds` Function

Returns the seconds component of a specified date.

**Signature** `date-seconds date => seconds`

**Parameters**
- `date` – An instance of `<date>`.

**Values**
- `seconds` – An instance of `<integer>`.

**Discussion** Returns the seconds component of the specified `date`. Note that this does not return the entire date object, represented as a number of seconds; it returns any value assigned to the `seconds` init-keyword when the `<date>` object was created.

**See also**
- `decode-date`
- `date-month`
- `date-day`
- `date-hours`
- `date-minutes`
- `date-year`
- `date-microseconds`
- `date-time-zone-offset`
- `date-day-of-week`

### `date-time-zone-offset` Function

Returns the time zone offset of a specified date.

**Signature** `date-time-zone-offset date => time-zone-offset`

**Parameters**
- `date` – An instance of `<date>`.
Values

- `time-zone-offset` – An instance of `<integer>`.

Discussion

Returns the time zone offset of the specified `date`. The values of the other components of `date` reflect this time zone.

A positive number represents an offset ahead of GMT, in minutes, and a negative number represents an offset behind GMT. The value returned is an instance of `<integer>` (for example, -300 represents the offset for EST, which is 5 hours behind GMT).

See also

- `decode-date`
- `date-month`
- `date-day`
- `date-hours`
- `date-minutes`
- `date-seconds`
- `date-year`
- `date-microseconds`
- `date-time-zone-offset-setter`
- `date-day-of-week`

`date-time-zone-offset-setter` Function

Change the time zone offset of a specified date, while maintaining the same point in time.

**Signature**

`date-time-zone-offset-setter new-time-zone-offset date => new-time-zone-offset`

**Parameters**

- `new-time-zone-offset` – An instance of `<integer>`.`
- `date` – An instance of `<date>`.

**Values**

- `new-time-zone-offset` – An instance of `<integer>`.

**Discussion**

Changes the time zone offset of `date` without changing the actual point in time identified by the `date`. The values of the other components of `date` are adjusted to reflect the new time zone.

The `new-time-zone-offset` argument should represent the offset from GMT, in minutes. Thus, if you wish to specify a new offset representing EST, which is 5 hours behind GMT, `new-time-zone-offset` should have the value -300.

**See also**

- `date-time-zone-offset`

`date-year` Function

Returns the year of a specified date.

**Signature**

`date-year date => year`

**Parameters**
• **date** – An instance of `<date>`.

**Values**

• **year** – An instance of `<integer>`.

**Discussion**  Returns the year of the specified `date`.

**See also**

• `decode-date`
• `date-month`
• `date-day`
• `date-hours`
• `date-minutes`
• `date-seconds`
• `date-microseconds`
• `date-time-zone-offset`
• `date-day-of-week`

:: todo Make this a type.

**<day-of-week> Class**

The days of the week.

**Discussion**

The days of the week. This is the type of the return value of the `date-day-of-week` function.

```plaintext
one-of(#"Sunday", #"Monday", #"Tuesday", #"Wednesday", #"Thursday", #
˓→"Friday", #"Saturday")
```

**Operations**

• `date-day-of-week`

**See also**

• `date-day-of-week`

**<day/time-duration> Sealed Class**

The class of objects representing durations in units of microseconds.

**Superclasses** `<duration>`

**Init-Keywords**

• **days** – An instance of `<integer>`.
• **hours** – An instance of `<integer>`. Default value: 0.
• **minutes** – An instance of `<integer>`.
• **seconds** – An instance of `<integer>`.
• **microseconds** – An instance of `<integer>`.

**Discussion**

The class of objects representing durations in units of microseconds. It is a subclass of `<duration>`.
Use this class to represent a number of days and fractions thereof. If you need to represent durations in calendar units of months or years, use `<year/month-duration>` instead.

Operations

- `<`
- `+`
- `−`
- `decode-duration`
- `encode-day/time-duration`

See also

- `<duration>`
- `<year/month-duration>`

decode-date Function

Returns the date and time stored in a date object.

Signature: `decode-date date => year month day hours minutes seconds day-of-week time-zone-offset`

Parameters

- `date` – An instance of `<date>`.

Values

- `year` – An instance of `<integer>`.
- `month` – An instance of `<integer>`.
- `day` – An instance of `<integer>`.
- `hours` – An instance of `<integer>`.
- `minutes` – An instance of `<integer>`.
- `seconds` – An instance of `<integer>`.
- `day-of-week` – An instance of `<day-of-week>`.
- `time-zone-offset` – An instance of `<integer>`.

Discussion

Returns the date and time stored in `date`.

Note that it does not return the millisecond component of a `<date>`, but it does return the appropriate `<day-of-week>`.

See also

- `encode-date`

decode-duration Sealed Generic function

Decodes a duration into its constituent parts.

Signature: `decode-duration duration => #rest components`

Parameters

- `duration` – An instance of `<duration>`.

Values
• #rest components – Instances of `<integer>`.

**Discussion** Decodes an instance of `<duration>` into its constituent parts. There are methods for this generic function that specialize on `<year/month-duration>` and `<day/time-duration>` respectively, as described below.

**See also**
- `decode-duration`
- `decode-duration`

`decode-duration(<day/time-duration>)` Sealed Method
Decodes a day/time duration into its constituent parts.

**Signature** `decode-duration duration => days hours minutes seconds microseconds`

**Parameters**
- `duration` – An instance of `<day/time-duration>`.

**Values**
- `days` – An instance of `<integer>`.
- `hours` – An instance of `<integer>`.
- `minutes` – An instance of `<integer>`.
- `seconds` – An instance of `<integer>`.
- `microseconds` – An instance of `<integer>`.

**Discussion** Decodes an instance of `<day/time-duration>` into its constituent parts.

**See also**
- `decode-duration`
- `decode-duration`

`decode-duration(<year/month-duration>)` Sealed Method
Decodes a year/month duration into its constituent parts.

**Signature** `decode-duration duration => years months`

**Parameters**
- `duration` – An instance of `<year/month-duration>`.

**Values**
- `years` – An instance of `<integer>`.
- `months` – An instance of `<integer>`.

**Discussion** Decodes an instance of `<year/month-duration>` into its constituent parts.

**See also**
- `decode-duration`
- `decode-duration`

`<duration>` Abstract Instantiable Sealed Class
The class of objects representing durations.
Superclasses  <number>

Init-Keywords

- **iso8601-string** – An instance of false-or(<string>). Default value: #f.
- **year** – An instance of limited(<integer>, min: 1800, max: 2199).
- **month** – An instance of limited(<integer>, min: 1, max: 12).
- **day** – An instance of limited(<integer>, min: 1, max: 31).
- **hours** – An instance of limited(<integer>, min: 0, max: 23). Default value: 0.
- **minutes** – An instance of limited(<integer>, min: 0, max: 59). Default value: 0.
- **seconds** – An instance of limited(<integer>, min: 0, max: 59). Default value: 0.
- **microseconds** – An instance of limited(<integer>, min: 0, max: 999999). Default value: 0.
- **time-zone-offset** – An instance of <integer>. Default value: 0.

Discussion  This class is the used to represent durations. It is a subclass of <number>, and it has two subclasses.

Operations

- =
- <
- +
- –
- *
- /

See also

- <day/time-duration>
- <year/month-duration>

**encode-date Function**

Creates a date object for the specified date and time.

**Signature** encode-date year month day hours minutes seconds #key*microseconds time-zone-offset* => date

**Parameters**

- **year** – An instance of <integer>.
- **month** – An instance of <integer>.
- **day** – An instance of <integer>.
- **hours** – An instance of <integer>.
- **minutes** – An instance of <integer>.
- **seconds** – An instance of <integer>. 

17.1. The date Module
• **microseconds** (\#key) – An instance of `<integer>`. Default value: 0.

• **\#keytime-zone-offset** – An instance of `<integer>`. Default value: `local-time-zone-offset()`.

**Values**

• **date** – An instance of `<date>.

**Discussion** Creates a `<date>` object for the specified date and time.

See also

• **decode-date**

• **local-time-zone-offset**

• **make**

**encode-day/time-duration Function**

Creates a day/time duration from a set of integer values.

**Signature** `encode-day/time-duration days hours minutes seconds microseconds => duration`

**Parameters**

• **days** – An instance of `<integer>`.

• **hours** – An instance of `<integer>`.

• **minutes** – An instance of `<integer>`.

• **seconds** – An instance of `<integer>`.

• **microseconds** – An instance of `<integer>`.

**Values**

• **duration** – An instance of `<day/time-duration>`.

**Discussion** Creates an instance of `<day/time-duration>`.

See also

• **decode-duration**

• **encode-year/month-duration**

**encode-year/month-duration Function**

Creates a year/month duration from a set of integer values.

**Signature** `encode-year/month-duration years months => duration`

**Parameters**

• **years** – An instance of `<integer>`.

• **months** – An instance of `<integer>`.

**Values**

• **duration** – An instance of `<year/month-duration>`.

**Discussion** Creates an instance of `<year/month-duration>`.

See also

• **decode-duration**

• **encode-day/time-duration**
**format-date Function**

Formats a date according to a format string.

**Signature**

`format-date format date => formatted-date`

**Parameters**

- `format` – An instance of `<byte-string>`.
- `date` – An instance of `<date>`.

**Values**

- `formatted-date` – An instance of `<byte-string>`.

**Discussion**

`format-date` interprets a control string, `format`, to create a string representing the `date`.

The control string can contain these directives:

- `%Y` - The year.
- `%y` - The year, in 2 digit form.
- `%H` - Hours, zero padded.
- `%k` - Hours, space padded.
- `%M` - Minutes, zero padded.
- `%S` - Seconds, zero padded.
- `%f` - Microseconds, 6 digits.
- `%F` - Milliseconds, 3 digits.
- `%T` - Time, each component zero padded.
- `%m` - Month in numeric form, zero padded.
- `%d` - Day of the month, zero padded.
- `%e` - Day of the month, space padded.
- `%A` - Name of the day of the week.
- `%a` - Short name of the day of the week.
- `%B` - Name of the month.
- `%b` - Short name of the month.
- `%z` - Time zone offset.
- `:%z` - Time zone offset, but using : between hours and minutes.
- `%n` - A new line.
- `% ` - A % character.

**See also**

- `as-rfc822-string`
- `as-rfc1123-string`
- `as-iso8601-string`
- `parse-date-string`
• `parse-iso8601-string`

**local-daylight-savings-time? Function**
Checks whether the local machine is using Daylight Savings Time.

**Signature**
`local-daylight-savings-time? () => dst?`

**Values**
- `dst?` – An instance of `<boolean>`.

**Discussion**
Returns `#t` if the local machine is using Daylight Savings Time, and `#f` otherwise.

**local-time-zone-name Function**
Returns the time zone name in use by the local machine.

**Signature**
`local-time-zone-name () => time-zone-name`

**Values**
- `time-zone-name` – An instance of `<string>`.

**Discussion**
Returns the time zone name in use by the local machine, if available, or a string of the form `+/-HHMM` if the time zone name is unknown.

**local-time-zone-offset Function**
Returns the offset of the time-zone from Greenwich Mean Time, expressed as a number of minutes.

**Signature**
`local-time-zone-offset () => time-zone-offset`

**Values**
- `time-zone-offset` – An instance of `<integer>`.

**Discussion**
Returns the offset of the time-zone from Greenwich Mean Time, expressed as a number of minutes. A positive number represents an offset ahead of GMT, and a negative number represents an offset behind GMT. The return value is an instance of `<integer>` (for example, -300 represents the offset for EST, which is 5 hours behind GMT). The return value incorporates daylight savings time when necessary.

**make(<date>) Method**
Creates an instance of the `<date>` class.

**Signature**
`make date-class #key iso8601-string year month day hours minutes seconds microseconds time-zone-offset => date-instance`

**Parameters**
- `date-class` – The class `<date>.
- `iso8601-string` (#key) – An instance of `false-or(<string>)`. Default value: `#f`
- `year` (#key) – An instance of `limited(<integer>, min: 1800, max: 2199)`.
- `month` (#key) – An instance of `limited(<integer>, min: 1, max: 12)`.
- `day` (#key) – An instance of `limited(<integer>, min: 0, max: 31)`.
- `hours` (#key) – An instance of `limited(<integer>, min: 0, max: 23)`.
- `minutes` (#key) – An instance of `limited(<integer>, min: 0, max: 59)`.

Default value: 0.
• **seconds** (#key) – An instance of limited(<integer>, min: 0, max: 59). Default value: 0.

• **microseconds** (#key) – An instance of limited(<integer>, min: 0, max: 999999). Default value: 0.

• **time-zone-offset** (#key) – An instance of <integer>. Default value: 0.

**Values**

• **date-instance** – An instance of <date>.

**Discussion**

Creates an instance of <date>.

The make method on <date> takes the same keywords as the <date> class.

**Note:** The iso8601-string keyword accepts a richer subset of the ISO 8601 specification than is produced by the `as-iso8601-string` function. See `parse-iso8601-string` for details.

**Example**

```dylan
make (<date>, iso8601-string: "19970717T1148-0400")
```

See also

• <date>

• encode-date

• parse-iso8601-string

**parse-date-string Function**

Parse a date in string form according to a control string, returning a date.

**Signature**

```dylan
parse-date-string date format => date
```

**Parameters**

• **date** – An instance of <string>.

• **format** – An instance of <string>.

**Values**

• **date** – An instance of <date>.

**Discussion** Parses a date in string form according to a control string, `format`. The control string uses the same directives as `format-date`.

See also

• `format-date`

• `parse-iso8601-string`

**parse-iso8601-string Function**

Parse a variety of variants on ISO-8601 formatted date strings.

**Signature**

```dylan
parse-iso8601-string string #key strict? => date
```

**Parameters**

• **string** – An instance of <string>.

• **strict?** (#key) – An instance of <boolean>, default #t.
Values

• **date** – An instance of `<date>`.

Discussion

This function parses the ISO 8601 formats listed below, with the following differences if strict?

- `#f`:
  - the ‘-’ in YYYY-MM-DD is optional
  - the ‘:’ in hh:mm:ss is optional
  - the ‘:’ in the timezone is optional
  - the date and time may be separated by a space character
  - TZD may be preceded by a space character

See [http://www.w3.org/TR/NOTE-datetime.html](http://www.w3.org/TR/NOTE-datetime.html).

**Year**: YYYY (eg 1997)

**Year and month**: YYYY-MM (eg 1997-07)

**Complete date**: YYYY-MM-DD (eg 1997-07-16)

**Complete date plus hours and minutes**: YYYY-MM-DDThh:mmTZD (eg 1997-07-16T19:20+01:00)

**Complete date plus hours, minutes and seconds**: YYYY-MM-DDThh:mm:ssTZD (eg 1997-07-16T19:20:30+01:00)

**Complete date plus hours, minutes, seconds and a decimal fraction of a second**: YYYY-MM-DDThh:mm:ss.sTZD (eg 1997-07-16T19:20:30.45+01:00)

where:

- YYYY = four-digit year
- MM = two-digit month (01=January, etc.)
- DD = two-digit day of month (01 through 31)
- hh = two digits of hour (00 through 23) (am/pm NOT allowed)
- mm = two digits of minute (00 through 59)
- ss = two digits of second (00 through 59)
- s = one or more digits representing a decimal fraction of a second
- TZD = time zone designator (Z or +hh:mm or -hh:mm)

See also

- `as-iso8601-string`
- `format-date`
- `parse-date-string`

### `<year/month-duration>` Sealed Class

The class of objects representing durations with a coarse resolution.

**Superclasses** `<duration>`

**Init-Keywords**

- **year** – An instance of `<integer>`.
• **month** – An instance of `<integer>`.

### Discussion

The class of objects representing durations in units of calendar years and months. It is a subclass of `<duration>`.

Use this class to represent a number of calendar years and months. If you need to represent durations in units of days or fractions thereof (to microsecond resolution), use `<day/time-duration>` instead.

### Operations

- `<`
- `+`
- `–`
- `decode-duration`
- `encode-year/month-duration`

### See also

- `<day/time-duration>`
- `<duration>`

## 17.2 The locators Module

The Locators module provides Dylan programs with a portable, flexible, and uniform facility for locating files and other objects on the internet.

### 17.2.1 The locators Module

**<locator> Open Abstract Class**

**Superclasses** `<object>`

This is the base class for all locators. This is the usual locator class for coercion (using `as`) or instantiation (using `make`) of new locators. Situations where this class is not appropriate are ones where there is not enough information provided to select the appropriate concrete class. For example, it is not appropriate to coerce a string representing a portion of a URL without a scheme, such as `as(<locator>, "toothpaste.html")`, because this would likely result in the instantiation of a native locator instead of the desired URL locator class.

**<physical-locator> Open Abstract Class**

**Superclasses** `<locator>`

A physical locator is a locator which refers to an object (such as a file or directory) in a physical file system. This locator class is useful for coercing an abstract locator into its corresponding physical counterpart.

**<file-system-locator> Open Abstract Class**

**Superclasses** `<physical-locator>`

A file system locator is a locator that refers to either a directory or a file within the file system.

**<file-system-file-locator> Class**

**Superclasses** `<file-system-locator>, <file-locator>`
This locator refers to a file within a file system.

```<file-system-directory-locator> Class```

**Superclasses** `<file-system-locator>, <directory-locator>`

This locator refers to a directory within a file system.

```<directory-locator> Open Abstract Class```

**Superclasses** `<physical-locator>`

**Slots**

- `locator-relative?` – #t if the locator is relative, #f if it is absolute.
- `locator-path` – the path to the directory.

A directory locator is a locator that refers to a directory as distinct from a file. This is important in file systems which can view a directory as either a file or a directory. This locator class is useful for coercing a file locator into a form where it can be manipulated as a directory (e.g. for constructing a locator to a file in a directory).

```<native-directory-locator> Constant```

This is bound to the native directory locator type for the host platform. On Windows, this is typically `<microsoft-directory-locator>` while on POSIX platforms, it is `<posix-directory-locator>`.

```<file-locator> Open Abstract Class```

**Superclasses** `<physical-locator>`

A file locator is a locator which refers to a file as distinct from a directory. This is important in file systems which can view a directory as either a file or a directory. This locator class is useful for coercing a directory locator into a form where it can be manipulated as a file.

```<native-file-locator> Constant```

This is bound to the native file locator type for the host platform. On Windows, this is typically `<microsoft-file-locator>` while on POSIX platforms, it is `<posix-file-locator>`.

```<locator-error> Class```

All errors raised by the locator system should be instances of this error.

**Superclasses** `<format-string-condition>, <error>`

```<server-locator> Open Abstract Class```

The abstract superclass of locators for servers.

**Superclasses** `<locator>`

See also

- `<server-url>`
- `<microsoft-server-locator>`

```list-locator Open Generic function```

Return a sequence of locators that are children of the given locator.

**Signature** `list-locator (locator) => (locators)`

**Parameters**

- `locator` – An instance of `<locator>`.

**Values**

- `locators` – An instance of `<sequence>`.
Discussion

Return a sequence of locators that are children of the given locator.

Note that this should only be called on a locator for which supports-list-locator? returns true.

See also

• supports-list-locator?

list-locator(<file-system-directory-locator>) Method

Returns a sequence of locators for the files and directories within the directory specified by the directory locator.

Parameters

• locator – An instance of <file-system-directory-locator>.

Values

• locators – An instance of <sequence>.

Discussion

Returns a sequence of locators for the files and directories within the directory specified by the directory locator.

Instances of <file-system-file-locator> for files and symbolic links. subdirectory-locator will be called to create locators for any directories.

See also

• supports-list-locator?(<file-system-directory-locator>)
• do-directory

locator-address Generic function

Signature locator-address (mailto) => (address)

Parameters

•mailto – An instance of <mail-to-locator>.

Values

•address – An instance of <string>.

Discussion Returns the email address specified by the mailto locator.

locator-as-string Open Generic function

Signature locator-as-string (class locator) => (string)

Parameters

•class – An instance of subclass(<string>).

•locator – An instance of <locator>.

Values

•string – An instance of <string>.

locator-base Open Generic function

Signature locator-base (locator) => (base)

Parameters
• **locator** – An instance of `<locator>`.

**Values**

• **base** – An instance of `false-or(<string>)`.

**Discussion** Returns the locator name without extension. For example, if a file locator’s path was `a/b/c.txt`, the locator-base would be `c`.

**locator-directory** **Open Generic function**

**Signature** `locator-directory (locator) => (directory)`

**Parameters**

• **locator** – An instance of `<locator>`.

**Values**

• **directory** – An instance of `false-or(<directory-locator>)`.

**Discussion** Returns the enclosing directory of a locator, or `#f` if it is not in a directory.

**locator-error** **Function**

**Signature** `locator-error (format-string #rest format-arguments) => (#rest results)`

**Parameters**

• **format-string** – An instance of `<string>`.

• **format-arguments** (#rest) – An instance of `<object>`.

**Values**

• **#rest results** – An instance of `<object>`.

**locator-extension** **Open Generic function**

**Signature** `locator-extension (locator) => (extension)`

**Parameters**

• **locator** – An instance of `<locator>`.

**Values**

• **extension** – An instance of `false-or(<string>)`.

**Discussion** Returns the extension part of the locator name. For example, if a file locator’s path was `a/b/c.txt`, the locator-extension would be `txt`. Returns `#f` if the locator does not have an extension.

**locator-file** **Generic function**

**Signature** `locator-file (url) => (file)`

**Parameters**

• **url** – An instance of `<file-index-url>` or `<cgi-url>`.

**Values**

• **file** – An instance of `<file-url>`.

**Discussion** Returns the file URL of a file index or CGI URL. For example, the locator-file of `http://example.com/index.html#tag` or `http://example.com/index.html?q=text` would be `http://example.com/index.html`
locator-host Open Generic function
Returns the host name.

Signature  locator-host (locator) => (host)

Parameters
• locator – An instance of <locator>.

Values
• host – An instance of false-or(<string>).

Discussion Returns the computer host name of a <server-url> or <microsoft-unc-locator>.

locator-name Generic function
Returns the name of this locator.

Signature  locator-name (locator) => (name)

Parameters
• locator – An instance of <locator>.

Values
• name – An instance of false-or(<string>).

Discussion This is typically the last component of the locator’s path but can be different for some specializations.

locator-name(<mailto-locator>) Method
Returns the email address of this locator.

Parameters
• locator – an instance of <mailto-locator>

Values
• name – An instance of <string>

locator-name(<microsoft-volume-locator>) Method
Returns the drive letter of this locator.

Parameters
• locator – an instance of <microsoft-volume-locator>

Values
• name – An instance of <string>

Discussion The drive is returned as a single letter, for example, ‘A’

locator-name(<microsoft-unc-locator>) Method
Returns the server name of this locator.
Parameters
  • locator – an instance of `<microsoft-unc-locator>`

Values
  • name – An instance of `<string>`

locator-path Open Generic function
Returns the directory path of a locator.

Signature  locator-path (locator) => (path)

Parameters
  • locator – An instance of `<directory-locator>`.

Values
  • path – An instance of `<sequence>`.

Discussion  Returns the directory path as a sequence of strings, each being the name of a path element.

Example

   locator-path(as(<file-locator>, "/a/b/c.d")) => #"a", "b"

locator-relative? Open Generic function

Signature  locator-relative? (locator) => (relative?)

Parameters
  • locator – An instance of `<locator>`.

Values
  • relative? – An instance of `<boolean>`.

locator-server Open Generic function

Signature  locator-server (locator) => (server)

Parameters
  • locator – An instance of `<locator>`.

Values
  • server – An instance of `false-or(<server-locator>)`.

locator-volume Open Generic function

Signature  locator-volume (locator) => (volume)

Parameters
  • locator – An instance of `<locator>`.

Values
  • volume – An instance of `false-or(<string>)`.

merge-locators Open Generic function

Signature  merge-locators (locator from-locator) => (merged-locator)

Parameters
• locator – An instance of `<physical-locator>`.
• from-locator – An instance of `<physical-locator>`.

Values
• merged-locator – An instance of `<physical-locator>`.

open-locator Open Generic function

Signature  open-locator (locator #key #all-keys) => (stream)

Parameters
• locator – An instance of `<locator>`.

Values
• stream – An instance of `<stream>`.

relative-locator Open Generic function

Returns a locator relative to another locator which references the same file as this locator.

Signature  relative-locator (locator from-locator) => (relative-locator)

Parameters
• locator – An instance of `<physical-locator>`.
• from-locator – An instance of `<physical-locator>`.

Values
• relative-locator – An instance of `<physical-locator>`.

Example  If self is `/a/b/c/d.txt` and root is `/a/b`

```dylan
let rel = relative-locator(self, root);
```

Then rel is `c/d.txt`

simplify-locator Open Generic function

Simplifies a locator by removing redundant elements from its path.

Signature  simplify-locator (locator) => (simplified-locator)

Parameters
• locator – An instance of `<physical-locator>`.

Values
• simplified-locator – An instance of `<physical-locator>`.

resolve-locator Open Generic function

Resolves all links, parent references (..), self references (.), and removes unnecessary path separators. Similar to `simplify-locator` except that it consults the file system to resolve links. A `<file-system-error>` is signaled if for any reason the path can’t be resolved. Examples include non-existent directory components, access denied, I/O error, etc. In short, this function follows the semantics of POSIX `realpath(3)`.

Signature  resolve-locator (locator) => (resolved-locator)

Parameters
• locator – An instance of `<physical-locator>`.

Values
• simplified-locator – An instance of `<physical-locator>`.
string-as-locator Open Generic function
Parse a string and create a locator.

**Signature**
string-as-locator (class string) => (locator)

**Parameters**
- class – An instance of subclass(<locator>).
- string – An instance of <string>.

**Values**
- locator – An instance of <locator>.

**Discussion**
This method should be specialized for each new locator class. It should return an instance of class, or raise a condition of type <locator-error>.

subdirectory-locator Open Generic function
Returns a directory locator for a subdirectory of a given directory.

**Signature**
subdirectory-locator (locator #rest sub-path) => (subdirectory)

**Parameters**
- locator – An instance of <directory-locator>.
- sub-path (#rest) – An instance of <object>.

**Values**
- subdirectory – An instance of <directory-locator>.

**Example**
```
let build-dir = subdirectory-locator(working-directory(), "_build");
```

supports-list-locator? Open Generic function
Returns whether or not a given locator supports the list-locator operation.

**Signature**
supports-list-locator? (locator) => (listable?)

**Parameters**
- locator – An instance of <locator>.

**Values**
- listable? – An instance of <boolean>.

**See also**
- list-locator

supports-list-locator? (file-system-directory-locator) Method
Returns true if the directory locator is not relative.

**Parameters**
- locator – An instance of <file-system-directory-locator>.

**Values**
- listable? – An instance of <boolean>.

**See also**
- list-locator(file-system-directory-locator)
supports-open-locator? Open Generic function
Returns whether or not a given locator supports the open-locator operation.

Signature supports-open-locator? (locator) => (openable?)

Parameters
  • locator – An instance of <locator>.

Values
  • openable? – An instance of <boolean>.

<web-locator> Abstract Class
Superclasses <locator>
The abstract superclass of locators that access a resource via web protocols, such as ftp or http.

<url> Abstract Sealed Class
Superclasses <web-locator>, <physical-locator>
The abstract superclass of web locators that reference a physical object. Use as(<url>, "...") to create an appropriate concrete subclass.

See also <file-url> <directory-url> <cgi-url> <file-index-url>

<directory-url> Class
Superclasses <url>, <directory-locator>
Represents directories that are accessible via web protocols.

<file-url> Class
Superclasses <url>, <file-locator>
Represents files that are accessible via web protocols.

<file-index-url> Class
Superclasses <url>
Represents a URL that has a fragment part, for example http://www.example.com/path/file.txt#fragment.

<cgi-url> Class
Superclasses <url>
Represents a URL that has a query part, for example http://www.example.com/path/file.txt?query=text.

locator-cgi-string Function
Return the query part of a <cgi-url>.

Signature locator-cgi-string(locator) => (string)

Parameters
  • locator – an instance of <cgi-url>

Values
  • string – an instance of <string>

locator-index Function
Return the fragment part of a :class:<file-index-url>
Signature  locator-index(locator) => (string)

Parameters

• locator – an instance of <file-index-url>

Values

• string – an instance of <string>

<mail-to-locator> Class

Superclasses <url>

Represents a locator which is an email address.

<server-url> Abstract Class

Represent a locator which is a machine accessible via web protocols.

Superclasses <url>, <server-locator>

Slots

• locator-host – The computer host
• locator-username – The user identifier
• locator-password – The user password

Operations locator-port, locator-default-port

The locator includes information on the protocol, host-name, port, user and password of the machine.

See also <http-server> <https-server> <ftp-server> <file-server>

<http-server> Sealed Class

A server for the http protocol.

Superclasses <server-url>

<https-server> Sealed Class

A server for the https protocol.

Superclasses <server-url>

<ftp-server> Sealed Class

A server for the ftp protocol.

Superclasses <server-url>

<file-server> Sealed Class

A locator using the file protocol.

Superclasses <server-url>

<microsoft-server-locator> Abstract Sealed Class

The abstract superclass of all servers using Microsoft protocols.

Superclasses <server-locator>

See also <microsoft-unc-locator> <microsoft-volume-locator>

<microsoft-unc-locator> Sealed Class

A server located using Microsoft’s Universal Naming Convention, for example \\ComputerName\Share

Superclasses <microsoft-server-locator>

<microsoft-volume-locator> Sealed Class

A server located using a volume name (drive letter) on a Microsoft system, for example C.
Superclasses <microsoft-server-locator>

<microsoft-file-system-locator> Abstract Class
The abstract superclass of files and directories on Microsoft file systems.

Superclasses <file-system-locator>

<microsoft-directory-locator> Class
A directory on a Microsoft file system.

Superclasses <microsoft-file-system-locator>,<directory-locator>

Slots
- locator-server – the server which holds this directory.

<microsoft-file-locator> Class
A file on a Microsoft file system.

Superclasses <microsoft-file-system-locator>,<file-locator>

Slots
- locator-directory – the directory that holds this file.
- locator-base – the file name without extension.
- locator-extension – the file extension.
- locator-name – the file name.

<posix-file-system-locator> Abstract Sealed Class
The abstract superclass of files and directories on a posix-like file system.

Superclasses <file-system-locator>

<posix-directory-locator> Sealed Class
A directory on a posix-like file system.

Superclasses <file-system-directory-locator>,<posix-file-system-locator>

<posix-file-locator> Sealed Class
A file on a posix-like file system.

Superclasses <file-system-file-locator>,<posix-file-system-locator>

Slots
- locator-directory – the directory that holds this file.
- locator-base – the file name without extension.
- locator-extension – the file extension.

locator-default-port Generic function
Return the default port associated with the locator’s protocol.

Signature locator-default-port(locator) => (port)

Parameters
- locator – An instance of <server-url>

Values
- port – An instance of false-or(<integer>)

Example
17.3 The file-system Module

The File-System module is part of the System library and provides a generic interface to the file system of the local machine. Remotely mounted file systems are also accessible using this module.

### Contents

- The file-system Module
  - Types specific to file system operations
  - Manipulating files
  - Manipulating directories
  - Finding out file system information
  - Finding out file information
  - The FILE-SYSTEM module

#### 17.3.1 Types specific to file system operations

The File-System module contains a number of types specifically designed for use by interfaces in the module.

- `<file-type>`
- `<pathname>`
- `<copy/rename-disposition>`

#### 17.3.2 Manipulating files

The File-System module contains a number of interfaces that let you perform standard file management operations on files already resident on the filesystem. You can rename, copy, or delete any file, and you can set any available properties for the file.

- `copy-file`
- `delete-file`
- `rename-file`
- `file-property-setter`

#### 17.3.3 Manipulating directories

The File-System module contains a number of interfaces that let you create and delete directories. These can be used in conjunction with the file manipulation operations described in Manipulating files to perform file management tasks at any position in the file system.
• create-directory
• delete-directory
• ensure-directories-exist
• do-directory
• working-directory-setter

17.3.4 Finding out file system information

A number of functions return environment information regarding the directory structure of the file system. Each function takes no arguments, and returns a pathname or list of pathnames. The return values can be use in conjunction with other functions to perform file-based operations relative to the directories involved.

• home-directory
• root-directories
• temp-directory
• working-directory

17.3.5 Finding out file information

Several interfaces in the File-System module allow you to interrogate files for information. You can find out whether a file exists, what its name is, or which directory it resides in, and you can find the current properties of the file.

• file-exists?
• file-properties
• file-property
• file-type

17.3.6 The FILE-SYSTEM module

This section contains a reference entry for each item included in the File-System module.

copy-file Function

Creates a copy of a file.

Signature  copy-file old-file new-file #key if-exists => ()

Parameters

• old-file – An instance of <pathname>.
• new-file – An instance of <pathname>.
• if-exists (#key) – An instance of <copy/rename-disposition>. Default value: #"signal".

Discussion  Copies old-file to new-file. If new-file already exists, the action of this function is controlled by the value of if-exists. The default is to prompt you before overwriting an existing file.

See also

• <copy/rename-disposition>
rename-file

<copy/rename-disposition> Type

The type that represents possible actions when overwriting existing files.

Equivalent one-of(#"signal", #"replace")

Discussion

This type represents the acceptable values for the if-exists: argument to the copy-file and rename-file functions. Only two values are acceptable:

- If #"signal" is used, then you are warned before a file is overwritten during a copy or move operation.
- If #"replace" is used, then you are not warned before a file is overwritten during a copy or move operation.

Operations

- copy-file
- rename-file

See also

- copy-file
- rename-file

create-directory Function

Creates a new directory in the specified parent directory.

Signature create-directory parent name => directory

Parameters

- parent – An instance of <pathname>.
- name – An instance of <string>.

Values

- directory – An instance of <pathname>.

Discussion Creates directory in the specified parent directory. The return value of this function can be used with concatenate to create pathnames of entities in the new directory.

See also

- delete-directory

delete-directory Function

Deletes the specified directory.

Signature delete-directory directory #key recursive? => ()

Parameters

- directory – An instance of <pathname>.
- recursive? (#key) – An instance of <boolean>. Default value: #f

Discussion Deletes the specified directory. By default the directory may only be deleted if it is empty. Pass recursive?: #t to delete the directory and its contents recursively.

See also

- create-directory
• delete-file

delete-file Function
Deletes the specified file system entity.

Signature  delete-file file => ()

Parameters

• file – An instance of <pathname>.

Discussion Deletes the file system entity specified by file. If file refers to a link, the link is removed, but the actual file that the link points to is not removed.

do-directory Function
Executes the supplied function once for each entity in the specified directory.

Signature  do-directory function directory => ()

Parameters

• function – An instance of <function>.

• directory – An instance of <pathname>.

Discussion
Executes function once for each entity in directory.

The signature of function is:

*function* *directory* *name* *type* => ()

where directory is an instance of <pathname>, name is an instance of <byte-string>, and type is an instance of <file-type>.

Within function, the values of directory and name can be concatenated to generate a <pathname> suitable for use by the other functions in the module.

The following calls are equivalent:

do-directory(my-function, "C:\USERS\JOHN\FOO.TEXT")
do-directory(my-function, "C:\USERS\JOHN\"")

as they both operate on the contents of C:\USERS\JOHN. The call:

do-directory(my-function, "C:\USERS\JOHN")

is not equivalent as it will operate on the contents of C:\\USERS.

ensure-directories-exist Function
Ensures that all the directories in the pathname leading to a file exist, creating any that do not, as needed.

Signature  ensure-directories-exist file => created?

Parameters

• file – An instance of <pathname>.

Values

• created? – An instance of <boolean>.
Discussion

Ensures that all the directories in the pathname leading to a file exist, creating any that do not, as needed. The return value indicates whether or not any directory was created.

The following calls are equivalent:

```dylan
ensure-directories-exist("C:\USERS\JOHN\FOO.TEXT")
ensure-directories-exist("C:\USERS\JOHN\")
```

as they will both create the directories USERS and JOHN if needed. The call:

```dylan
ensure-directories-exist("C:\USERS\JOHN")
```

is not equivalent as it will only create USERS if needed.

Example

```dylan
ensure-directories-exist("C:\USERS\JOHN\FOO.TEXT")
```

See also

- `create-directory`

file-exists? Function

Returns #t if the specified file exists.

**Signature**  
file-exists? file => exists?

**Parameters**

- `file` – An instance of `<pathname>`.

**Values**

- `exists?` – An instance of `<boolean>`.

**Discussion**  
Returns #t if file exists. If it refers to a link, the target of the link is checked.

file-properties Function

Returns all the properties of a file system entity.

**Signature**  
file-properties file => properties

**Parameters**

- `file` – An instance of `<pathname>`.

**Values**

- `properties` – An instance of a concrete subclass of `<explicit-key-collection>`.

**Discussion**  
Returns all the properties of file. The keys to the properties collection are the same as those use by `file-property`, above.

**Example**

```dylan
file-properties() ["#"size"]
```

See also

- `file-property`
- `file-property-setter`
file-property Sealed Generic function
Returns the specified property of a file system entity.

Signature  file-property file #key key => property

Parameters

• file – An instance of <pathname>.

• key (#key) – One of #"author", #"size", #"creation-date", #"access-date", #"modification-date", #"readable?", #"writeable?", #"executable?".

Values

• property – The value of the property specified by key. The type of the value returned depends on the value of key: see the description for details.

Discussion

Returns the property of file specified by key. The value returned depends on the value of key, as shown in Table Return value types of file-property.

<table>
<thead>
<tr>
<th>Value of key</th>
<th>Type of return value</th>
</tr>
</thead>
<tbody>
<tr>
<td>#&quot;author&quot;</td>
<td>false-or(&lt;string&gt;)</td>
</tr>
<tr>
<td>#&quot;size&quot;</td>
<td>&lt;integer&gt;</td>
</tr>
<tr>
<td>#&quot;creation-date&quot;</td>
<td>&lt;date&gt;</td>
</tr>
<tr>
<td>#&quot;access-date&quot;</td>
<td>&lt;date&gt;</td>
</tr>
<tr>
<td>#&quot;modification-date&quot;</td>
<td>&lt;date&gt;</td>
</tr>
<tr>
<td>#&quot;readable?&quot;</td>
<td>&lt;boolean&gt;</td>
</tr>
<tr>
<td>#&quot;writeable?&quot;</td>
<td>&lt;boolean&gt;</td>
</tr>
<tr>
<td>#&quot;executable?&quot;</td>
<td>&lt;boolean&gt;</td>
</tr>
</tbody>
</table>

Not all platforms implement all of the above keys. Some platforms may support additional keys. The #"author" key is supported on all platforms but may return #f if it is not meaningful on a given platform. Use of an unsupported key signals an error.

All keys listed above are implemented by Win32, though note that #"author" always returns #f.

See also

• file-property-setter
• file-properties

file-property-setter Sealed Generic function
Sets the specified property of a file system entity to a given value.

Signature  file-property-setter new-value file key => new-value

Parameters

• new-value – The type of this depends on the value of key. See the description for details.

• file – An instance of <pathname>.

• key – One of #"author", #"size", #"creation-date", #"access-date", #"modification-date", #"readable?", #"writeable?", #"executable?".
Values

- **new-value** – The type of this depends on the value of *key*. See the description for details.

Discussion

Sets the property of *file* specified by *key* to *new-value*. The type of *new-value* depends on the property specified by *key*, as shown in Table *New value types of file-property-setter* below.

<table>
<thead>
<tr>
<th>Value of <em>key</em></th>
<th>Type of <em>new-value</em></th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;author&quot;</td>
<td>false-or(&lt;string&gt;)</td>
</tr>
<tr>
<td>&quot;size&quot;</td>
<td>&lt;integer&gt;</td>
</tr>
<tr>
<td>&quot;creation-date&quot;</td>
<td>&lt;date&gt;</td>
</tr>
<tr>
<td>&quot;access-date&quot;</td>
<td>&lt;date&gt;</td>
</tr>
<tr>
<td>&quot;modification-date&quot;</td>
<td>&lt;date&gt;</td>
</tr>
<tr>
<td>&quot;readable?&quot;</td>
<td>&lt;boolean&gt;</td>
</tr>
<tr>
<td>&quot;writeable?&quot;</td>
<td>&lt;boolean&gt;</td>
</tr>
<tr>
<td>&quot;executable?&quot;</td>
<td>&lt;boolean&gt;</td>
</tr>
</tbody>
</table>

Note that *file-property-setter* returns the value that was set, and so return values have the same types as specified values, depending on the value of *key*.

Not all platforms implement all of the above keys. Some platforms may support additional keys. Use of an unsupported key signals an error.

The only property that can be set on Win32 is "writeable?'".

See also

- *file-property*
- *file-properties*

<file-system-error> Class

Error type signaled when any other functions in the File-System module signal an error.

**Superclasses** <error>, <simple-condition>

**Discussion** Signalled when one of the file system functions triggers an error, such as a permissions error when trying to delete or rename a file.

file-type Function

Returns the type of the specified file system entity.

**Signature** file-type *file* => file-type

**Parameters**

- **file** – An instance of <pathname>.

**Values**

- **file-type** – An instance of <file-type>.

**Discussion** Returns the type of *file*, the specified file system entity. A file system entity can either be a file, a directory, or a link to another file or directory.

<file-type> Type

The type representing all possible types of a file system entity.

**Equivalent** one-of("file", "directory", "link")
**Discussion** The type representing all possible types of a file system entity. An entity on the file system can either be a file, a directory or folder, or a link to another file or directory. The precise terminology used to refer to these different types of entity depends on the operating system you are working in.

**Operations**
- **do-directory**

**home-directory Function**
Returns the current value of the home directory.

**Signature** home-directory () => home-directory

**Values**
- **home-directory** – An instance of `<pathname>`.

**Discussion** Returns the current value of the home directory. The return value of this function can be used with concatenate to create pathnames of entities in the home directory.

**<pathname> Type**
The type representing a file system entity.

**Equivalent** type-union(<string>, <file-system-locator>)

**Discussion** A type that identifies a file system entity. This can be either a `<string>` or a `<file-system-locator>`.

**Operations**
- **copy-file**
- **create-directory**
- **delete-directory**
- **delete-file**
- **do-directory**
- **ensure-directories-exist**
- **file-exists?**
- **file-properties**
- **file-property**
- **file-property-setter**
- **file-type**
- **home-directory**
- **rename-file**

**rename-file Function**
Renames a specified file.

**Signature** rename-file old-file new-file #key if-exists => ()

**Parameters**
- **old-file** – An instance of `<pathname>`.
- **new-file** – An instance of `<pathname>`.
• **if-exists** – An instance of `<copy/rename-disposition>`. Default value: "signal".

**Discussion**
Renames `old-file` to `new-file`. If `new-file` already exists, the action of this function is controlled by the value of `if-exists`. The default is to prompt you before overwriting an existing file. This operation may fail if the source and destination are not on the same file system.

**See also**
- `copy-file`
- `<copy/rename-disposition>`

**root-directories Function**
Returns a sequence containing the pathnames of the root directories of the file systems on the local machine.

**Signature**
root-directories () => roots

**Values**
- roots – An instance of `<sequence>`.

**Discussion**
Returns a sequence containing the pathnames of the root directories of the file systems on the local machine.

**temp-directory Function**
Returns the pathname of the temporary directory in use.

**Signature**
temp-directory () => temp-directory

**Values**
- temp-directory – An instance of `<pathname>`, or false.

**Discussion**
Returns the pathname of the temporary directory in use. The return value of this function can be used with `concatenate` to create pathnames of entities in the temporary directory. If no temporary directory is defined, `temp-directory` returns #f. On Windows the temporary directory is specified by the TMP environment variable.

**working-directory Function**
Returns the working directory for the current process.

**Signature**
working-directory () => working-directory

**Values**
- working-directory – An instance of `<pathname>`.

**Discussion**
Returns the `<pathname>` of the current working directory in the current process on the local machine. You can use the return value of `working-directory` in conjunction with `concatenate` to specify pathnames of entities in the working directory.

**See also**
- `working-directory-setter`

**working-directory-setter Function**
Sets the working directory for the current process.

**Signature**
working-directory-setter directory => directory

**Parameters**
- directory – An instance of `<pathname>`.
Values

- **directory** – An instance of `<pathname>`.

Discussion

Sets the working directory for the current process.

Note that the following calls are equivalent:

```dylan
working-directory() := "C:\\USERS\\JOHN\\FOO.TEXT";
working-directory() := "C:\\USERS\\JOHN\";
```

as they will both set the working directory to `C:USERS\JOHN`. The call

```dylan
working-directory() := "C:\\USERS\\JOHN"
```

is not equivalent as it sets the working directory to `C:USERS`.

Example

```dylan
working-directory() := "C:\\USERS\\JOHN\";
```

See also

- **working-directory**

17.4 The operating-system Module

The operating-system module is part of the System library. It provides an interface to some features of the host machine’s operating system.

Contents

- The operating-system Module
  - Manipulating environment information
  - Running Applications
  - Manipulating application information
  - Working with shared libraries
  - The operating-system module

17.4.1 Manipulating environment information

The operating-system module contains a number of interfaces for examining and specifying information about the operating system environment of the host machine. As well as providing constants that you can use in your code, you can examine and set the value of any environment variable in the system.

The following constants contain machine-specific information:

- `$architecture-little-endian`
- `$machine-name`
These constants contain information about the hardware and software resident on the host machine. The constants let you programmatically check the current system conditions before executing a piece of code.

The following function also returns information about the machine:

- `machine-concurrent-thread-count`

The following two functions let you manipulate the values of any environment variables in the system.

- `environment-variable`
- `environment-variable-setter`

The following functions access information about the user logged on to the current machine, where available.

- `login-name`
- `login-group`
- `owner-name`
- `owner-organization`

### 17.4.2 Running Applications

- `run-application`

### 17.4.3 Manipulating application information

The operating-system module contains a number of functions for manipulating information specific to a given application, rather than the environment as a whole. You can run or quit any application, and interrogate the running application for application-specific information.

- `exit-application`
- `register-application-exit-function`
- `application-arguments`
- `application-name`
- `application-filename`
- `tokenize-command-string`
- `current-process-id`
- `parent-process-id`

### 17.4.4 Working with shared libraries

- `load-library`
17.4.5 The operating-system module

This section contains a reference entry for each item exported from the System library’s operating-system module.

application-arguments Function
Returns the arguments passed to the running application.

Signature  application-arguments => arguments

Values
  • arguments  – An instance of <simple-object-vector>.

Discussion  Returns the arguments passed to the running application as a vector of instances of <byte-string>.

See also
  • application-filename
  • application-name
  • tokenize-command-string

application-filename Function
Returns the full filename of the running application.

Signature  application-filename => false-or-filename

Values
  • false-or-filename  – An instance of false-or(<byte-string>).

Discussion  Returns the full filename (that is, the absolute pathname) of the running application, or #f if the filename cannot be determined.

Example  The following is an example of an absolute pathname naming an application:

"C:\\Program Files\\foo\\bar.exe"

See also
  • application-arguments
  • application-name
  • tokenize-command-string

application-name Function
Returns the name of the running application.

Signature  application-name => name

Values
  • name  – An instance of <byte-string>.

Discussion  Returns the name of the running application. This is normally the command name as typed on the command line and may be a non-absolute pathname.

Example  The following is an example of a non-absolute pathname used to refer to the application name:

"foo\\bar.exe"

See also
• application-arguments
• application-filename
• tokenize-command-string

$architecture-little-endian? Constant
Constant specifying whether the processor architecture is little-endian.

Type <boolean>

Discussion This constant is a boolean value that is true if the processor architecture is little-endian and false if it is big-endian. (A processor is little-endian if the rightmost bit in a word is the least-significant bit.) For processors implementing the Intel x86 architecture this value is #t.

See also
• $machine-name
• $os-name
• $os-variant
• $os-version
• $platform-name

current-process-id Function
Returns the integer value for the current process ID.

Signature current-process-id => pid

Values
• pid – An instance of <integer>.

Discussion Returns the integer value of the current process ID.

See also
• current-thread-id
• parent-process-id

environment-variable Function
Returns the value of a specified environment variable.

Signature environment-variable name => value

Parameters
• name – An instance of <byte-string>.

Values
• value – An instance of <byte-string>, or #f.

Discussion Returns the value of the environment variable specified by name, or #f if there is no such environment variable.

See also
• environment-variable-setter

evironment-variable-setter Function
Sets the value of an environment variable.

Signature environment-variable-setter new-value name => new-value
Parameters

- **new-value** – An instance of `<byte-string>`, or `#f`.
- **name** – An instance of `<byte-string>`.

Values

- **new-value** – An instance of `<byte-string>`, or `#f`.

Discussion

Changes the value of the environment variable specified by `name` to `new-value`. If `new-value` is `#f`, the environment variable is undefined. If the environment variable does not already exist, `environment-variable-setter` creates it.

**Note:** Windows 95 places restrictions on the number of environment variables allowed, based on the total length of the names and values of the existing environment variables. The function `environment-variable-setter` only creates a new environment variable if it is possible within these restrictions. See the relevant Windows 95 documentation for more details.

See also

- `environment-variable`

exit-application Function

Terminates execution of the running application.

**Signature** `exit-application status => ()`

**Parameters**

- **status** – An instance of `<integer>`.

**Discussion**

Terminates execution of the running application, returning the value of `status` to whatever launched the application, for example an MS-DOS window or Windows 95/NT shell.

**Note:** This function is also available from the `dylan-extensions` module in the `dylan` library and the `common-extensions` module of the `common-dylan` library.

See also

- `register-application-exit-function`

load-library Function

Loads a shared library into the current process.

**Signature** `load-library name => module`

**Parameters**

- **name** – An instance of `<string>`.

**Values**

- **module** – An instance of `<machine-word>`.

**Discussion**

Loads the library specified by `name` into the current process. The library must be a shared library.
If the library is a library written in Dylan, then when it loaded, constructor functions will run which set up the various methods and other Dylan objects within the shared library. Top level code within the library will be executed.

Load-time failures, for example due to missing files or unsatisfied symbol dependencies, will cause an `<error>` condition to be signaled.

**Note:** Dynamic loading of Dylan code is currently only supported on non-Windows platforms using the LLVM back-end, and on Windows using the HARP back-end.

---

**login-name Function**

Returns as an instance of `<string>` the name of the user logged on to the current machine, or `#f` if unavailable.

**Signature**

```
login-name () => name-or-false
```

**Values**

- **name-or-false** – An instance of `false-or(<string>)`.

**Discussion**

Returns as an instance of `<string>` the name of the user logged on to the current machine, or `#f` if unavailable.

**See also**

- `login-group`

---

**login-group Function**

**Signature**

```
login-group () => group-or-false
```

**Values**

- **group-or-false** – An instance of `false-or(<string>)`.

**Discussion**

Returns as an instance of `<string>` the group (for example NT domain, or Windows Workgroup) of which the user logged on to the current machine is a member, or `#f` if the group is unavailable.

**See also**

- `login-name`

---

**$machine-name Constant**

Constant specifying the type of hardware installed in the host machine.

**Type** `<symbol>`

**Value** `#"x86", #"x86-64", #"ppc"`

**Discussion**

This constant is a symbol that represents the type of hardware installed in the host machine.

**See also**

- `$architecture-little-endian?`
- `$os-name`
- `$os-variant`
- `$os-version`
- `$platform-name`
$os-name Constant  
Constant specifying the operating system running on the host machine.

**Type** <symbol>  
**Value** #"win32", #"linux", #"darwin", #"freebsd"

**Discussion** This constant is a symbol that represents the operating system running on the host machine.

**See also**  
• $architecture-little-endian?  
• $machine-name  
• $os-variant  
• $os-version  
• $platform-name

$os-variant Constant  
Constant specifying which variant of an operating system the current machine is running, where relevant.

**Type** <symbol>  

**Discussion** This constant is a symbol value distinguishing between variants of the operating system identified by $os-name, where relevant; otherwise it has the same value as $os-name. On Windows, the possible values are #"win3.1", #"win95", #"win98", and #"winnt".

**See also**  
• $architecture-little-endian?  
• $machine-name  
• $os-name  
• $os-version  
• $platform-name

$os-version Constant  
Constant specifying which version of an operating system the current machine is running.

**Type** <string>  

**Discussion** The constant $os-version is a string value that identifies the version of the operating system. For Windows NT, a typical value would be "4.01381 Service Pack 3". For Windows 95, a typical value would be "4.01212 B".

**See also**  
• $architecture-little-endian?  
• $machine-name  
• $os-name  
• $os-variant  
• $platform-name

owner-name Function  
Returns the name of the user who owns the current machine, if available.

**Signature** owner-name () => name-or-false
Values

- name-or-false – An instance of false-or(<string>).

Discussion Returns as an instance of <string> the name of the user who owns the current machine (that is, the name entered when the machine was registered), or #f if the name is unavailable.

owner-organization Function
Returns the organization to which the user who owns the current machine belongs, if available.

Signature owner-organization () => organization-or-false

Values

- organization-or-false – An instance of false-or(<string>).

Discussion Returns as an instance of <string> the organization to which the user who owns the current machine belongs, or #f if the name is unavailable.

parent-process-id Function
Returns the integer value for the parent process ID.

Signature parent-process-id => pid

Values

- pid – An instance of <integer>.

Discussion Returns the integer value of the parent process ID.

Note: This is not yet implemented on Windows.

See also

- current-process-id
- current-thread-id

$platform-name Constant
Constant specifying the operating system running on and the type of hardware installed in the host machine.

Type <symbol>

Value #"x86-win32", #"x86-linux", etc.

Discussion This constant is a symbol that represents both the operating system running on, and the type of hardware installed in, the host machine. It is a combination of the $os-name and $machine-name constants.

Example #"x86-win32", #"x86_64-linux"

See also

- $machine-name
- $os-name

machine-concurrent-thread-count Function
Return the number of concurrent execution threads available.

Signature machine-concurrent-thread-count => count

Values
• **count** – An instance of `<integer>`.

**Discussion** Returns the number of execution threads currently available. This normally corresponds to the number of logical processor cores currently online, and may vary over the lifetime of the program.

**register-application-exit-function Function**

Register a function to be executed when the application is about to exit.

**Signature** `register-application-exit-function function => ()`

**Parameters**

- **function** – An instance of `<function>`.

**Discussion**

Register a function to be executed when the application is about to exit. The Dylan runtime will make sure that these functions are executed.

The `function` should not expect any arguments, nor expect that any return values be used.

**Note:** Currently, the registered functions will be invoked in the reverse order in which they were added. This is not currently a contractual guarantee and may be subject to change.

**Note:** This function is also available from the `dylan-extensions` module in the `dylan` library and the `common-extensions` module of the `common-dylan` library.

**Example**

**See also**

- `exit-application`

**run-application Function**

Launches an application using the specified name and arguments.

**Signature** `run-application command #key minimize? activate? under-shell? inherit-console? => status`

**Parameters**

- **command** – An instance of `<string>`.
- **minimize? (#key)** – An instance of `<boolean>`.
- **activate? (#key)** – An instance of `<boolean>`.
- **under-shell? (#key)** – An instance of `<boolean>`.
- **inherit-console? (#key)** – An instance of `<boolean>`.

**Values**

- **status** – An instance of `<integer>`.

**Discussion**

Launches an application using the name and arguments specified in command. Using this function is equivalent to typing the command in a MS-DOS window. The return value is the exit status returned by the application.

If the `minimize?` keyword is `#t`, the command’s shell will appear minimized. It is `#f` by default.
If the `activate?` keyword is `#t`, the shell window becomes the active window. It is `#t` by default.

If the `under-shell?` keyword is `#t`, an MS-DOS shell is created to run the application; otherwise, the application is run directly. It is `#f` by default.

If the `inherit-console?` keyword is `#t`, the new application uses the same console window as the current application; otherwise, the new application is created with a separate console window. It is `#t` by default.

See also

- `exit-application`

### tokenize-command-string Function

Parses a command line into a command name and arguments.

**Signature**

\[ \text{tokenize-command-string\ line} \Rightarrow \text{command \#rest\ arguments} \]

**Parameters**

- `line` – An instance of `<byte-string>`.

**Values**

- `command` – An instance of `<byte-string>`.
- `#rest\ arguments` – Instances of `<byte-string>`.

**Discussion**

Parses the command specified in `line` into a command name and arguments. The rules used to tokenize the string are given in Microsoft’s C/C++ reference in the section “Parsing C Command-Line Arguments”.

See also

- `application-arguments`
- `application-name`
18.1 The t-lists Module

<t-list> Open Primary Abstract Class

Superclasses <deque>

Init-Keywords

• first-pair – An instance of <list>.
• last-pair – An instance of <list>.

Discussion

The t-lists library is an implementation of a tail concatenate list that supports efficient append operations.

The <t-list> maintains a reference to the last object in the list. This allows appends to happen in O(1) time.

In the Lisp world, this data structure is known as tconc.

Operations

• concatenate!
• empty?
• last
• pop
• pop-last
• push
• push-last
This document describes Open Dylan’s set of Win32 API interface libraries. These libraries provide a low-level Dylan interface to the Win32 API in Microsoft Windows.

Each Dylan library is a simple translation of Win32 API header files into a set of interface declarations from Open Dylan’s C-FFI library. So you can write Windows applications in Dylan by using the same functions and types as you would in C, albeit with slightly modified names so that they conform to Dylan naming conventions and requirements.

The Open Dylan Win32 API has been constructed from several Dylan libraries. Win32 functionality is divided among these libraries, matching the contents of Microsoft’s DLLs, allowing Dylan applications to avoid references to DLLs they do not need to use.

With the exception of changes necessitated by Dylan naming conventions and requirements, the names of C items have been preserved. Hence this document does not provide an exhaustive list of the items available in the libraries. Instead, it explains the name mapping scheme we used in the conversion, and provides a collection of tips for writing Dylan applications with the libraries. Finally, there is a list of items not supported in our versions of these libraries.

**Contents**

- The Win32 API Libraries
  - Supported Win32 libraries
  - Content and organization of the Win32 API libraries
    - Notes on the translations
  - Naming and mapping conventions
    - Simple naming conventions
    - Mapping the null value
    - Mapping C types onto Dylan classes
    - Creating methods from Windows alias functions
    - Mapping C structure fields onto Dylan slot names
    - Handling return of multiple values
  - Defining callback functions
  - Error messages
  - Handling Dylan conditions in a Win32 application
  - Dealing with the C function WinMain
  - Combining bit mask constants
19.1 Supported Win32 libraries

Each Dylan library representing a portion of the Win32 API has a single module of the same name as the library itself. For example, the library Win32-Common has a module also called Win32-Common. An exception to this rule is Win32-User, which also exports the module Win32-Default-Handler.

The libraries are:

Win32-Common

- Data types, constants (including error codes), and structure accessors that are shared by the other modules.
- Most of these come from the Win32 header files WINDEF.H, WINNT.H, and WINERROR.H. (There is no DLL file supplied as standard with Windows that corresponds with this library, because there are no C functions in the header files to which it forms an interface.)
- Win32-Kernel Non-GUI system services, as implemented in KERNEL32.DLL and declared in WINBASE.H (files, pipes, semaphores, atoms, time, and so on) and WINNLS.H (National Language Support).
- **Note:** This library does not provide thread support. Thread support is being handled at a higher level by Dylan’s own Threads library. See the Core Features manual for details.
• Win32-GDI Graphics Device Interface, drawing graphics and text, and printing. Corresponds to WINGDI.H and GDI32.DLL.

• Win32-User Other windowing functions. Corresponds to WINUSER.H and USER32.DLL. Also contains win32-last-handler which can handle conditions and display them to the application user a simply Win32 dialog. That function is exported from the module Win32-Default-Handler.

• Win32-Version Version management. Corresponds to WINVER.H and VERSION.DLL.

• Win32-Dialog Common dialog boxes, as implemented in COMDLG32.DLL and declared in COMMDLG.H, DLGS.H, and CDERR.H.

Win32-Controls

• “Common controls”, including list view, tree view, property sheets, and so on (COMMCTRL.H and COMCTL32.DLL).

Win32-Registry

• Registry (WINREG.H and ADVAPI32.DLL).

Win32-Rich-Edit

• “Rich edit” controls (RICHEDIT.H and RICHED32.DLL).

• Win32-DDE Dynamic Data Exchange (DDE.H and DDEML.H).

• Win32-Shell API for querying and extending the Windows Shell. Corresponds to SHELLAPI.H and SHELL32.DLL.

• Winsock2 Corresponds to WINSOCK2.H, QOS.H, and MSWSOCK.H.

19.2 Content and organization of the Win32 API libraries

The Open Dylan Win32 API libraries are modeled closely upon Microsoft’s Win32 C libraries. Most names available in the Dylan libraries are the same as those available in the C libraries, though of course to conform to Dylan naming conventions and restrictions, many of the C names have been translated.

Note: Look at the library.dylan file in each library to see what each library provides. (Look in comlib.dylan for Win32-Common.) The libraries generally include only features that apply to both Windows NT and Windows 95/98. If there is an additional area of Win32 you would like to see these libraries support, please inform the Open Dylan support team.

19.2.1 Notes on the translations

The Win32-Common module re-exports some names from the C-FFI module that its user may need to use directly, without needing to use (or know about) the C-FFI module itself. These names are: null-pointer, null-pointer?, pointer-address, pointer-value, pointer-value, pointer-cast, <C-string>, <C-unicode-string>, destroy, and with-stack-structure.

Names that are documented as being obsolete and/or included in Win32 only for compatibility with Win16, are generally not defined in the Dylan libraries. The names excluded are listed in Index of Win32 names excluded from the Dylan libraries.

The extended API macros, defined in the optional C header file WINDOWSX.H, are not supported.
19.3 Naming and mapping conventions

A Dylan application using the Win32 API will generally use the same API names as a C program would, with the following modifications for consistency with Dylan conventions.

19.3.1 Simple naming conventions

Type names are enclosed in angle brackets. For example, \texttt{HANDLE} becomes \texttt{<HANDLE>}. Names of constants are prefixed by \$$. For example, \texttt{OPAQUE} becomes \texttt{$OPAQUE}.

Underscores are replaced by hyphens. Thus, a constant called \texttt{NO\_ERROR} becomes \texttt{$NO\_ERROR} and a class called \texttt{LIST\_ENTRY} becomes \texttt{<LIST\_ENTRY>}.

Hyphens will \textit{not} be inserted between capitalized words (for example, \texttt{CreateWindow} does not become \texttt{Create-Window}) since that is a less obvious mapping that is more likely to cause confusion when switching between Dylan code and Windows documentation.

19.3.2 Mapping the null value

In place of \texttt{NULL}, there are several constants providing null values for frequently used types, such as \texttt{$NULL\_HANDLE}, \texttt{NULL\_RECT}, and \texttt{$NULL\_STRING}. Null values for other pointer types may be designated by the expression \texttt{null-pointer(<FOO>)}.

Use the function \texttt{null-pointer?} to test whether a value is null. Do not use the expression \texttt{if(ptr)...} as is often done in C, since a null pointer is not the same as \texttt{#f}. There are also functions \texttt{null-handle} and \texttt{null-handle?} for creating and testing handles, since conceptually they are not necessarily pointers.

19.3.3 Mapping C types onto Dylan classes

The multitude of integer data types in C code (\texttt{int}, \texttt{long}, \texttt{unsigned}, \texttt{ULONG}, \texttt{DWORD}, \texttt{LRESULT}, and so on) are all designated as \texttt{<integer>} (or some appropriate subrange thereof) in Dylan method argument types. However, a \texttt{<machine-word>} needs to be used to represent values that do not fit in the signed 30-bit representation of an integer.

Names such as \texttt{<DWORD>} should not be used in application code because they refer to the FFI designation of the C value representation, not to a Dylan data type.

The C types \texttt{BOOL} and \texttt{BOOLEAN} are both mapped to \texttt{<boolean>} in Dylan. Use \texttt{#t} and \texttt{#f} instead of \texttt{TRUE} and \texttt{FALSE}.

\textbf{Note:} Beware that some functions, such as \texttt{TranslateAccelerator}, though documented to return \texttt{TRUE} or \texttt{FALSE}, actually return \texttt{int} instead of \texttt{BOOL}; in such a case, you will have to compare the result to 0.

Similarly, watch out for cases where C code passes \texttt{TRUE} or \texttt{FALSE} as an integer argument. To handle one common case, the Dylan implementation of \texttt{MAKELPARAM} accepts either an \texttt{<integer>} or \texttt{<boolean>} as the first argument.

The C types \texttt{CHAR}, \texttt{WCHAR}, and \texttt{TCHAR} are all mapped to \texttt{<character>} in Dylan. However, \texttt{UCHAR} is mapped to \texttt{<integer>} since that is how it is actually used.

Most of the pointer types in the Windows API have several names; for example: \texttt{PRECT}, \texttt{NPRECT}, and \texttt{LPRECT}. In 16-bit code, these distinguished between “near” and “far” pointers, but in 32-bit code there is no difference. Rather than carry the duplicate names over into Dylan, it would be simpler to use only the basic \texttt{P}... prefix names. However,
the \( LP \) names seem to be used much more often, and hence may be more familiar, and the Microsoft documentation still tends to use the \( LP \) names in most places. So the Dylan interface defines both the \( <P> \) and \( <LP> \) names even though they have identical values. The \( NP \) names are not defined in Dylan since they are not as commonly used.

Values of type \texttt{char*} in C are represented as instances of class \texttt{<C-string>} in Dylan. This is a subclass of \texttt{<string>}, so all of the normal string operations can be used directly. C function parameters of type \texttt{char*} will also accept an instance of \texttt{<byte-string>}; a C pointer is created to point to the characters of the Dylan data, so the string does not need to be copied. (Dylan byte strings maintain a NUL character at the end in order to allow them to be used directly by C.)

in the current implementation, that involves automatically copying the string at run time, but the need for copying is intended to be removed later.

The \texttt{TEXT} function can also be used to coerce a string literal to a \texttt{<C-string>}. This usage is consistent with the Win32 \texttt{TEXT} macro, although the current purpose is different.

The Dylan declarations for C types will generally follow the strict alternative versions of the C declarations. This means, for example, that the various handle types such as \texttt{<hmenu>} and \texttt{<hwnd>} are disjoint subclasses of \texttt{<handle>}, instead of all being equivalent.

### 19.3.4 Creating methods from Windows alias functions

Consider a Windows function called \texttt{Foo} which is an alias for either \texttt{FooA} (an 8-bit character version) or \texttt{FooW} (a 16-bit character version). In Dylan, only the name \texttt{Foo} will be defined, but it will be a generic function with separate methods for arguments of types \texttt{<C-string>}, \texttt{<C-unicode-string>}, \texttt{<byte-string>} or \texttt{<unicode-string>}. (Only the 8-bit versions will be supported in the initial implementation, both because the compiler is not ready to handle Unicode and because it will not work on Windows 95.)

### 19.3.5 Mapping C structure fields onto Dylan slot names

Because slot names are not in a separate name space in Dylan, the names of C structure fields will have the suffix \texttt{-value} added to form the name of the Dylan accessor function. For example, the C statement:

\[
\text{pt->x = x;}
\]

becomes in Dylan:

\[
\text{pt.x-value := x;}
\]

There is not any attempt to append \( ? \) to the names of predicate functions since it is not obvious exactly which functions that should apply to. The Dylan convention of \( *...* \) for global variables is not relevant since there are no global variables involved.

### 19.3.6 Handling return of multiple values

In cases where the C library function takes a pointer argument as a place to store a pointer, integer, or boolean value, the corresponding Dylan function uses multiple return values to return such output parameters following the original function return value. For example, where C code does:

\[
\text{BOOL ok;}
\text{DWORD NumberRead;}
\text{ok = ReadConsoleInput(handle, buffer, length, \& NumberRead);}
\]
in Dylan it would be:

```
let ( ok :: <boolean>, NumberRead :: <integer> ) =
    ReadConsoleInput(handle, buffer, length);
```

Similarly, this function returns multiple values instead of a structure:

```
let ( x, y ) = GetLargestConsoleWindowSize(handle);
```

### 19.4 Defining callback functions

The Win32-common library provides a define callback macro to make it easy to define callback functions without the application programmer needing to use the FFI define c-callable-wrapper macro directly. It is used like this:

```
define callback WndProc :: <WNDPROC> = my-window-function;
```

This says that WndProc is being defined as a C function pointer of type <WNDPROC>, which when called from C causes the Dylan function my-window-function to be run. The Dylan function will be defined normally using define method or define function, and it is the responsibility of the programmer to ensure that its argument signature is consistent with what <WNDPROC> requires. For example:

```
define method my-window-function(    hWnd :: <HWND>, // window handle    message :: <integer>, // type of message    uParam, // additional information    lParam) // additional information => return :: <integer>;
...```

Note that the uParam and lParam arguments might receive values of either type <integer> or <machine-word>, so it may be best not to specialize them. Often these values are not used directly anyway, but are passed to other functions (such as LOWORD and HIWORD) which have methods for handling either representation.

The other types of function supported by define callback are dialog functions (<DLGPROC>) and dialog hooks (<LP...HOOKPROC>), both of which have the same argument types as a window function, but return a <boolean>. (The dialog hook functions are actually declared in COMMDLG.H as returning a UINT, but the value is always supposed to be TRUE or FALSE, so the Dylan callback interface has been implemented using BOOL instead.)

### 19.5 Error messages

The Win32-Kernel library provides the following utility functions.

**win32-error-message Function**

*Signature*  

```
win32-error-message error-code => message
```

*Discussion*  

The error-code is an instance of <integer> or <machine-word> (type unioned).

The error-code argument is either a Windows a Windows error code (such as returned by GetLastError) or an SCODE (also known as an HRESULT) value (such as returned by most OLE/COM functions).
The function returns a text message (in a string) corresponding to the error code, #f if the code is not recognized. The returned string might have more than one line but does not have a newline at the end.

Example

```
win32-error-message(5) => "Access is denied."
```

**report-win32-error Function**

**Signature**

```
report-win32-error name #key error
```

**Discussion**

Signals a Dylan error if the Win 32 error code specified is not NO_ERROR. If no code is specified, the value returned by the Win32 API GetLastError is used. The error that is signaled includes both the error code and the error message, as computed by the function `win32-error-message`.

**check-win32-result Function**

**Signature**

```
check-win32-result name result
```

**Discussion**

Many Windows functions return #f or NULL to mean failure. The function `check-win32-result` checks the result to see if it indicates failure, and if so it calls `report-win32-error`.

Example

```
check-win32-result("SetWindowText", SetWindowText(handle, label))
```

**ensure-no-win32-error Function**

**Signature**

```
ensure-no-win32-error name
```

**Discussion**

Ensures that the Win32 API GetLastError does not indicate that an error occurred. If an occurs, it is signaled using `report-win32-error`.

### 19.6 Handling Dylan conditions in a Win32 application

The Win32-User library exports from its Win32-Default-Handler module a handler utility function called `win32-last-handler`, defined on objects of class `<serious-condition>`.

**win32-last-handler Function**

**Discussion**

Displays a rudimentary Win32 dialog to allow the user to decide what to do with the Dylan condition that has been signalled.

It is a handler utility function that can be by bound dynamically around a computation via `let handler` or installed globally via `last-handler-definer`. It is automatically installed as the last handler simply by using the Win32-User library.

The function has the following call syntax:

```
win32-last-handler (serious-condition, next-handler)
```

The `serious-condition` argument is an object of class serious condition. The `next-handler` argument is a function. The `win32-last-handler` function returns no values.

The following form defines a dynamic handler around some body:
let handler <serious-condition> = win32-last-handler;

while the following form installs a globally visible last-handler:

```dylan
define last-handler <serious-condition> = win32-last-handler;
```

**seealso** last-handler-definer and default-last-handler, exported from the Functional Dylan-Extensions library and module, in the *Core Features* reference manual.

### 19.7 Dealing with the C function WinMain

In C, the programmer has to supply a *WinMain* function as the main program for a GUI application, but in Dylan there is no main program as such. The beginning of execution is indicated simply by a top-level function call expression; this needs to be at the bottom of the last file listed in the project file. The Win32-Kernel and Win32-User libraries export functions to obtain the values which would have been the arguments to *WinMain*:

- `application-instance-handle() => <HINSTANCE>`
- `application-command-line() => <string>`
  // arguments without program name
- `application-show-window() => <integer>` // one of $SW-...

There is no accessor provided for the *WinMain* previous instance parameter because on Win32, that parameter is always null, even for Win32s as well as NT and Windows 95.

The program can be terminated, with an exit code, by calling either the Win32 ExitProcess function or the exit-application function in Open Dylan’s System library. The latter method is preferred if the application might actually be run as part of another process.

The start of an application program might look something like this:

```dylan
define method my-main ()
  let hInstance :: <HINSTANCE> = application-instance-handle();
  let wc :: <PWNDCLASS> = make(<PWNDCLASS>);
  wc.style-value := 0;
  wc.lpfnWndProc-value := MainWndProc;
  ...
  RegisterClass(wc);
  let hWnd = CreateWindow( ... );
  ShowWindow(hWnd, application-show-window());
  UpdateWindow(hWnd);
  let msg :: <PMSG> = make(<PMSG>);
  while ( GetMessage(msg, $NULL-HWND, 0, 0) )
    TranslateMessage(msg);
    DispatchMessage(msg);
  end;
  ExitProcess(msg.wParam-value);
end method my-main;
```

my-main(); // this is what initiates execution.

For a complete example program, see

```
Examples\Win32\windows-ffi-example\example.dylan
```
in the Open Dylan installation directory.
19.8 Combining bit mask constants

Where C code would use the | operator to combine bit mask constants, Dylan code usually uses the logior function. However, a few such constants have values of type <machine-word> when they will not fit in a small integer, and logior only works on instances of <integer>. Because of this, the win32-common library exports a %logior function which is used like logior except that it accepts values of either type <integer> or <machine-word> and returns a <machine-word> result. It can be used in most places that accept a bit mask (C types DWORD, ULONG, LPARAM, and so on), but must be used if any of the arguments are a <machine-word>. The contexts where this is likely to occur are:

- Window style parameter of CreateWindow ($WS-\ldots)$
- Flags value for CreateFile or CreateNamedPipe ($FILE-FLAG-\ldots$)
- $SLOCALE-NOUSEROVERRIDE$ for LCTYPE parameters for GetLocaleInfoA, GetLocaleInfo, and possibly others, or $dwFlags$ parameter of GetTimeFormat, GetNumberFormat, GetCurrencyFormat, or GetDateFormat.
- Mask and effects values in CHARFORMAT structure for “rich edit” controls ($SCF\ldots$ and $S\ldots$)
- Mask value in PARAFORMAT structure for “rich edit” controls ($PF\ldots$)

19.9 Other minor details

The types <FARPROC> and <PROC> are defined as equivalent to <C-function-pointer>, so any C function wrapper object can be passed to a routine taking a <FARPROC> without needing to do any type conversion like that needed in C.

Type casts between handles and integers (<integer> or <machine-word>) can be done by using as. For example:

```dylan
window-class.hbrBackground-value :=
as(<HBRUSH>, $COLOR-WINDOW + 1);
```

Note that pointers and handles must be compared using =, not ==, in order to compare the C address instead of the Dylan wrapper objects.

For type casts from one pointer type to another, use the function pointer-cast instead of as. Think of as as converting the data structure pointed to, while pointer-cast operates on just the pointer itself.

The Dylan function pointer-value can be used to convert between a Dylan integer and a LARGE-INTEGER or ULARGE-INTEGER. For example:

```dylan
let li :: make(<PLARGE-INTEGER>); pointer-value(li) := 0;
```

allocates a LARGE-INTEGER and sets its value to 0, without needing to be concerned with the individual fields of the internal representation. Alternatively, you can use an initialization keyword:

```dylan
let li :: make(<PLARGE-INTEGER>, value: 0);
```

The C macros MAKEPOINT, MAKEPOINTS, and LONG2POINT do not easily translate to Dylan. Instead, use the Dylan function lparam-to-xy to split a parameter into two signed numbers. For example:

```dylan
let ( x, y ) = LPARAM-TO-XY(lParam);
```

In Dylan, <RECTL> is an alias of <RECT> instead of being a distinct type. (In Win32, they are structurally equivalent but were separate types for the sake of source code compatibility with Win16; there is no need to maintain that artificial distinction in Dylan.)
Windows resource files (.rc files) can be included by using the LID file field RC-Files:

19.10 Index of Win32 names excluded from the Dylan libraries

The names listed in the index below are excluded from the Open Dylan Win32 API libraries because they are obsolete. Functions for old-style metafiles (CreateMetaFile, CloseMetaFile, and so on) are described in the Win32 API as being obsolete, but they are being supported because they are needed for OLE applications to exchange data with 16-bit applications.

Functions `wsprintf` and `wvsprintf` are not supported because the Dylan function `format-to-string` serves the same purpose. Also, the FFI doesn’t currently support C functions with a variable number of arguments.

The extended API macros defined in optional C header file `windowsx.h` will not be supported by the Dylan interface.

The 64-bit utility macros `Int32x32To64`, `Int64ShllMod32`, `Int64ShraMod32`, and `UInt32x32To64` are not planned to be supported since there is no clear need for them and the functionality can be obtained by using Dylan extended integers. However, an interface to function `MulDiv` is provided, since it is an ordinary C function that is commonly used.

19.10.1 Characters

    _hread, _hwrite, _lclose, _lcreat, _llseek, _lopen, _read, _write

19.10.2 A


19.10.3 B

    BN_DBLCLK, BN_DISABLE, BN_DOUBLECLICKED, BN_HILITE, BN_PAINT, BN_PUSHED, BN_UNPUSHED, BS_USERBUTTON

19.10.4 C

    CPL_INQUIRE, ChangeSelector, CloseComm, CloseSound, CopyLZFile, CountVoiceNotes

19.10.5 D

    DOS3Call, DefHookProc, DefineHandleTable, DeviceMode, DlgDirSelect, DlgDirSelectComboBox

19.10.6 E

    EnumFonts, ERR_. . ., ExtDeviceMode
19.10.7 F

FixBrushOrgEx, FlushComm, FreeModule, FreeProcInstance, FreeSelector

19.10.8 G


19.10.9 H

HFILE, HFILE_ERROR

19.10.10 L

LZDone, LZStart, LimitEmsPages, LocalCompact, LocalInit, LocalNotify, LocalShrink, LockSegment

19.10.11 M

MAKEPOINT, MakeProcInstance, MoveTo

19.10.12 N

NetBIOSCall

19.10.13 O

OemToAnsi, OemToAnsiBuff, OffsetViewportOrg, OffsetWindowOrg, OpenComm, OpenFile, OpenSound

19.10.14 P

PM_NOYIELD, ProfClear, ProfFinish, ProfFlush, ProfInsChk, ProfSampRate, ProfSetup, ProfStart, ProfStop
19.10.15 R

READ, READ_WRITE, ReadComm, RegCreateKey, RegEnumKey, RegOpenKey, RegQueryValue, RegSetValue

19.10.16 S


19.10.17 U

UngetCommChar, UnhookWindowsHook, UnlockSegment

19.10.18 V

ValidateCodeSegments, ValidateFreeSpaces

19.10.19 W

WM_CTLCOLOR, WNetAddConnection, WNetCancelConnection, WRITE, WaitSoundState, WriteComm, WritePrivateProfileSection, WritePrivateProfileString, WritePrivateProfileStruct, WriteProfileSection, WriteProfileString

19.10.20 Y

Yield
INDICES AND TABLES

- genindex
- search
API INDEX

% (function), 155
%* (function), 155
%- (function), 155
* (method), 421
+ (function), 155
- (function), 155
* (method), 421
+ (method), 418
system:date:+(<date>) (method), 418
system:date:+(<duration>) (method), 418
- (method), 419
system:date:-(<date>) (method), 419
system:date:-(<duration>) (method), 420
/ (method), 421
/ (method), 421

< (method), 418
strings:strings:alphabetic? (generic function), 387
strings:strings:alphabetic?(<character>) (method), 388
strings:strings:alphabetic?(<string>) (method), 388
strings:strings:alphanumeric? (generic function), 388
strings:strings:alphanumeric?(<character>) (method), 389
strings:strings:alphanumeric?(<string>) (method), 389
application-arguments (function), 461
application-filename (function), 461
application-name (function), 461
common-dylan:machine-words:as (function), 151
io:streams:as(<integer>, <stream-position>) (method), 267
network:sockets:as(<string>, <ipv4-numeric-address>) (method), 267

%abs (function), 157
*all-connections* (constant), 349
*all-connections-lock* (constant), 349
<abstract-log> (class), 308
<abstract-socket> (class), 319
<address-in-use> (class), 326
<ambiguous-cursor-name> (class), 349
<assertion-constraint> (class), 349
network:sockets:accept (generic function), 321
network:sockets:accept (<tcp-server-socket>) (method), 322
acos (generic function), 171
acosh (generic function), 171
acquire-null-value (generic function), 366
add-option (function), 119
add-target (generic function), 310
adjust-stream-position (generic function), 266
aliases (generic function), 317
alignment-of (function), 50
all-addresses (generic function), 317
strings:strings:alphabetic? (generic function), 387
strings:strings:alphabetic?(<character>) (method), 388
strings:strings:alphabetic?(<string>) (method), 388
strings:strings:alphanumeric? (generic function), 388
strings:strings:alphanumeric?(<character>) (method), 389
strings:strings:alphanumeric?(<string>) (method), 389
application-arguments (function), 461
application-filename (function), 461
application-name (function), 461
common-dylan:machine-words:as (function), 151
io:streams:as(<integer>, <stream-position>) (method), 267
network:sockets:as(<string>, <ipv4-numeric-address>) (method), 267

A
$architecture-little-endian? (constant), 462
485
as-iso8601-string (function), 421
as-rfc1123-string (function), 423
as-rfc822-string (function), 422
as-unsigned (function), 153
asin (generic function), 171
asinh (generic function), 172
assert (macro), 128
associated-lock (function), 226
asynchronous (generic function), 366
atan (generic function), 172
atan2 (generic function), 172
atanh (generic function), 173
atomic-decrement! (macro), 232
atomic-increment! (macro), 232
automatic-finalization-enabled? (function), 185
automatic-finalization-enabled?-setter (function), 186

B
$bright-intensity (constant), 114
<bit-set> (class), 91
<bit-vector> (class), 95
<bit> (type), 95
<blocking-call-interrupted> (class), 326
<buffer> (class), 267
<buffered-socket> (class), 322
<buffered-stream> (class), 267
common-dylan:common-extensions:<byte-character> (class), 129
io:streams:<byte-character> (type), 268
<byte-string-stream> (class), 268
common-dylan:byte-vector:<byte-vector> (class), 125
io:streams:<byte-vector> (class), 269
common-dylan:byte-vector:<byte> (type), 125
io:streams:<byte> (type), 268
bit-count (function), 95
bit-vector-and (function), 96
bit-vector-and! (function), 96
bit-vector-andc2 (function), 96
bit-vector-andc2! (function), 97
bit-vector-not (function), 97
bit-vector-not! (function), 98
bit-vector-or (function), 98
bit-vector-or! (function), 98
bit-vector-xor (function), 99
bit-vector-xor! (function), 99
block-size (generic function), 236
common-dylan:byte-vector:byte-storage-address (generic function), 125
common-dylan:byte-vector:byte-storage-address (method), 125
io:streams:byte-storage-address (buffer) (method), 268
common-dylan:byte-vector:byte-storage-offset-address (generic function), 126
common-dylan:byte-vector:byte-storage-offset-address (method), 126
common-dylan:byte-vector:byte-storage-offset-address (method), 126
io:streams:byte-storage-offset-address (buffer) (method), 268
common-dylan:byte-vector:byte-vector-fill (generic function), 126
common-dylan:byte-vector:byte-vector-fill (byte-vector-character) (method), 126
common-dylan:byte-vector:byte-vector-fill (byte-vector-integer) (method), 126
byte-vector-ref (function), 126
byte-vector-ref-setter (function), 127

C
$color-black (constant), 115
$color-blue (constant), 115
$color-cyan (constant), 115
$color-default (constant), 115
$color-green (constant), 115
$color-magenta (constant), 115
$color-red (constant), 115
$color-white (constant), 115
$color-yellow (constant), 115
%ceiling (function), 156
%count-high-zeros (function), 154
%count-low-zeros (function), 154
%count-ones (function), 154
<C-boolean> (class), 86
<C-character> (class), 86
<C-double> (class), 52
<C-Dylan-object> (class), 88
<C-float> (class), 52
<C-function-pointer> (class), 74
<C-number> (class), 52
<C-pointer> (class), 53
<C-statically-typed-pointer> (class), 54
<C-string> (class), 86
<C-struct> (class), 64
<C-union> (class), 64
<C-value> (class), 50
<C-void*> (class), 54
<C-void> (class), 50
<cardinality-violation> (class), 349
<case-insensitive-string-table> (class), 350
common-dylan:byte-vector:copy-bytes(<sequence>,<integer>,<sequence>,<integer>,<integer>) (method), <dbms-not-specified> (class), 352
<dbms> (class), 352
common-dylan:byte-vector:copy-bytes(<simple-object-vector>,<integer>,<byte-vector>,<integer>,<integer>) (method), <diagnostic-table> (class), 352
<diagnostic> (class), 352
common-dylan:byte-vector:copy-bytes(<string>,<integer>,<string>,<integer>,<integer>) (method), <dbms-not-specified> (class), 352
common-dylan:byte-vector:copy-bytes(<string>,<integer>,<vector>,<integer>,<integer>) (method), <diagnostic-table> (class), 352
common-dylan:byte-vector:copy-bytes(<vector>,<integer>,<string>,<integer>,<integer>) (method), <diagnostic-table> (class), 352
common-dylan:byte-vector:copy-bytes(<vector>,<integer>,<vector>,<integer>,<integer>) (method), <diagnostic-table> (class), 352
copy-bytes! (function), 87
copy-file (function), 451
copy-into! (function), 87
cos (generic function), 173
cosh (generic function), 173
count-substrings (generic function), 396
create-directory (function), 452
current-date (function), 423
current-process-id (function), 462
current-thread (function), 216
current-thread-id (function), 216
cursor-name (generic function), 371

d$divide (function), 160
d$floor / (function), 159
d$tround / (function), 160
d$ttruncate / (function), 160
database (generic function), 371
database-object-name (generic function), 371
database-object-name-setter (generic function), 371
datatype-hints (generic function), 371
datatype-hints-setter (generic function), 371
date-day (function), 425
date-day-of-week (function), 425
date-hours (function), 426
date-microseconds (function), 426
date-minutes (function), 427
date-month (function), 427
date-seconds (function), 428
date-time-zone-offset (function), 428
date-time-zone-offset-setter (function), 429
date-year (function), 429
dbms (generic function), 372
dbms-name (generic function), 372
dbms-version (generic function), 372
debug-assert (macro), 130
debug-message (function), 130
strings:strings:decimal-digit? (generic function), 393
strings:strings:decimal-digit?(<character>) (method), 393
strings:strings:decimal-digit?(<string>) (method), 393
decode-date (function), 431
system:date:decode-duration (generic function), 431
system:date:decode-duration(<day/time-duration>) (method), 432
system:date:decode-duration(<year/month-duration>) (method), 432
fill-table! (function), 134
common-dylan:finalization:finalize (generic function), 187
common-dylan:finalization:finalize(<object>) (method), 188
finalize-when-unreachable (function), 187
find-diagnostic (function), 376
find-element (generic function), 135
find-substring (generic function), 396
float-to-string (function), 396
force-err (function), 245
force-out (function), 245
force-output (generic function), 272
io:format:format (generic function), 242
io:format:format(<byte-string>) (method), 242
format-date (function), 434
io:format-out:format-err (generic function), 245
io:format-out:format-err(<byte-string>) (method), 245
common-dylan:simple-format:format-out (function), 165
io:format-out:format-out (generic function), 244
io:format-out:format-out(<byte-string>) (method), 244
common-dylan:simple-format:format-to-string (function), 165
io:format:format-to-string (generic function), 242
io:format-to-string (generic function), 242
io:format-to-string(<byte-string>) (method), 242
found? (function), 143
from-hexstring(<byte-string>) (method), 128

get-log (generic function), 309
get-option-value (function), 120
get-property (generic function), 101
get-root-log (generic function), 309
strings:strings:graphic? (generic function), 390
strings:strings:graphic?(<character>) (method), 390
strings:strings:graphic?(<string>) (method), 390

<hash-state> (class), 105
<hash> (class), 235
<help-requested> (class), 119
<host-not-found> (class), 326
<host-unreachable> (class), 326
<http-server> (class), 448
<https-server> (class), 448
strings:strings:hexadecimal-digit?(generic function), 394
strings:strings:hexadecimal-digit?(<character>) (method), 394
strings:strings:hexadecimal-digit?(<string>) (method), 394
hash-algorithms:hash-algorithms:hexdigest(<byte-vector>) (method), 237
hash-algorithms:hash-algorithms:hexdigest(<hash>) (method), 237
hexstring(<byte-vector>) (method), 127
home-directory (function), 457
host-address (generic function), 317
host-name (generic function), 317
network:sockets:host-order (generic function), 318
network:sockets:host-order(<ipv4-numeric-address>) (method), 319
hypot (generic function), 174

$info-level (constant), 307
<implicit-zero-bit-padding> (class), 354
<incomplete-read-error> (class), 272
<indenting-stream> (class), 273
@index> (class), 354
<indicator-overflow> (class), 354
<indicator-policy> (constant), 354
<insufficient-item-descriptor-areas> (class), 354
<integrity-constraint-violation> (class), 354
<internal-socket-error> (class), 325
<internet-address> (class), 316
<interval-field-overflow> (class), 354
<invalid-address> (class), 326
<invalid-argument> (class), 354
<invalid-authorization-specification> (class), 354
<invalid-catalog-name> (class), 355
<invalid-character-set-name> (class), 355
<invalid-character-value-for-cast> (class), 355
<invalid-condition-number> (class), 355
<invalid-cursor-name> (class), 355
<invalid-datatype-hint> (class), 355
<invalid-datetime-format> (class), 355
<invalid-descriptor-count> (class), 355
<invalid-descriptor-index> (class), 355
<invalid-escape-character> (class), 356
<invalid-escape-sequence> (class), 356
locator-path (generic function) 444
locator-relative? (generic function) 444
locator-server (generic function) 444
locator-volume (generic function) 444
lock-stream (generic function) 275
locked (macro) 230
log (generic function) 175
log-additive? (generic function) 309
log-additive?-setter (generic function) 310
log-debug (function) 308
log-debug-if (function) 308
log-enabled? (generic function) 310
log-enabled?-setter (generic function) 310
log-error (function) 307
log-formatter (generic function) 311
log-formatter-setter (generic function) 311
log-message (generic function) 436
log-name (generic function) 436
log-targets (generic function) 309
log-to-target (generic function) 464
log-name (function) 175
strings:strings:lowercase (generic function) 397
strings:strings:lowercase (<character>) (method) 397
strings:strings:lowercase (<string>) (method) 397
strings:strings:lowercase! (generic function) 398
strings:strings:lowercase! (<character>) (method) 398
strings:strings:lowercase! (<string>) (method) 398
strings:strings:lowercase? (generic function) 399
strings:strings:lowercase? (<character>) (method) 399
strings:strings:lowercase? (<string>) (method) 399
$maximum-unsigned-machine-word (constant) 152
$minimum-signed-machine-word (constant) 152
$minimum-unsigned-machine-word (constant) 152
<machine-word> (class) 152
<mail-to-locator> (class) 448
<md5> (class) 237
<microsoft-directory-locator> (class) 449
<microsoft-file-locator> (class) 449
<microsoft-file-system-locator> (class) 449
<microsoft-server-locator> (class) 448
<microsoft-unc-locator> (class) 448
<microsoft-volume-locator> (class) 448
<multiple-server-transaction> (class) 357
<mutable-object-with-elements> (class) 26
<no-coercion> (constant) 349
<no-indicator> (constant) 349
<no-value> (constant) 349
%negative (function) 157
<network-not-responding> (class) 325
<no-data> (class) 357
<notification> (class) 225
<null-log-target> (class), 311
<null-value-eliminated-in-set-function> (class), 357
<null-value-no-indicator-parameter> (class), 357
<null-value> (class), 357
<num-addr> (class), 318
<num-addr-out-of-range> (class), 357
common-dylan:machine-words:negative? (function), 150
dylan:dylan:negative? (generic function), 11
dylan:dylan:negative? (<complex>) (method), 11
dylan:dylan:negative? (<machine-number>) (method), 11
network:sockets:network-order (generic function), 318
network:sockets:network-order (<ipv4-numeric-address>) (method), 319
new-line (generic function), 279
next-dbms-diagnostic (generic function), 379
null-pointer (function), 53
null-pointer? (function), 54
nullable? (generic function), 379
numeric-host-address (generic function), 317

O
$os-name (constant), 464
$os-variant (constant), 465
$os-version (constant), 465
<object-set> (class), 104
<object-with-elements> (class), 26
<option> (class), 121
<optional-parameter-option> (class), 122
<out-of-resources> (class), 326
object-hash (function), 5
strings:strings:octal-digit? (generic function), 395
strings:strings:octal-digit? (<character>) (method), 395
strings:strings:octal-digit? (<string>) (method), 395
common-dylan:machine-words:odd? (function), 149
dylan:dylan:odd? (generic function), 10
dylan:dylan:odd? (<complex>) (method), 10
dylan:dylan:odd? (<integer>) (method), 10
one-of (function), 138
open-locator (generic function), 445
open-target-stream (generic function), 312
option-parser-definer (macro), 123
option-present? (function), 120
outer-stream (generic function), 279
outer-stream-setter (generic function), 279
output-indicator (generic function), 380
output-indicator-setter (generic function), 380
dylan:threads:owned? (generic function), 221
dylan:threads:owned? (<read-write-lock>) (method), 225
dylan:threads:owned? (<recursive-lock>) (method), 222
dylan:threads:owned? (<simple-lock>) (method), 223
owner-name (function), 465
owner-organization (function), 466

P
$platform-name (constant), 466
*print-circle?* (variable), 248
*print-escape?* (variable), 248
*print-length* (variable), 249
*print-miser-width* (variable), 252
*print-pretty?* (variable), 249
<parameter-option> (class), 121
<pathname> (type), 457
<physical-locator> (class), 439
<position-type> (type), 280
<positionable-stream> (class), 280
<posix-directory-locator> (class), 449
<posix-file-locator> (class), 449
<posix-file-system-locator> (class), 449
<prepared-statement-not-a-cursor-specification> (class), 357
<privilege-not-granted> (class), 357
<privilege-not-revoked> (class), 358
<profiling-state> (type), 167
<profiling-timer> (class), 169
<progress-stream> (class), 329
pad (generic function), 408
pad-left (generic function), 408
pad-right (generic function), 409
parent-process-id (function), 466
parse-command-line (function), 119
parse-date-string (function), 437
parse-iso8601-string (function), 437
peek (generic function), 280
pointer-address (function), 53
pointer-cast (function), 53
pointer-value (generic function), 55
pointer-value-address (generic function), 56
pointer-value-setter (generic function), 56
position (generic function), 138
positional-options (generic function), 120
common-dylan:machine-words:positive? (function), 149
dylan:dylan:positive? (generic function), 11
dylan:dylan:positive?(<complex>)
  (method), 11
dylan:dylan:positive?(<machine-number>)
  (method), 11
possible-explanation (generic function), 380
pprint-indent (function), 250
pprint-logical-block (function), 250
pprint-newline (function), 251
pprint-tab (function), 251
primitive-address-add (primitive), 201
primitive-address-and (primitive), 201
primitive-address-equals? (primitive), 201
primitive-address-left-shift (primitive), 201
primitive-address-multiply (primitive), 201
primitive-address-not (primitive), 201
primitive-address-or (primitive), 201
primitive-address-right-shift (primitive), 201
primitive-address-subtract (primitive), 201
primitive-allocate (primitive), 200
primitive-allocate-thread-variable (primitive), 196
primitive-big-integer-at (primitive), 205
primitive-big-integer-at-setter (primitive), 205
primitive-bits-as-single-float (primitive), 203
primitive-byte-allocate (primitive), 200
primitive-byte-character-at (primitive), 205
primitive-byte-character-at-setter (primitive), 205
primitive-byte-element (primitive), 204
primitive-byte-element-setter (primitive), 204
primitive-continue-unwind (primitive), 200
primitive-copy-vector (primitive), 200
primitive-current-thread (primitive), 190
primitive-decoded-bits-as-single-float (primitive), 203
primitive-destroy-notification (primitive), 194
primitive-destroy-recursive-lock (primitive), 192
primitive-destroy-semaphore (primitive), 193
primitive-destroy-simple-lock (primitive), 191
primitive-destroy-thread (primitive), 189
primitive-double-float-at (primitive), 205
primitive-double-float-at-setter (primitive), 205
primitive-extended-float-at (primitive), 205
primitive-extended-float-at-setter (primitive), 205
primitive-false? (primitive), 200
primitive-fill! (primitive), 204
primitive-function-code (primitive), 200
primitive-function-environment (primitive), 200
primitive-ieee-double-float-at (primitive), 207
primitive-ieee-double-float-at-setter (primitive), 207
primitive-ieee-extended-float-at (primitive), 207
primitive-ieee-extended-float-at-setter (primitive), 207
primitive-ieee-single-float-at (primitive), 207
primitive-ieee-single-float-at-setter (primitive), 207
primitive-iep-apply (primitive), 200
primitive-initialize-current-thread (primitive), 189
primitive-inlined-nlx (primitive), 201
primitive-int? (primitive), 201
primitive-machine-integer-at (primitive), 205
primitive-machine-integer-at-setter (primitive), 205
primitive-make-box (primitive), 200
primitive-make-environment (primitive), 200
primitive-make-notification (primitive), 194
primitive-make-recursive-lock (primitive), 192
primitive-make-semaphore (primitive), 193
primitive-make-simple-lock (primitive), 190
primitive-make-string (primitive), 200
primitive-make-thread (primitive), 189
primitive-nlx (primitive), 201
primitive-owned-recursive-lock (primitive), 193
primitive-owned-simple-lock (primitive), 191
primitive-pointer-at (primitive), 205
primitive-pointer-at-setter (primitive), 205
primitive-read-cycle-counter (primitive), 188
primitive-read-return-address (primitive), 189
primitive-release-all-notification (primitive), 195
primitive-release-notification (primitive), 195
primitive-release-recursive-lock (primitive), 192
primitive-release-semaphore (primitive), 194
primitive-release-simple-lock (primitive), 191
primitive-replace! (primitive), 204
primitive-replace-bytes! (primitive), 204
primitive-signed-16-bit-integer-at (primitive), 206
primitive-signed-16-bit-integer-at-setter (primitive), 206
primitive-signed-32-bit-integer-at (primitive), 206
primitive-signed-32-bit-integer-at-setter (primitive), 206
primitive-signed-64-bit-integer-at (primitive), 206
primitive-signed-64-bit-integer-at-setter (primitive), 206
primitive-single-float-add (primitive), 203
primitive-single-float-as-bits (primitive), 203
primitive-single-float-at (primitive), 205
primitive-single-float-at-setter (primitive), 205
primitive-single-float-divide (primitive), 204
primitive-single-float-equals? (primitive), 203
primitive-single-float-greater-than-or-equal? (primitive), 203
primitive-single-float-greater-than? (primitive), 203
primitive-single-float-less-than-or-equal? (primitive), 203
primitive-single-float-less-than? (primitive), 203
primitive-single-float-multiply (primitive), 203
primitive-single-float-negate (primitive), 203
primitive-single-float-not-equals? (primitive), 203
primitive-single-float-or (primitive), 203
primitive-single-float-right-shift (primitive), 203
primitive-single-float-subtract (primitive), 203
primitive-single-float-xor (primitive), 203
primitive-thread-join-multiple (primitive), 190
primitive-thread-join-single (primitive), 190
primitive-thread-yield (primitive), 190
primitive-true? (primitive), 200
primitive-unsigned-16-bit-integer-at (primitive), 206
primitive-unsigned-16-bit-integer-at-setter (primitive), 206
primitive-unsigned-32-bit-integer-at (primitive), 206
primitive-unsigned-32-bit-integer-at-setter (primitive), 206
primitive-unsigned-64-bit-integer-at (primitive), 206
primitive-unsigned-64-bit-integer-at-setter (primitive), 206
primitive-unsigned-8-bit-integer-at (primitive), 206
primitive-unsigned-8-bit-integer-at-setter (primitive), 206
primitive-unsigned-machine-integer-at (primitive), 205
primitive-unsigned-machine-integer-at-setter (primitive), 205
primitive-untyped-at (primitive), 204
primitive-untyped-at-setter (primitive), 204
primitive-variable-lookup (primitive), 201
primitive-variable-lookup-setter (primitive), 201
primitive-wait-for-notification (primitive), 194
primitive-wait-for-notification-timed (primitive), 195
primitive-wait-for-recursive-lock (primitive), 192
primitive-wait-for-recursive-lock-timed (primitive), 192
primitive-wait-for-semaphore (primitive), 193
primitive-wait-for-semaphore-timed (primitive), 194
primitive-wait-for-simple-lock (primitive), 191
primitive-wait-for-simple-lock-timed (primitive), 191
primitive-xep-apply (primitive), 200
print (function), 247
io:format:print-message (generic function), 243
io:format:print-message (<condition>) (method), 243
io:format:print-message (<string> or <character>) (method), 244
io:format:print-message (<symbol>) (method), 243
print-object (generic function), 249
print-synopsis (generic function), 119
print-to-string (function), 249
strings:strings:printable? (generic function), 391
strings:strings:printable? (<character>) (method), 391
strings:strings:printable? (<string>) (method), 391
profiling (macro), 166
common-dylan:simple-profiling:profiling-state (generic function), 167
common-dylan:simple-profiling:profiling-state (<cpu-profiling-type>) (method), 167
common-dylan:simple-profiling:profiling-state singleton (#"allocation") (method), 167
common-dylan:simple-profiling:profiling-state singleton (#"allocation-stats") (method), 167
put-property! (macro), 101
Q
<query-expression-too-long-for-information-schema> (class), 358
R
$read-committed (constant), 349
$read-only (constant), 349
$read-uncommitted (constant), 349
$read-write (constant), 349
$repeatable-read (constant), 349
$reset-attributes (constant), 114
%round/ (function), 156
<random> (class), 168
<read-write-lock> (class), 224
<record> (class), 358
<recoverable-socket-condition> (class), 325
<recursive-lock> (class), 222
<referential-constraint> (class), 358
<regex-error> (class), 332
<regex-match> (class), 332
<regex> (class), 332
<remote-database-access> (class), 358
<repeated-parameter-option> (class), 122
<restricted-data-type-attribute-violation> (class), 358
<result-set-mutation-error> (class), 358
<result-set-policy> (class), 358
<result-set> (class), 359
<rolling-file-log-target> (class), 312
random (function), 168
range (function), 9
read (generic function), 281
read-element (generic function), 281
read-into! (generic function), 282
read-line (generic function), 283
read-line-into! (generic function), 283
read-through (generic function), 284
read-to (generic function), 285
read-to-end (generic function), 285
record-available? (generic function), 380
record-coercion-policy (generic function), 380
type-result-type (function), 54
regex-error-pattern (generic function), 332
type-result-type (profiling-state), 332
regex-group-count (generic function), 333
type-result-type (profiling-state), 333
regex-pattern (generic function), 333
type-result-type (profiling-state), 333
regex-position (generic function), 333
regex-replace (generic function), 333
regex-search (generic function), 334
regex-search-strings (generic function), 334
register-application-exit-function (function), 467
register-C-Dylan-object (function), 88
register-diagnostic-installer (function), 381
relative-selector (generic function), 445

release (generic function), 218

release (method), 221

release (method), 227

release (method), 244

release (method), 222

release (method), 220

release (method), 223

release-all (function), 227

remote-host (generic function), 323

remote-port (generic function), 323

remove-all-keys! (generic function), 108

remove-all-keys! (method), 109

remove-all-keys! (method), 109

remove-all-targets (generic function), 310

remove-keywords (generic function), 102

remove-property! (macro), 102

rename-target (generic function), 311

release-substrings (generic function), 396

report-win32-error (function), 477

resolve-locator (generic function), 445

row-count (generic function), 381

rowset-size (generic function), 381

run-application (function), 467

S

$scrollable-result-set-policy (constant), 349

$serializable (constant), 349

$single-e (constant), 177

$single-pi (constant), 177

$stderr-log-target (constant), 313

$stdout-log-target (constant), 313

%shift-left (function), 157

%shift-right (function), 158

*standard-error* (variable), 304

*standard-input* (variable), 303

*standard-output* (variable), 303

<schema-not-found> (class), 359

<scrollable-result-set> (class), 359

<search-condition-too-long-for-information-schema> (class), 359

<semaphore> (class), 219

<sequence-stream> (class), 286

<server-locator> (class), 440

<server-not-responding> (class), 326

<server-socket> (class), 320

<server-url> (class), 448

<set> (class), 104

<sha1> (class), 238

<sha224> (class), 238

<sha256> (class), 238

<sha384> (class), 238

<sha512> (class), 239

<simple-condition> (class), 139

<simple-lock> (class), 223

<socket-accessor-error> (class), 327

<socket-keyed-key-collection> (class), 325

<socket-condition> (class), 325

<socket> (class), 322

<sql-binary> (class), 360

<sql-bit-varying> (class), 360

<sql-bit> (class), 360

<sql-character-varying> (class), 360

<sql-character> (class), 360

<sql-client-unable-to-establish-connection> (class), 360

<sql-datatype> (class), 360

<sql-date> (class), 360

<sql-day-time-interval> (class), 360

<sql-decimal> (class), 360

<sql-double-precision> (class), 360

<sql-double> (class), 360

<sql-error> (class), 360

<sql-float> (class), 360

<sql-integer> (class), 360

<sql-longvarbinary> (class), 360

<sql-longvarchar> (class), 360

<sql-nationl-character-varying> (class), 361

<sql-national-character> (class), 361

<sql-numeric> (class), 361

<sql-real> (class), 361

<sql-server-rejected-establishment-of-connection> (class), 361

<sql-smallint> (class), 361

<sql-statement> (class), 361

<sql-table> (class), 362

<sql-time-with-time-zone> (class), 362

<sql-time> (class), 362
stream-open? (generic function), 291
stream-position (generic function), 291
stream-position-setter (generic function), 292
stream-size (generic function), 293
stream-supports-ansi-color? (generic function), 113
stream-supports-color? (generic function), 113
stream-supports-show-progress? (generic function), 329
string-as-locator (generic function), 445
string-compare (generic function), 113
string-greater? (generic function), 405
string-greater-ic? (generic function), 405
string-hash (function), 107
string-less? (generic function), 405
string-less-ic? (generic function), 407
string-to-integer (function), 140
strip (generic function), 410
strip-left (generic function), 410
strip-right (generic function), 410
subclass (function), 140
subclass-code (generic function), 382
subclass-origin (generic function), 383
subdirectory-locator (generic function), 446
supplied? (function), 145
system:locators:supports-list-locator? (generic function), 446
system:locators:supports-list-locator? (method), 446
supports-open-locator? (generic function), 446
synchronization-name (generic function), 218
synchronize-output (generic function), 294
synchronize-side-effects (function), 213

T
$trace-level (constant), 306
%truncate/ (function), 156
<table-not-found> (class), 469
<table-from-name> (generic function), 383
<table-name> (generic function), 383
<table-protocol> (generic function), 4
<transaction> (class), 364
<triggered-data-change-violation> (class), 364
<trim-error> (class), 364
table (macro), 109
table-name-from-name (generic function), 383
table-name (generic function), 383
table-protocol (generic function), 4
tan (generic function), 178
tanh (generic function), 178
target-pathname (generic function), 311
target-stream (generic function), 312
temp-directory (function), 458
text (generic function), 383
text-attributes (function), 114
text-setter (generic function), 383
thread (macro), 228
thread-id (generic function), 215
thread-name (function), 215
thread-yield (function), 216
timer-accumulated-time (generic function), 169
timer-running? (generic function), 170
timer-start (generic function), 169
timer-stop (generic function), 169
timing (macro), 165
tokenize-command-string (function), 468
transaction-mode (generic function), 384
transaction-mode-setter (generic function), 384
transaction-mode-setter (generic function), 384
type-for-file-stream (generic function), 294
type-system:directory-locator (generic function), 295

U
$unfound (constant), 143
$unsupplied (constant), 144
<udp-server-socket> (class), 324
<udp-socket> (class), 324
<unhandled-diagnostic> (class), 364
<unicode-character> (type), 295
<unicode-string-stream> (type), 295
<unique-constraint> (class), 364
<unknown-sqlstate> (class), 364
<unterminated-c-string> (class), 365
-url> (class), 447
<usage-error> (class), 119
$user> (class), 365
<using-clause-does-not-match-dynamic-parameter-specification> (class), 365
<using-clause-does-not-match-target-specification> (class), 365
<using-clause-required-for-dynamic-parameters> (class), 365
<using-clause-required-for-result-fields> (class), 365

API Index 499
u%* (function), 162
u%+ (function), 161
u%- (function), 161
u%< (function), 163
u%divide (function), 162
u%rotate-left (function), 162
u%rotate-right (function), 162
u%shift-left (function), 163
u%shift-right (function), 163
ud%divide (function), 163
ud%shift-left (function), 164
ud%shift-right (function), 164
unfound (function), 143
unfound? (function), 144
unique-index? (generic function), 384
unlock-stream (generic function), 296
unread-element (generic function), 296
unregister-C-Dylan-object (function), 89
unsupplied (function), 145
unsupplied? (function), 145
update-hash (generic function), 236
strings:strings:uppercase (generic function), 400
strings:strings:uppercase (<character>) (method), 400
strings:strings:uppercase (<string>) (method), 400
strings:strings:uppercase! (generic function), 401
strings:strings:uppercase! (<character>) (method), 401
strings:strings:uppercase! (<string>) (method), 401
strings:strings:uppercase? (generic function), 402
strings:strings:uppercase? (<character>) (method), 402
strings:strings:uppercase? (<string>) (method), 402
user (generic function), 384
V
value-sequence (generic function), 103
values-hash (function), 107
W
$warn-level (constant), 307
<warning-cursor-operation-conflict> (class), 365
<warning-string-data-right-truncation> (class), 366
<web-locator> (class), 447
<win32-socket-error> (class), 327
<with-check-option-violation> (class), 366
<wrapper-stream> (class), 299
dylan:threads:wait-for (generic function), 217
dylan:threads:wait-for (<notification>) (method), 226
dylan:threads:wait-for (<read-write-lock>) (method), 224
dylan:threads:wait-for (<recursive-lock>) (method), 222
dylan:threads:wait-for (<semaphore>) (method), 220
dylan:threads:wait-for (<simple-lock>) (method), 223
wait-for-io-completion (generic function), 297
when (macro), 146
strings:strings:whitespace? (generic function), 392
strings:strings:whitespace? (<character>) (method), 392
strings:strings:whitespace? (<string>) (method), 392
win32-error-message (function), 476
win32-last-handler (function), 477
with-c-string (macro), 87
with-connection (macro), 384
with-database (macro), 385
with-dbms (macro), 385
with-indentation (macro), 297
with-keywords-removed (macro), 103
with-lock (macro), 219
with-open-file (macro), 297
with-server-socket (macro), 321
with-socket-thread (macro), 316
with-stack-structure (macro), 85
with-stream-locked (macro), 298
with-transaction (macro), 385
working-directory (function), 458
working-directory-setter (function), 458
write (generic function), 300
write-element (generic function), 301
write-line (generic function), 302
write-message (generic function), 313
Y
<year/month-duration> (class), 438
Z
common-dylan:machine-words:zero? (function), 149
dylan:dylan:zero? (generic function), 11
dylan:dylan:zero? (<complex>) (method), 11
dylan:dylan:zero? (<machine-number>) (method), 11
\(^\)

\(^{(\text{generic function}), 170}\)
Symbols

* *(<duration>), 421
*all-connections*, 349
*all-connections-lock*, 349
*default-line-length*, 250
*print-circle?*, 248
*print-escape?*, 248
*print-length*, 249
*print-level*, 249
*print-miser-width*, 252
*print-pretty?*, 249
*standard-error*, 304
*standard-input*, 303
*standard-output*, 303
+/(<duration>), 421
+ method, 45
-<date>, 419
-<duration>, 420
= method, 57
$architecture-little-endian?, 462
$bright-intensity, 114
$color-black, 115
$color-blue, 115
$color-cyan, 115
$color-default, 115
$color-green, 115
$color-magenta, 115
$color-red, 115
$color-white, 115
$color-yellow, 115
$debug-level, 307
$default-coercion, 349
$default-log-formatter, 313
$default-result-set-policy, 349
$diagnostic-table, 349
$dim-intensity, 114
$double-e, 174
$double-pi, 174
$error-level, 307
$info-level, 307
$machine-name, 464
$machine-word-size, 152
$maximum-signed-machine-word, 152
$maximum-unsigned-machine-word, 152
$minimum-signed-machine-word, 152
$minimum-unsigned-machine-word, 152
$no-coercion, 349
$no-indicator, 349
$normal-intensity, 115
$null-log-target, 313
$null-value, 349
$sos-name, 464
$sos-variant, 465
$sos-version, 465
$platform-name, 466
$read-committed, 349
$read-only, 349
$read-uncommitted, 349
$read-write, 349
$repeatable-read, 349
$reset-attributes, 114
$scrollable-result-set-policy, 349
$serializable, 349
$single-e, 177
$single-pi, 177
$stderr-log-target, 313
$stdout-log-target, 313
$trace-level, 306
$unfound, 143
$unsupplied, 144
$warn-level, 307
%*, 155
<cardinality-violation>, 349
<case-insensitive-string-table>, 105
<catalog-not-found>, 350
<catalog>, 350
<cgi-url>, 447
<character-not-in-repertoire>, 350
<check-constraint>, 350
<choice-option>, 122
<coercion-policy>, 350
<coercion-record>, 350
<coloring-stream>, 112
<column>, 350
<command-line-parser-error>, 119
<command-line-parser>, 118
<connection-closed>, 326
<connection-does-not-exist>, 350
<connection-exception>, 350
<connection-failed>, 326
<connection-failure>, 351
<connection-name-in-use>, 351
<connection-not-specified>, 351
<connection>, 351
<constraint>, 351
<copy/rename-disposition>, 452
<cpu-profiling-type>, 167
<cursor-operation-conflict>, 351
<cursor-specification-cannot-be-executed>, 351
<data-exception>, 351
<data-not-available>, 351
<database-collection>, 351
<database-error>, 352
<database-object-not-found>, 352
<database-object>, 352
<database-statement>, 352
<database>, 352
<date>, 423
<datetime-field-overflow>, 352
<day-of-week>, 430
<day/time-duration>, 430
<dbms-not-specified>, 352
<dbms>, 352
<dependent-privilege-descriptors-still-exist>, 352
<diagnostic-table>, 352
<diagnostics>, 352
<directory-locator>, 440
<directory-url>, 447
<disconnect-error>, 353
<division-by-zero>, 353
<duration>, 432
<dynamic-sql-error>, 353
<empty-result-set>, 353
<end-of-stream-error>, 270
<error-in-assignment>, 353
<exclusive-lock>, 221
<feature-not-supported>, 353
<file-does-not-exist-error>, 271
<file-error>, 271
<file-exists-error>, 271
<file-index-url>, 447
<file-locator>, 440
<file-log-target>, 311
<file-server>, 448
<file-stream>, 272
<file-system-directory-locator>, 440
<file-system-error>, 456
<file-system-file-locator>, 439
<file-system-locator>, 439
<file-type>, 456
<file-url>, 447
<flag-option>, 121
<format-string-condition>, 135
<forward-only-result-set>, 353
<ftp-server>, 448
<hash-state>, 105
<hash>, 235
<help-requested>, 119
<host-not-found>, 326
<host-unreachable>, 326
<http-server>, 448
<https-server>, 448
<implicit-zero-bit-padding>, 354
<incomplete-read-error>, 272
<indenting-stream>, 273
<index>, 354
<indicator-overflow>, 354
<indicator-policy>, 354
<insufficient-item-descriptor-areas>, 354
<integrity-constraint-violation>, 354
<internal-socket-error>, 325
<internet-address>, 316
<interval-field-overflow>, 354
<invalid-address>, 326
<invalid-argument>, 354
<invalid-authorization-specification>, 354
<invalid-catalog-name>, 355
<invalid-character-set-name>, 355
<invalid-character-value-for-cast>, 355
<invalid-condition-number>, 355
<invalid-cursor-name>, 355
<invalid-datatype-hint>, 355
<invalid-datetime-format>, 355
<invalid-descrptor-count>, 355
<invalid-descrptor-index>, 355
<invalid-fetch-sequence>, 356
<invalid-escape-character>, 356
<invalid-escape-sequence>, 356
<invalid-match-group>, 332
<invalid-match-group>, 332
<invalid-parameter-value>, 356
<invalid-regex>, 332
<invalid-schema-name>, 356
<invalid-sql-descriptor-name>, 356
<invalid-sql-statement-name>, 356
<invalid-time-zone-displacement-value>, 356
<invalid-transaction-state>, 356
<invalid-transaction-termination>, 357
<ipv4-host-order-address>, 319
<ipv4-network-order-address>, 319
<ipv4-numeric-address>, 318
<isolation-level>, 357
<keyed-option>, 123
<locator-error>, 440
<locator>, 439
<log>, 218
<log-formatter>, 313
<log-level>, 306
<log-target>, 311
<numeric-address>, 318
<numeric-value-out-of-range>, 357
<option>, 121
<optional-parameter-option>, 122
<out-of-resources>, 326
<parameter-option>, 121
<pathname>, 457
<physical-locator>, 439
<position-type>, 280
<positionable-stream>, 280
<posix-directory-locator>, 449
<posix-file-locator>, 449
<posix-file-system-locator>, 449
<prepared-statement-not-a-cursor-specification>, 357
<privilege-not-granted>, 357
<privilege-not-revoked>, 358
<profiling-state>, 167
<profiling-timer>, 169
<progress-stream>, 329
<query-expression-too-long-for-information-schema>, 358
<random>, 168
<read-write-lock>, 224
<record>, 358
<recoverable-socket-condition>, 325
<recursive-lock>, 222
<referential-constraint>, 358
<regex-error>, 332
<regex-match>, 332
<regex>, 332
<remote-database-access>, 358
<repeated-parameter-option>, 122
<restricted-data-type-attribute-violation>, 358
<result-set-mutation-error>, 358
<result-set-policy>, 358
<result-set>, 359
<rolling-file-log-target>, 312
<schema-not-found>, 359
<scm>, 359
<search-condition-too-long-for-information-schema>, 359
<semaphore>, 219
<sequence-stream>, 286
<server-locator>, 440
<server-not-responding>, 326
<server-socket>, 320
<server-url>, 448
<set>, 104
<sha1>, 238
<sha224>, 238
<sha256>, 238
<sha384>, 238
<sha512>, 239
<simple-condition>, 139
<simple-lock>, 223
<socket-accessor-error>, 327
<socket-closed>, 326
<socket-condition>, 325
<socket-error>, 325
<socket>, 322
<sql-bigint>, 359
<sql-binary>, 360
<sql-bit-varying>, 360
<sql-bit>, 360  
<sql-character-varying>, 360  
<sql-character>, 360  
<sql-client-unable-to-establish-connection>, 360  
<sql-datatype>, 360  
<sql-date>, 360  
<sql-day-time-interval>, 360  
<sql-decimal>, 360  
<sql-double-precision>, 360  
<sql-double>, 360  
<sql-error>, 360  
<sql-float>, 360  
<sql-integer>, 360  
<sql-longvarbinary>, 360  
<sql-longvarchar>, 360  
<sql-national-character-varying>, 361  
<sql-national-character>, 361  
<sql-numeric>, 361  
<sql-real>, 361  
<sql-server-rejected-establishment-of-connection>, 361  
<sql-smallint>, 361  
<sql-statement>, 361  
<sql-table>, 361  
<sql-time-with-time-zone>, 362  
<sql-time>, 362  
<sql-timestamp-with-time-zone>, 362  
<sql-timestamp>, 362  
<sql-tinyint>, 362  
<sql-type-timestamp>, 362  
<sql-unknown-type>, 362  
<sql-unsupported-type>, 362  
<sql-varbinary>, 362  
<sql-warning>, 362  
<sql-year-month-interval>, 362  
<statement-completion-unknown>, 362  
<stream-log-target>, 312  
<stream-position>, 292  
<stream>, 287  
<stretchy-sequence>, 139  
<string-data-length-mismatch>, 363  
<string-data-right-truncation>, 363  
<string-stream>, 293  
<string-table>, 105, 139  
<string-error>, 363  
<successful-completion>, 363  
<synchronization>, 217  
<syntax-error-or-access-rule-violation-in-direct-sql-statement>, 363  
<syntax-error-or-access-rule-violation-in-dynamic-sql-statement>, 363  
<syntax-error-or-access-rule-violation>, 363  
<t-list>, 469  
<table-not-found>, 363  
<tcp-server-socket>, 321  
<tcp-socket>, 323  
<text-attributes>, 113  
<thread>, 214  
<transaction-mode>, 363  
<transaction-resolution-unknown>, 363  
<transaction-rollback-due-to-integrity-constraint-violation>, 364  
<transaction-rollback-due-to-serialization-failure>, 364  
<transaction-rollback>, 364  
<transaction>, 364  
<triggered-data-change-violation>, 364  
<trim-error>, 364  
<udp-server-socket>, 324  
<udp-socket>, 324  
<unhandled-diagnostic>, 364  
<unicode-character>, 295  
<unicode-string-stream>, 295  
<unique-constraint>, 364  
<unknown-sqlstate>, 364  
<unterminated-c-string>, 365  
<url>, 447  
<usage-error>, 119  
<user>, 365  
<using-clause-does-not-match-dynamic-parameter-specification>, 365  
<using-clause-does-not-match-target-specification>, 365  
<using-clause-required-for-dynamic-parameters>, 365  
<using-clause-required-for-result-fields>, 365  
<warning-cursor-operation-conflict>, 365  
<warning-string-data-right-truncation>, 366  
<web-locator>, 447  
<win32-socket-error>, 327  
<with-check-option-violation>, 366  
<wrapper-stream>, 299  
<year/month-duration>, 438  
A  
accept, 321  
    accept(<tcp-server-socket>), 322  
acos, 171  
acosh, 171  
acquire-null-value, 366  
add-option, 119  
add-target, 310  
adjust-stream-position, 266  
alias functions  
    creating methods from, 475  
aliases, 317  
alignment-of, 50  
alignment-of function, 50  
all-addresses, 317  
allocation  
    C storage, 83  
alphabetic?, 387
Index

 alphabetic?(<character>), 388
 alphabetic?(<string>), 388
 alphanumeric?, 388
 alphanumeric?(<character>), 389
 alphanumeric?(<string>), 389
 API libraries, 469
 application-arguments, 461
 application-filename, 461
 application-name, 461
 as, 151
 as(<integer>, <stream-position>), 267
 as(<string>, <ipv4-numeric-address>), 319
 as-iso8601-string, 421
 as-rfc1123-string, 423
 as-rfc822-string, 422
 asUnsigned, 153
 asin, 171
 asinh, 172
 assert, 128
 associated-lock, 226
 asynchronous, 366
 atan, 172
 atan2, 172
 atanh, 173
 atomic-decrement!, 232
 atomic-increment!, 232
 automatic-finalization-enabled?, 185
 automatic-finalization-enabled?-setter, 186

 basic options in C-FFI macros, 48
 bit mask constants
 combining, 478
 bit-count, 95
 bit-vector-and, 96
 bit-vector-and!, 96
 bit-vector-andc2, 96
 bit-vector-andc2!, 97
 bit-vector-not, 97
 bit-vector-not!, 98
 bit-vector-or, 98
 bit-vector-or!, 98
 bit-vector-xor, 99
 bit-vector-xor!, 99
 block-size, 236
 byte-storage-address, 125
 byte-storage-address(<buffer>), 268
 byte-storage-address(<byte-string>), 125
 byte-storage-address(<byte-vector>), 125
 byte-storage-offset-address, 126
 byte-storage-address(<buffer>), 268
 byte-storage-offset-address(<byte-string>), 126
 byte-storage-offset-address(<byte-vector>), 126
 byte-vector-fill, 126
 byte-vector-fill(<byte-vector>, <byte-character>), 126
 byte-vector-fill(<byte-vector>, <integer>), 126
 byte-vector-ref, 126
 byte-vector-ref-setter, 127

 C
 C describing Dylan functions for use by, 78
 C functions
 dealing with WinMain, 74
 describing to Dylan, 74
 in Dylan, 44
 C structure fields
 mapping onto Dylan slot names, 475
 C type macros
 notes on, 64
 C types
 in Dylan, 42
 mapping onto Dylan classes, 474
 C variables in Dylan, 46
 C-char-at, 59
 C-char-at function, 58, 59
 C-char-at-setter function, 58
 C-double-at function, 62
 C-double-at-setter function, 62
 C-FFI library, 40
 introduction, 41
 terminology, 48
 variables, 82
 C-FFI macros, 48
 C-float-at function, 62
 C-float-at-setter function, 63
 C-int-at function, 62
 C-int-at-setter function, 62
 C-long-at function, 61
 C-long-at-setter function, 61
 C-pointer-at function, 63
 C-pointer-at-setter function, 63
 C-short-at function, 59
 C-short-at-setter function, 60
 C-signed-char-at-setter function, 59
 C-signed-int-at function, 61
 C-signed-int-at-setter function, 62
 C-signed-int-at-setter function, 62
 C-signed-long-at function, 60
 C-signed-long-at-setter function, 61
 C-signed-short-at function, 59
 C-signed-short-at-setter function, 60
 C-type-cast, 55
 C-unsigned-char-at-setter function, 59
 C-unsigned-int-at function, 61
 C-unsigned-int-at-setter function, 61
 C-unsigned-long-at function, 60
 C-unsigned-long-at-setter function, 60

 508 Index
C-unsigned-short-at, 59
C-unsigned-short-at function, 59
C-unsigned-short-at-setter function, 59
C-variable definition macro, 82
callback functions
  defining, 476
case-insensitive-equal, 108
case-insensitive-string-hash, 107
catalog-from-name, 366
catalog-name, 366
catalogs, 367
catalogs-assist, 367
char-compare, 403
char-compare-ic, 403
char-equal-ic?, 403
characters, 480
check-win32-result, 477
check-win32-result function, 476
class-code, 367
class-origin, 367
classes
  <C-Dylan-object>, 88
  <C-boolean>, 86
  <C-char>, 51
  <C-character>, 86
  <C-double*>, 43
  <C-double>, 43, 52
  <C-example-string>, 69
  <C-float>, 52
  <C-function-pointer>, 74, 76
  <C-int*>, 52
  <C-int>, 42, 51, 52
  <C-number>, 52
  <C-pointer>, 49, 53–55, 58, 71, 74, 82
  <C-raw-char>, 51
  <C-raw-int>, 51
  <C-raw-signed-char>, 51
  <C-raw-signed-int>, 51
  <C-raw-signed-long>, 51
  <C-raw-signed-short>, 51
  <C-raw-unsigned-char>, 51
  <C-raw-unsigned-int>, 51
  <C-raw-unsigned-long>, 51
  <C-raw-unsigned-short>, 51
  <C-signed-char>, 51
  <C-signed-int>, 51
  <C-signed-long>, 51
  <C-signed-short>, 51
  <C-statically-typed-pointer>, 49, 54, 85
  <C-string>, 49, 86
  <C-struct>, 64, 71
  <C-union>, 64
  <C-unsafe-char>, 51
  <C-unsafe-int>, 51
  <C-unsafe-unsigned-char>, 51
  <C-unsafe-unsigned-int>, 51
  <C-unsafe-unsigned-long>, 51
  <C-unsafe-unsigned-short>, 51
  <C-unsafe-unsigned-unsigned-char>, 51
  <C-unsafe-unsigned-unsigned-int>, 51
  <C-unsafe-unsigned-unsigned-long>, 51
  <C-unsafe-unsigned-unsigned-short>, 51
  <C-value>, 50, 52, 53, 64
  <C-void*>, 54
  <C-void>, 50
  <Point>, 43
  <byte-string>, 69
  <machine-word>, 51
defining specialized designator classes, 65, 67
designator, 49
fundamental numeric type designator classes, 51
mapping C types onto, 474
pointer designator, 52
utility designator, 86
clear-memory!, 87
close, 269
close(<file-stream>), 269
coercion-policy, 367
coercion-policy-setter, 367
collect, 100
collect-first, 100
collect-first-into, 100
collect-into, 100
collect-last, 100
collect-last-into, 100
collected, 100
collecting, 100
collection-hash, 106
collector-protocol, 100
colorize-stream, 112
column-name, 368
combining bit mask constants, 478
command-function, 368
commit-transaction, 368
compile-regex, 332
concatenate!, 129
condition-number, 368
condition-to-string, 130
conditional-update!, 231
conditional-update! (extended), 233
conditions-not-recorded?, 368
connect, 368
connect-with-prompt, 369
connect-with-prompt?, 369
Index

connection, 369
connection-name, 369
connection-setter, 369
connections, 370
constraint-catalog, 370
constraint-name, 370
constraint-schema, 370
constraints, 370
content of the Win32 API libraries, 473
cntrol?, 389
control?(<character>), 389
control?(<string>), 390
conventions
mapping conventions for Win32 libraries, 473
naming conventions for Win32 libraries, 473
simple naming, 474
convert-value, 370
copy-bit-set!, 91
copy-bytes, 127
  copy-bytes(<byte-string>, <integer>, <byte-string>,
  <integer>, <integer>), 127
  copy-bytes(<byte-string>, <integer>, <byte-vector>,
  <integer>, <integer>), 127
  copy-bytes(<byte-vector>, <integer>, <byte-string>,
  <integer>, <integer>), 127
  copy-bytes(<byte-vector>, <integer>, <byte-vector>,
  <integer>, <integer>), 127
  copy-bytes(<byte-vector>, <integer>, <simple-object-vector>,
  <integer>, <integer>), 127
  copy-bytes(<sequence>, <integer>, <sequence>,
  <integer>, <integer>), 127
  copy-bytes(<simple-object-vector>, <integer>,
  <byte-vector>, <integer>, <integer>), 127
  copy-bytes(<string>, <integer>, <string>, <integer>), 127
  copy-bytes(<string>, <integer>, <vector>, <integer>), 127
  copy-bytes(<vector>, <integer>, <string>, <integer>), 127
  copy-bytes(<vector>, <integer>, <vector>, <integer>, <integer>), 127
copy-bytes!, 87
copy-file, 451
copy-into!, 87
cos, 173
cosh, 173
count-substrings, 396
create-directory, 452
creating methods from Windows alias functions, 475
current-date, 423
current-process-id, 462
current-thread, 216
current-thread-id, 216
cursor-name, 371

D
d%ceiling/, 160
d%divide, 160
d%floor/, 159
d%round/, 160
d%truncate/, 160
database, 371
database-object-name, 371
database-object-name-setter, 371
datatype-hints, 371
datatype-hints-setter, 371
date-day, 425
date-day-of-week, 425
date-hours, 426
date-microseconds, 426
date-minutes, 427
date-month, 427
date-seconds, 428
date-time-zone-offset, 428
date-time-zone-offset-setter, 429
date-year, 429
dbms, 372
dbms-name, 372
dbms-version, 372
dealing with the C function WinMain, 478
deallocating
  C-storage, 83
debug-assert, 130
debug-message, 130
decimal-digit?, 393
  decimal-digit?(<character>), 393
  decimal-digit?(<string>), 393
decode-date, 431
decode-duration, 431
  decode-duration(<day/time-duration>), 432
  decode-duration(<year/month-duration>), 432
default-connection, 372
default-conversion, 372
default-dbms, 373
default-diagnostics-size, 373
default-handler
  default-handler(<warning>), 131
default-isolation-level, 373
default-last-handler, 132
default-transaction-mode, 373
default-value, 373
define C-address, 83
define C-address definition macro, 46, 83
define C-callable definition macro, 74
define C-callable-wrapper, 78
define C-callable-wrapper definition macro, 45, 77, 78
define C-function, 74
define C-function definition macro, 45, 46, 66, 74, 76, 78
define C-mapped-subtype, 67
Index
equal-memory?, 88
error messages, 476
even?, 10, 149
   even?(<complex>), 10
   even?(<integer>), 11
execute, 375
exit-application, 463
exp, 174
export-C-Dylan-object, 89
export-C-Dylan-object function, 89
export-function
   designator-property, 49
export-type
   designator-property, 49
F
false-or, 134
fields, 376
fields-setter, 376
file-exists?, 454
file-properties, 454
file-property, 454
file-property-setter, 455
file-type, 456
fill-table!, 134
finalize, 187
   finalize(<object>), 188
finalze-when-unreachable, 187
find-diagnostic, 376
find-element, 135
find-substring, 396
float-to-string, 135
force-err, 245
force-out, 245
force-output, 272
format, 242
   format(<byte-string>), 242
format-date, 434
format-err, 245
   format-err(<byte-string>), 245
format-out, 165, 244
   format-out(<byte-string>), 244
format-to-string, 165, 242
   format-to-string(<byte-string>), 242
found?, 143
from-hexstring
   from-hexstring(<byte-string>), 128
function types
   C-ffi, 74
functions
   alignment-of, 50
   C-char-at-setter, 59
   C-double-at, 62
   C-double-at-setter, 62
   C-float-at, 62
   C-float-at-setter, 63
   C-int-at, 62
   C-int-at-setter, 62
   C-long-at, 61
   C-long-at-setter, 61
   C-pointer-at, 63
   C-pointer-at-setter, 63
   C-short-at, 60
   C-short-at-setter, 60
   C-signed-char-at, 58
   C-signed-int-at, 61
   C-signed-int-at-setter, 62
   C-signed-long-at, 60
   C-signed-long-at-setter, 61
   C-signed-short-at, 59
   C-signed-short-at-setter, 60
   C-unsigned-char-at, 59
   C-unsigned-char-at-setter, 59
   C-unsigned-int-at, 61
   C-unsigned-int-at-setter, 61
   C-unsigned-long-at, 60
   C-unsigned-long-at-setter, 60
   C-unsigned-short-at, 59
   C-unsigned-short-at-setter, 59
   check-win32-result, 476
creating methods from Windows alias functions, 475
defining callback, 476
describing Dylan functions for use by C, 78
describing for use by C, 78
ensure-no-win32-error, 476
import-C-Dylan-object, 89
null-pointer, 53
null-pointer?, 54
pointer-address, 53
referenced-type, 54
register-C-Dylan-object, 88
report-win32-error, 476
size-of, 50
unregister-C-Dylan-object, 89
win32-error-message, 476
fundamental numeric type, 51
fundamental numeric type designator classes, 51
G
generic functions
   destroy, 85
   pointer-value, 51, 55
   pointer-value-address, 56
   pointer-value-setter, 51, 55, 56
get-log, 309
get-option-value, 120
Index

H
handling return of multiple values, 475
hexadecimal-digit?, 394
    hexadecimal-digit?(<character>), 394
    hexadecimal-digit?(<string>), 394
hexdigest
    hexdigest(<byte-vector>), 237
    hexdigest(<hash>), 237
hexstring
    hexstring(<byte-vector>), 127
home-directory, 457
host-address, 317
host-name, 317
host-order, 318
    host-order(<ipv4-numeric-address>), 319
hypot, 174

I
ignorable, 136
ignore, 136
ilog2, 176
import-C-Dylan-object, 89
import-C-Dylan-object function, 89
import-function
    designator property, 49
import-type
    designator-property, 49
indent, 273
indexed-table, 376
indexed-table-setter, 377
indexes, 377
indicator-policy, 377
infinite?, 91
inner-stream, 274
inner-stream-setter, 274
input-indicator, 377
input-indicator-setter, 377
install-diagnostic, 377
install-diagnostic-key, 378
installation-functions, 378
integer-to-string, 137
integral?, 11
    integral?(<complex>), 11
    integral?(<machine-number>), 11
introduction to
    C-FFI library, 41
    Win32 libraries, 471
is-null?, 378
isolation-level, 378
isolation-level-setter, 378
isqrt, 175
iterate, 137

J
join, 147
    join(<sequence>, <sequence>), 148
join-thread, 215

K
keyword-sequence, 101

L
level-name, 307
liaison, 378
liaison-setter, 379
libraries
    C-FFI, 40
    introduction to C-FFI, 41
    introduction to Win32 libraries, 471
    Win32 library mapping conventions, 473
    Win32 library naming conventions, 473
    Win32 names excluded from, 480
    Win32 names excluded from Dylan libraries, 480
libraries provided
    list of Win32, 472
limited, 9, 151
list of Win32 libraries provided, 472
list-locator, 440
    list-locator(<file-system-directory-locator>), 441
load-library, 463
local-daylight-savings-time?, 436
local-host, 320
local-host-name, 317
local-port, 320
local-time-zone-name, 436
local-time-zone-offset, 436
locator-address, 441
locator-as-string, 441
locator-base, 441
locator-cgi-string, 447
locator-default-port, 449
locator-directory, 442
locator-error, 442
locator-extension, 442
locator-file, 442
locator-host, 442
locator-index, 447
locator-name, 443
    locator-name(<mailto-locator>), 443
    locator-name(<microsoft-unc-locator>), 443
    locator-name(<microsoft-volume-locator>), 443
locator-path, 444
locator-relative?, 444
locator-server, 444
locator-volume, 444
lock-stream, 275
locked, 230
log, 175
log-additive?, 309
log-additive?-setter, 310
log-debug, 308
log-debug-if, 308
log-enabled?, 310
log-enabled?-setter, 310
log-error, 307
log-formatter, 311
log-formatter-setter, 311
log-info, 307
log-level, 309
log-level-applicable?, 308
log-level-setter, 309
log-message, 307
log-name, 310
log-targets, 309
log-to-target, 312
log-trace, 308
log-warning, 307
login-group, 464
login-name, 464
logn, 175
lowercase, 397
    lowercase(<character>), 397
    lowercase(<string>), 397
lowercase!, 398
    lowercase!(<character>), 398
    lowercase!(<string>), 398
lowercase?, 399
    lowercase?(<character>), 399
    lowercase?(<string>), 399

machine-concurrent-thread-count, 466

make
    make(<byte-string-stream>), 275
    make(<date>), 436
    make(<file-stream>), 276
    make(<sequence-stream>), 277
    make(<string-stream>), 278
    make(<unicode-string-stream>), 278
    make(subclass(<C-pointer>)), 84
make subclass(<C-pointer>) method, 84
make-dbms-specific, 379

mapping
    C structure fields onto Dylan slot names, 475
    C types onto Dylan classes, 474
conventions
    for Win32 libraries, 473
    null value, the, 474
match-group, 335
md5, 237
member?, 92
merge-hash-ids, 4
merge-locators, 444
message-text, 379
messages
    error, 476
methods
    +, 45
    =, 57
    <, 58
creating from Windows alias functions, 475
element, 57
element-setter, 57
make subclass(<C-pointer>), 84
multiple values
    handling return of, 475
multiple-connections?, 379

N
naming conventions
    for Win32 libraries, 473
    simple, 474
negative?, 11, 150
    negative?(<complex>), 11
    negative?(<machine-number>), 11
network-order, 318
    network-order(<ipv4-numeric-address>), 319
new-line, 279
next-dbms-diagnostic, 379
notes on
    C type macros, 64
    translations, 473
null value
    mapping, 474
null-pointer, 53
null-pointer function, 53
null-pointer?, 54
null-pointer? function, 54
nullable?, 379
numeric-host-address, 317

O
object-hash, 5
octal-digit?, 395
  octal-digit?(<character>), 395
  octal-digit?(<string>), 395
odd?, 10, 149
  odd?(<complex>), 10
  odd?(<integer>), 10
one-of, 138
open-locator, 445
open-target-stream, 312
option-parser-definer, 123
option-present?, 120
optional_xep (C function), 198
options in C-FFI macros, 48
organization of the Win32 API libraries, 473
outer-stream, 279
outer-stream-setter, 279
output-indicator, 380
output-indicator-setter, 380
owned?, 221
  owned?(<read-write-lock>), 225
  owned?(<recursive-lock>), 222
  owned?(<simple-lock>), 223
owner-name, 465
owner-organization, 466

P
pad, 408
pad-left, 408
pad-right, 409
parent-process-id, 466
parse-command-line, 119
parse-date-string, 437
parse-iso8601-string, 437
peek, 280
pointer designator classes and related functions, 52
pointer types, 43
pointer-address, 53
pointer-address function, 53
pointer-cast, 53
pointer-type
  designator property, 49
pointer-value, 55
pointer-value generic function, 51, 55
pointer-value-address, 56
pointer-value-address generic function, 56
pointer-value-setter, 56
pointer-value-setter generic function, 51, 55, 56
position, 138
positional-options, 120
positive?, 11, 149
  positive?(<complex>), 11
  positive?(<machine-number>), 11
possible-explanation, 380
pprint-indent, 250
pprint-logical-block, 250
pprint-newline, 251
pprint-tab, 251
primitive-address-add, 201
primitive-address-and, 201
primitive-address-equals?, 201
primitive-address-left-shift, 201
primitive-address-multiply, 201
primitive-address-not, 201
primitive-address-or, 201
primitive-address-right-shift, 201
primitive-address-subtract, 201
primitive-allocate, 200
primitive-allocate-thread-variable, 196
primitive-big-integer-at, 205
primitive-big-integer-at-setter, 205
primitive-bits-as-single-float, 203
primitive-byte-allocate, 200
primitive-byte-character-at, 205
primitive-byte-character-at-setter, 205
primitive-byte-element, 204
primitive-byte-element-setter, 204
primitive-continue-unwind, 200
primitive-copy-vector, 200
primitive-current-thread, 190
primitive-decoded-bits-as-single-float, 203
primitive-destroy-notification, 194
primitive-destroy-recursive-lock, 192
primitive-destroy-semaphore, 193
primitive-destroy-simple-lock, 191
primitive-destroy-thread, 189
primitive-double-float-at, 205
primitive-double-float-at-setter, 205
primitive-element, 204
primitive-element-setter, 204
primitive-equals?, 200
primitive-extended-float-at, 205
primitive-extended-float-at-setter, 205
primitive-false?, 200
primitive-fill!, 204
primitive-function-code, 200
primitive-function-environment, 200
primitive-ieee-double-float-at, 207
primitive-ieee-double-float-at-setter, 207
primitive-ieee-extended-float-at, 207
primitive-ieee-extended-float-at-setter, 207
primitive-ieee-single-float-at, 207
primitive-ieee-single-float-at-setter, 207
primitive-iep-apply, 200
primitive-initialize-current-thread, 189
primitive-inlined-nlx, 201
primitive-int?, 201
primitive-machine-integer-at, 205
primitive-machine-integer-at-setter, 205
primitive-make-box, 200
primitive-make-environment, 200
primitive-make-notification, 194
primitive-make-recursive-lock, 192
primitive-make-semaphore, 193
primitive-make-simple-lock, 190
primitive-make-string, 200
primitive-make-thread, 189
primitive-nlx, 201
primitive-owned-recursive-lock, 193
primitive-owned-simple-lock, 191
primitive-pointer-at, 205
primitive-pointer-at-setter, 205
primitive-read-cycle-counter, 188
primitive-read-return-address, 189
primitive-release-all-notification, 195
primitive-release-notification, 195
primitive-release-recursive-lock, 192
primitive-release-semaphore, 194
primitive-release-simple-lock, 191
primitive-replace!, 204
primitive-replace-bytes!, 204
primitive-signed-16-bit-integer-at, 206
primitive-signed-16-bit-integer-at-setter, 206
primitive-signed-32-bit-integer-at, 206
primitive-signed-32-bit-integer-at-setter, 206
primitive-signed-64-bit-integer-at, 206
primitive-signed-64-bit-integer-at-setter, 206
primitive-signed-8-bit-integer-at, 206
primitive-signed-8-bit-integer-at-setter, 206
primitive-single-float-add, 203
primitive-single-float-as-bits, 203
primitive-single-float-at, 205
primitive-single-float-at-setter, 205
primitive-single-float-divide, 204
primitive-single-float-equals?, 203
primitive-single-float-greater-than-or-equal?, 203
primitive-single-float-greater-than?, 203
primitive-single-float-less-than-or-equal?, 203
primitive-single-float-less-than?, 203
primitive-single-float-multiply, 203
primitive-single-float-negate, 203
primitive-single-float-not-equals?, 203
primitive-single-float-subtract, 203
primitive-single-float-unary-divide, 204
primitive-sleep, 196
primitive-small-integer-add, 202
primitive-small-integer-and, 202
primitive-small-integer-at, 205
primitive-small-integer-at-setter, 205
primitive-small-integer-divide, 202
primitive-small-integer-equals?, 201
primitive-small-integer-greater-than-or-equal?, 202
primitive-small-integer-greater-than?, 202
primitive-small-integer-less-than?, 201
primitive-small-integer-modulo, 202
primitive-small-integer-multiply, 202
primitive-small-integer-negate, 202
primitive-small-integer-not, 202
primitive-small-integer-not-equals?, 201
primitive-small-integer-or, 202
primitive-small-integer-right-shift, 202
primitive-small-integer-subtract, 202
primitive-small-integer-xor, 202
primitive-thread-join-multiple, 190
primitive-thread-join-single, 190
primitive-thread-yield, 190
primitive-true?, 200
primitive-unsigned-16-bit-integer-at, 206
primitive-unsigned-16-bit-integer-at-setter, 206
primitive-unsigned-32-bit-integer-at, 206
primitive-unsigned-32-bit-integer-at-setter, 206
primitive-unsigned-64-bit-integer-at, 206
primitive-unsigned-64-bit-integer-at-setter, 207
primitive-unsigned-8-bit-integer-at, 206
primitive-unsigned-8-bit-integer-at-setter, 206
primitive-unsigned-machine-integer-at, 205
primitive-unsigned-machine-integer-at-setter, 205
primitive-untyped-at, 204
primitive-untyped-at-setter, 204
primitive-variable-lookup, 201
primitive-variable-lookup-setter, 201
primitive-wait-for-notification, 194
primitive-wait-for-notification-timed, 195
primitive-wait-for-recursive-lock, 192
primitive-wait-for-recursive-lock-timed, 192
primitive-wait-for-semaphore, 193
primitive-wait-for-semaphore-timed, 194
primitive-wait-for-simple-lock, 191
primitive-wait-for-simple-lock-timed, 191
primitive-xep-apply, 200
primitive_allocate (C function), 196
primitiveAllocate_vector (C function), 199
primitive_basic_iep_apply (C function), 197
primitive_byte_allocate (C function), 196
primitive_continue_unwind (C function), 197
primitive_copy_vector (C function), 197
primitive_fill_E_ (C function), 197
primitive_fillbytes (C function), 197
primitive_iep_apply (C function), 199
primitive_initialize_vector_from_buffer (C function), 197
primitive_inlined_nlx (C function), 197
primitive_make_box (C function), 198
primitive_make_environment (C function), 198
primitive_make_string (C function), 197
primitive_nlx (C function), 197
primitive_replace_E_(C function), 197
primitive_replace_vector_E_(C function), 197
primitive_xep_apply (C function), 199
print, 247
print-message, 243
  print-message(<condition>), 243
  print-message(<string> or <character>), 244
  print-message(<symbol>), 243
print-object, 249
print-synopsis, 119
print-to-string, 249
printable?, 391
  printable?(<character>), 391
  printable?(<string>), 391
profiling, 166
profiling-type-result, 167
  profiling-type-result(<profiling-state>,cpu-type), 167
  profiling-type-result(<profiling-state>,single-ton(#"allocation")), 167
  profiling-type-result(<profiling-state>,singleton(#"allocation-stats")), 167
properties
  designator type, 49
  put-property!, 101

R
random, 168
range, 9
read, 281
read-element, 281
read-into!, 282
read-line, 283
read-line-into!, 283
read-through, 284
read-to, 285
read-to-end, 285
record-available?, 380
record-coercion-policy, 380
referenced-type, 54
  designator property, 49
referenced-type function, 54
regex-error-pattern, 332
regex-group-count, 333
regex-pattern, 333
regex-position, 333
regex-replace, 333
regex-search, 334
regex-search-strings, 334
register-application-exit-function, 467
register-C-Dylan-object, 88
register-C-Dylan-object function, 88
register-diagnostic-installer, 381
relative-locator, 445
release, 218
  release(<exclusive-lock>), 221
  release(<notification>), 227
  release(<read-write-lock>), 224
  release(<recursive-lock>), 222
  release(<semaphore>), 220
  release(<simple-lock>), 223
release-all, 227
remote-host, 323
remote-port, 323
remove-all-keys!, 108, 139
  remove-all-keys!(<mutable-explicit-key-collection>), 109
  remove-all-keys!(<table>), 109
remove-all-targets, 310
remove-keywords, 102
remove-property!, 102
remove-target, 311
rename-file, 457
replace-substrings, 396
report-win32-error, 477
report-win32-error function, 476
resolve-locator, 445
return of multiple values
  handling, 475
returned-sqlstate, 381
rollback-transaction, 381
root-directories, 458
row-count, 381
rowset-size, 381
run-application, 467
S
schema-from-name, 381
schema-name, 381
scroll-window, 382
scrollable?, 382
sequence-hash, 106
sequence-point, 213
set-add, 92
set-add!, 92
set-complement, 92
set-complement!, 93
set-difference, 93
set-difference!, 93
set-intersection, 93
set-intersection!, 94
set-remove, 94

Index
set-remove!, 94
set-union, 94
set-union!, 94
sha1, 238
sha224, 238
sha256, 238
sha384, 238
sha512, 239
show-progress, 329
simple naming conventions, 474
simplify-locator, 445
sin, 176
sincos, 177
sinh, 177
size, 95
size-of, 50
size-of function, 50
skip-through, 286
sleep, 228
slot names
    mapping C structure fields onto, 475
so%*, 158
so%+, 158
so%-, 158
so%abs, 159
so%negative, 159
so%shift-left, 159
socket-descriptor, 320
specialized designator classes
    defining, 65, 67
split, 146
split-lines, 409
sql, 382
sqrt, 178
start-profiling, 167
start-profiling-type, 167
    start-profiling-type(<profiling-state>,
        profiling-type), 168
    start-profiling-type(<profiling-state>,
        ton(#"allocation")), 168
    start-profiling-type(<profiling-state>,
        singleton(#"allocation-stats")), 168
start-server, 321
start-sockets, 315
start-transaction, 382
starts-with?, 407
statement macros
    with-c-string, 87
    with-stack-structure, 83
statement-column-names, 382
stop-profiling, 168
stop-profiling-type, 168
    stop-profiling-type(<profiling-state>,
        profiling-type), 168
    stop-profiling-type(<profiling-state>,
        singleton(#"allocation-stats")), 168
stream-at-end?, 288
stream-console?, 290
stream-contents, 289
stream-element-type, 289
stream-input-available?, 289
stream-lock, 290
stream-lock-setter, 291
stream-open?, 291
stream-position, 291
stream-position-setter, 292
stream-size, 293
stream-supports-ansi-color?, 113
stream-supports-color?, 113
stream-supports-show-progress?, 329
string-as-locator, 445
string-compare, 404
string-equal-ic?, 405
string-equal?, 404
string-greater-ic?, 406
string-greater?, 405
string-hash, 107
string-less-ic?, 407
string-less?, 406
string-to-integer, 140
strip, 410
strip-left, 410
strip-right, 410
structure fields
    mapping onto Dylan slot names, 475
structure types, 64
    describing, 70
subclass, 140
subdirectory-locator, 446
supplied?, 145
supports-list-locator?, 446
    supports-list-locator?(<file-system-directory-
        locator>), 446
supports-open-locator?, 446
synchronization-name, 218
synchronize-output, 294
synchronize-side-effects, 213

T

table, 109
table-from-name, 383
table-name, 383
table-protocol, 4
tan, 178
tanh, 178
target-pathname, 311
target-stream, 312
temp-directory, 458
terminology in C-FFI library, 48
text, 383
text-attributes, 114
text-setter, 383
thread, 228
thread-id, 215
thread-name, 215
thread-yield, 216
timer-accumulated-time, 169
timer-running?, 170
timer-start, 169
timer-stop, 169
timing, 165
tokenize-command-string, 468
transaction-mode, 384
transaction-mode-setter, 384
translations
  notes on, 473
type designator classes
  fundamental numeric, 51
type properties
  designator, 49
type-for-file-stream, 294
type-for-sequence-stream, 295
types
  C-FFI function, 74
  defining, 65
  describing structure types, 70
  describing union types, 73
  mapping C types onton Dylan classes, 474
  structure, 64
  union, 64
unique-index?, 384
universal-bit-set!, 95
unlock-stream, 296
unread-element, 296
unregister-C-Dylan-object, 89
unregister-C-Dylan-object function, 89
unsupplied, 145
unsupplied?, 145
update-hash, 236
uppercase, 400
  uppercase(<character>), 400
  uppercase(<string>), 400
uppercase!, 401
  uppercase!(<character>), 401
  uppercase!(<string>), 401
uppercase?, 402
  uppercase?(<character>), 402
  uppercase?(<string>), 402
user, 384
utility designator classes, 86

V
value-sequence, 103
values
  handling return of multiple values, 475
  mapping null value, 474
values-hash, 107

W
wait-for, 217
  wait-for(<notification>), 226
  wait-for(<read-write-lock>), 224
  wait-for(<recursive-lock>), 222
  wait-for(<semaphore>), 220
  wait-for(<simple-lock>), 223
wait-for-io-completion, 297
when, 146
whitespace?, 392
  whitespace?(<character>), 392
  whitespace?(<string>), 392
Win32 API, 469
Win32 API libraries, 469
  content of, 473
  introduction, 471
  mapping conventions, 473
  minor details, 479
  naming conventions, 473
  organization of, 473
Win32 names excluded from the Dylan libraries, 480
win32-error-message, 476
win32-error-message function, 476
win32-last-handler, 477
Windows alias functions
  creating methods from, 475
WinMain C function
dealing with, 478
with-c-string, 87
with-c-string statement macro, 87
with-connection, 384
with-database, 385
with-dbms, 385
with-indentation, 297
with-keywords-removed, 103
with-lock, 219
with-open-file, 297
with-server-socket, 321
with-socket-thread, 316
with-stack-structure, 85
with-stack-structure statement macro, 83
with-stream-locked, 298
with-transaction, 385
working-directory, 458
working-directory-setter, 458
write, 300
write-element, 301
write-line, 302
write-message, 313

X
xep (C function), 198
xep_0 (C function), 198
xep_1 (C function), 198
xep_2 (C function), 198
xep_3 (C function), 198
xep_4 (C function), 198
xep_5 (C function), 198
xep_6 (C function), 198
xep_7 (C function), 198
xep_8 (C function), 198
xep_9 (C function), 198

Z
zero?, 11, 149
zero?(<complex>), 11
zero?(<machine-number>), 11