

---

# **Open Dylan Hacker's Guide Documentation**

*Release 0.1*

**Dylan Hackers**

April 29, 2012



# CONTENTS

<b>1</b>	<b>Copyright</b>	<b>3</b>
<b>2</b>	<b>How to contribute to Open Dylan</b>	<b>5</b>
2.1	Getting the Sources . . . . .	5
2.2	Before you commit . . . . .	5
<b>3</b>	<b>Jam-based Build System</b>	<b>7</b>
3.1	Introduction . . . . .	7
3.2	Why Jam-based? . . . . .	7
3.3	Choosing Build Scripts . . . . .	8
3.4	How the Compiler Uses the Build System . . . . .	8
3.5	Automatically-invoked Jam Rules . . . . .	9
3.6	Additional Built-In Jam Rules . . . . .	11
3.7	Editing Jam Files . . . . .	11
<b>4</b>	<b>Open Dylan Compiler Internals</b>	<b>13</b>
4.1	Introduction . . . . .	13
4.2	dfmc-management . . . . .	14
4.3	dfmc-reader . . . . .	14
4.4	dfmc-definitions . . . . .	15
4.5	Excursion into run-time and compile-time . . . . .	17
4.6	dfmc-macro-expander . . . . .	17
4.7	dfmc-convert . . . . .	17
4.8	dfmc-modeling . . . . .	18
4.9	dfmc-flow-graph . . . . .	18
4.10	dfmc-typist . . . . .	18
4.11	dfmc-optimization . . . . .	19
<b>5</b>	<b>Open Dylan Compiler Design</b>	<b>21</b>
5.1	Adding a DFM computation . . . . .	21
5.2	DFM block constructs . . . . .	21
5.3	DFM local assignment . . . . .	24
5.4	DFM multiple values . . . . .	26
5.5	define compilation-pass macro . . . . .	28
<b>6</b>	<b>Open Dylan Runtime Design</b>	<b>31</b>
6.1	Warning . . . . .	31
6.2	The Dylan Implementation Model . . . . .	31
6.3	In-line Call Caches . . . . .	39
6.4	Static Booting . . . . .	39

6.5	FFI . . . . .	39
6.6	Allocation . . . . .	39
6.7	HARP instruction set . . . . .	39
6.8	Compiler Support for Threads . . . . .	39
6.9	Runtime System Functions . . . . .	44
6.10	Compiler Primitives . . . . .	57
<b>7</b>	<b>Library Documentation</b>	<b>71</b>
7.1	Example documentation . . . . .	71
<b>8</b>	<b>Glossary</b>	<b>73</b>
<b>9</b>	<b>Indices and tables</b>	<b>75</b>
	<b>API Index</b>	<b>77</b>
	<b>Index</b>	<b>79</b>

Contents:



# COPYRIGHT

Portions copyright © 1995-1999 Harlequin, Ltd.

Portions copyright © 1995-2000 Functional Objects, Inc.

Portions copyright © 2011 Dylan Hackers.

Companies, names and data used in examples herein are fictitious unless otherwise noted.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the “Software”), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED “AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Other brand or product names are the registered trademarks or trademarks of their respective holders.



# HOW TO CONTRIBUTE TO OPEN DYLAN

## 2.1 Getting the Sources

The first thing you'll need to start contributing is a source checkout of the Git repository. The [Open Dylan sources](#) are hosted on GitHub, along with sources for the [opendylan.org web site](#) and various other repositories. If you don't yet have a GitHub account and ssh keys, now is a good time to get them.

To checkout the main “opendylan” repository:

```
git clone git@github.com:dylan-lang/opendylan.git
```

You will want to fork this repository so you can push changes to your fork and then submit pull requests.

## 2.2 Before you commit

- Please read the [Dylan style guide](#)
- We use [Vincent Driessen's branching model](#) (see [gitflow](#) for tool integration) but we don't separate master from develop branch. Instead we develop in the master branch.
- We also emphasize [this note about git commit messages](#)
- Open Dylan is distributed under the MIT license. We expect contributions to be the same.
- Open Dylan is of collective authorship of the “Dylan Hackers”.



# JAM-BASED BUILD SYSTEM

This document describes the Jam-based build system for Open Dylan.

## 3.1 Introduction

The purpose of the Open Dylan `build-system` is to coordinate the final stages in the building of a Dylan library project. The Open Dylan compiler takes a Dylan library in the form of Dylan source files, and directly generates x86 or PPC machine language object files (in either COFF or ELF format, depending on the target platform). These object files need to be linked together to construct either an executable object file, or a loadable dynamically-linked library (also called a shared library). This link needs to be performed by an external tool such as the Microsoft or GNU linker. The `build-system` component, controlled by a user-specified script file, directs the execution of these external tools.

## 3.2 Why Jam-based?

Use of the Jam scripting language increases the flexibility of Open Dylan for users, and makes it more maintainable by removing hard-coded assumptions spread throughout various layers of the compiler. Previous versions of the Open Dylan `build-system` component permitted users to choose either the Microsoft or GNU Binutils linker on the Win32 platform, and always used the GNU Binutils ELF linker on Linux platforms. New versions of these tools require modifications that go beyond the current flexibility of the `build-system` component, as will new Open Dylan target platforms.

Though the logic of `build-system` (and a former companion library, `linker-support`) was hard-coded, it did allow a limited amount of parameterization using script files. A typical parameterization, from the Microsoft linker script, looked like this:

```
#
linkdll
link /NODEFAULTLIB /INCREMENTAL:NO /PDB:NONE /NOLOGO \
  /ENTRY:$(mangled-dllname)Dll@12 -debug:full -debugtype:cv \
  /nologo /dll /out:$(full-dll-name) kernel32.lib \
  @$ (dllname).link $(objects) $(c-libs) /base:$(base) \
  /version:$(image-version) $(linkopts)
```

Though these script files were poorly documented and insufficiently flexible, they did inspire the introduction of a real scripting language to direct the final stages of compilation in Open Dylan.

**Jam** is a build tool designed by Christopher Seiwald, founder of [Perforce Software](#). It is similar in some ways to `make`, the traditional Unix build tool. However, instead of using only simple declarative rules to define build targets and the dependencies between them, Jam contains a full scripting language, allowing build script authors to define

high-level build instructions that match particular applications. The Jam program also includes Jambase, a library of rules (functions) for building executables and libraries from C, C++, and Fortran sources.

The original Jam tool is a standalone program written in C and YACC. Peter Housel re-implemented the Jam language interpreter and build logic as a reusable Dylan library for use in the Open Dylan `build-system`.

### 3.3 Choosing Build Scripts

Normally you can simply use the build script supplied with Open Dylan that corresponds to the external linker you will be using. The supplied build scripts include the following:

**x86-win32-vc6-build.jam** Build script for Microsoft Visual C++ 6.0.

**x86-win32-vc7-build.jam** Build script for Microsoft Visual C++ .NET.

**x86-win32-mingw-build.jam** Build script for MinGW gcc.

**x86-linux-build.jam** Build script for x86 Linux systems using gcc.

The default build script is `platform-name-build.jam`. You can select a different build script from the Link page of the Environment Options dialog in the IDE, or using the `-build-script` option on the console compiler or console environment command-line.

Build scripts are written using the Jam script language, as described in the [Jam manual page](#). Most Open Dylan build scripts include the `mini-jambase.jam` file, which contains excerpts from the `Jambase` file included with Perforce Jam and described in the [Jambase Reference](#). They can also make use of additional built-in rules defined by the Open Dylan build system, as described in [Additional Built-In Jam Rules](#).

### 3.4 How the Compiler Uses the Build System

When you compile a library, the Open Dylan compiler constructs a new `build` directory and places the generated object files in it. It also constructs a text file called `dylanmakefile.mkf` to be read by the build system. This file contains information imported from the original LID or HDP project file, as well as information generated during compilation. Here is a sample `dylanmakefile.mkf`, in this case the one generated for the `build-system` component itself:

```
comment:      This build file is generated, please don't edit
library:      build-system
base-address: 0x63F20000
major-version: 0
minor-version: 0
library-pack: 0
compilation-mode:      tight
target-type:   executable
files:  library
        paths
        variables
        build
        jam-build
used-projects: functional-dylan
            dummy
            ..\functional-dylan\
            io
            dummy
            ..\io\
            system
```

```
dummy
..\system\
file-source-records
dummy
..\file-source-records\
release-info
dummy
..\release-info\
dfmc-mangling
dummy
..\dfmc-mangling\
jam
dummy
..\jam\
all-c-libraries: advapi32.lib
shell32.lib
```

External files are used to communicate with the build system in order for the information to persist between invocations of the compiler. On the Win32 platform, `dylanmakefile.mkf` files are also copied into the `lib` directory on installation so that other libraries can link against the actual DLL (whose name might not be identical to the library name).

When Open Dylan needs to link a project, it calls the `build-system`, passing the name of the build directory and a list of targets to be built. The build system reads the `dylanmakefile.mkf` file and builds the targets accordingly.

The Open Dylan compiler's project manager expects the build script to define the following pseudo (`NotFile`) targets:

**exports** Describe exports.

**unify-dll** Describe unify-dll.

**dll** Link the project as a dynamically-linked library.

**unify-exe** Describe unify-exe.

**exe** Link the project as an executable program.

**release** Describe release.

**clean-all** Remove build products in the top-level project, and in all of the non-system libraries that it uses.

**clean** Remove build products in the top-level project.

## 3.5 Automatically-invoked Jam Rules

When the build system reads a `dylanmakefile.mkf` file, it invokes several of the Jam rules (functions) defined in the user's build script. These rules in turn register the necessary targets and their dependencies with the Jam build mechanism.

All of the rules described below take *image* as their first parameter; this is a list whose first element is the library name (from the `Library:` keyword of the `.mkf` file) and whose optional second component is the base name of the executable or shared library (from the `Executable:` keyword of the `.mkf` file).

### 3.5.1 `DylanLibrary image : version ;`

Link a Dylan library as a shared library or executable image. This is always the first rule invoked for a given library, and it is usually charged with establishing the library target and setting global and target-specific variables.

The `version` argument normally contains two components, the first obtained from the `Major-version:` keyword of the `.mkf` file, and the second from the `Minor-version:` keyword.

### 3.5.2 `DylanLibraryLinkerOptions image : options ;`

Add the given options to the link command line of the shared library and executable images. The link options provided in the `Linker-options:` keyword of the `.mkf` file are expanded using the usual Jam variable expansion rules before being passed to this rule. (This allows `Linker-options:` keywords in LID and HDP files to refer to platform-specific variables such as `$(guilflags)`).

### 3.5.3 `DylanLibraryBaseAddress image : address ;`

Set the base address of the shared library. The compiler-computed base addresses are probably only usable on the Win32 platform.

### 3.5.4 `DylanLibraryCLibraries image : libraries ;`

Link C (or other externally-derived) libraries into the shared library. The link options provided in the `C-libraries:` keyword of the `.mkf` file are expanded using the usual Jam variable expansion rules before being passed to this rule.

### 3.5.5 `DylanLibraryCObjects image : objects ;`

Link C (or other externally-derived) object files into the shared library.

### 3.5.6 `DylanLibraryCSources image : sources ;`

Link C source files into the shared library.

### 3.5.7 `DylanLibraryCHeaders image : headers ;`

This rule normally does nothing. The `C-header-files:` HDP/LID file is normally used to ensure that files of various sorts (not just C header files) are copied into the build directory.

### 3.5.8 `DylanLibraryRCFiles image : rcfiles ;`

Link Win32 resource files into the shared library and executable.

### 3.5.9 `DylanLibraryJamIncludes image : includes ;`

Not yet implemented.

### 3.5.10 DylanLibraryUses *image* : *library* : *dir* ;

Link other Dylan libraries into the shared library. The *library* argument gives the name of the other library, and the *dir* argument gives the name of the other library's build directory. If *dir* is `system`, then the library is an installed system library.

## 3.6 Additional Built-In Jam Rules

The build system defines the following additional built-in rules.

### 3.6.1 IncludeMKF *includes* ;

Read each of the given `.mkf` files and invoke Jam rules as described in [Automatically-invoked Jam Rules](#).

### 3.6.2 DFMCMangle *name* ;

Mangle the given *name* according to the Open Dylan compiler's mangling rules. If *name* has a single component, it is considered to be a raw name; if there are three components they correspond to the variable-name, module-name, and library-name respectively.

## 3.7 Editing Jam Files

An Emacs major mode for Jam files can be found [here](#).



# OPEN DYLAN COMPILER INTERNALS

## 4.1 Introduction

This chapter is an overview of the libraries involved during compilation, information was gathered while hacking on the compiler. It focuses only on DFMC, the Dylan Flow Machine Compiler, located in `source/dfmc` of the `opendylan` repository.

But first look how to get there: lets consider the command-line compiler, invoked with `-build hello-world`. This is parsed by the executable (in `environment/console/command-line.dylan`, in method `execute-main-command`: this has some code like `if (build?) run(<build-project-command>`,  
....

This `<build-project-command>` is defined in `environment/commands/build.dylan`, there a method `do-execute-command` is specified with a `<build-project-command>` as argument. This runs the method `build-project`, defined in `environment/dfmc/projects/projects.dylan`, calling `compile-library`.

This is defined in `project-manager/projects/compilation.dylan` and calls `parse-and-compile`, which calls `parse-project-sources` (defined in `dfmc/management/definitions-driver`) and `compile-project-definitions`, which is defined in `dfmc/browser-support/glue-routines.dylan`, calling `dfmc-compile-library-from-definitions`, finally defined in `dfmc/management/world.dylan` (here under the name `compile-library-from-definitions`).

To explain these long call chains, we need some more understanding of the different libraries of the compiler: `environment` is the public API, `project-manager` is a bunch of hacks to care about finding the project (by using the registry) and calling the compiler, and the linker (to create a `dll/so` and executable) afterwards. The libraries `dfmc/browser-support` and `environment/dfmc` are the glue from `environment` to DFMC.

The big picture is pretty simple: `management` drives the different libraries, some are the front-end (reader, macro-expander) translating into definitions; some intermediate language (conversion, optimization, typist) which work on the flow-graph; others are back-end, including linker. There is some support needed for the actual runtime, which is sketched in `modeling` (parts of which are put into the dylan runtime library), `namespace` (which handles namespaces and defines the dylan library and its modules).

First we need to introduce some terminology and recapitulate some conventions:

- the unit of compilation is a single dylan library
- the metadata of a library is stored in DOOD, the dylan object-oriented database
- loose (development) vs tight (production): loose mode allows runtime updates of definitions, like adding generic function into a sealed domain, subclassing sealed classes - production mode has stricter checks
- batch compilation: when invoked from command line, or building a complete library

- interactive compilation: IDE feature to play around, adding a single definition to a library

DFMC is well structured, but sadly some libraries use each others, which they shouldn't (typist, conversion, optimization).

In the remainder of this guide, we will focus on a simple example, which prints `Hello` `x` times:

```
define method hello-world (x :: <integer>) => ()
  do (curry (format-out, "Hello %d\n"), range (to: x))
end
```

## 4.2 dfmc-management

The library `dfmc-management` drives the compilation process, prints general information what is happening at the moment (progress, warnings) and takes care of some global settings like opening and closing source records, etc.

The main external entry point is `compile-library-from-definitions` in `world.dylan`. This requires that the source has already been parsed (really? but it calls `compute-library-definitions` itself!).

It then calls in sequence `compute-library-definitions`, `ensure-library-compiled`, `ensure-library-glue-linked`, `ensure-library-stripped` and `ensure-database-saved`, apart from console output (warnings, stats).

The very first method, `compute-library-definitions`, calls `ensure-library-definitions-installed`, which calls `update-compilation-record-definitions`, which mainly calls `compute-source-record-top-level-forms`. This opens the compilation record as a stream and calls `read-top-level-fragment` to get a fragment and then `top-level-convert-forms`.

The reader library defines the `read-top-level-fragment`, the definitions library the `top-level-convert-forms`. Thus, a fragment is read and converted into definitions.

The method `ensure-library-compiled` computes, finishes and checks the data models. Afterwards the code models are generated (the control flow graph), then type inference is done and the optimizer is run. Finally the heaps are generated. These are methods defined in `compilation-driver.dylan`, calling out to the modeling, conversion, flow-graph, typist and optimizer libraries.

Finally the glue is emitted (in `back-end-driver.dylan`) and the database is saved, which contains metadata of each library (like type information, code models, etc.).

Some global warnings for libraries are defined and checked for in the management library.

The unused file `interface.dylan` compares module and binding definitions, in order to judge whether the public API of a library/module has changed between two versions. Usage of this would allow lazy recompilation: only recompile if the API has changed of a linked library.

## 4.3 dfmc-reader

This library reads the Dylan source, tokenizes it, annotates source locations, and builds a parse tree. The parser is in `parser.dylgram`, which uses `app/parser-compiler` to generate `infix-parser.dylan`. The API used is `read-top-level-fragment` and `re-read-fragments`.

Error handling is on the token level, thus a mismatched `end` is noticed. Other sorts of errors are invalid token, integer too large, character too large, ratios not being supported, end of input (while more tokens were required).

Every `<fragment>`, the base class of the abstract syntax tree, has a `compilation-record` and a `source-position`.

So, `read-top-level-fragment` returns the following parse tree:

```

<body-definition-fragment>:
  fragment-macro: <simple-variable-name-fragment>
                                fragment-name: #"method-definer"

  fragment-modifiers: #()
  fragment-body-fragment:
    <simple-variable-name-fragment>:
      fragment-name: #"hello-world"
    <parens-fragment>:
      fragment-left-delimiter: <lparen-fragment>
      fragment-nested-fragments:
        <simple-variable-name-fragment>:
          fragment-name: #"x"
        <colon-colon-fragment>
        <simple-variable-name-fragment>:
          fragment-name: #"<integer>"
      fragment-right-delimiter: <rparen-fragment>
    <simple-variable-name-fragment>:
      fragment-name: #"do"
    <parens-fragment>:
      fragment-left-delimiter: <lparen-fragment>
      fragment-nested-fragments:
        <simple-variable-name-fragment>:
          fragment-name: #"curry"
        <parens-fragment>:
          fragment-left-delimiter: <lparen-fragment>
          fragment-nested-fragments:
            <simple-variable-name-fragment>:
              fragment-name: #"format-out "
            <comma-fragment>
            <string-fragment>:
              fragment-value: "Hello %d\n"
          fragment-right-delimiter: <rparen-fragment>
        <comma-fragment>
        <simple-variable-name-fragment>:
          fragment-name: #"range"
        <parens-fragment>:
          fragment-left-delimiter: <lparen-fragment>
          fragment-nested-fragments:
            <fragment-syntax-symbol-fragment>:
              fragment-value: #"to"
            <simple-variable-name-fragment>:
              fragment-name: #"x"
          fragment-right-delimiter: <rparen-fragment>
      fragment-right-delimiter: <rparen-fragment>
    <semicolon-fragment>

```

NB: the type hierarchy for <body-definition-fragment> is: <definition-fragment>, <macro-call-fragment>, <compound-fragment>, <fragment>, <object>

## 4.4 dfmc-definitions

Once the abstract syntax tree is generated (by the reader), it's time to convert this into definitions, which are the names in dylan. There are several top-level definitions in dylan, namely: binding, class, constant, (copy-down), domain, function, generic, macro, method, module, namespace (library) and variable. Every definition has it's own class, inheriting from <top-level-form> (defined in common/top-level-forms.dylan). A top level form at least contains information about its compilation record, source location, parent form, sequence number and dependencies

and referenced variables. Additional information available are adjectives, the word defined, its library, original library, top level methods. As a side note, dependency tracking is also defined in `common/top-level-forms.dylan`.

The main entry point for the definition library is `top-level-convert` on a fragment, defined in `top-level-convert.dylan`.

The building of definition objects relies heavily on the macro-expander, especially on procedural macros described in *D-Expressions: Lisp Power, Dylan Style* (<http://people.csail.mit.edu/jrb/Projects/dexprs.pdf>). Open Dylan extends the definitions with `compiler`, `optimizer`, `primitive` and `shared-symbols`, mainly used internally in the compiler.

Looking into `define-method.dylan`, we can see a class `<method-definition>`. This is built by the parser, more specifically there is a `define &definition method-definer`, which has two rules to match fragments, whereas the second rule is the error case. The first matches any `define method` syntax and calls `do-define-method` with the arguments. The method `do-define-method` defers the work to helper methods `parse-method-adjectives` and `parse-method-signature`, and instantiates a `<method-definition>` object.

For our small example, `do-define-method` creates a single object:

The result of our small example is:

```
<method-definition>
  private-form-body: <body-fragment>
    fragment-constituents: <prefix-call-fragment>
      fragment-arguments:
        <prefix-call-fragment>
          fragment-arguments:
            <simple-variable-name-fragment>
              fragment-name: #"format-out"
            <string-fragment>
              fragment-value: "Hello %d\n"
          fragment-function: <simple-variable-name-fragment>
            fragment-name: #"curry"
        <prefix-call-fragment>
          fragment-arguments:
            <keyword-syntax-symbol-fragment>
              fragment-value: #"to"
            <simple-variable-name-fragment>
              fragment-name: #"x"
          fragment-function: <simple-variable-name-fragment>
            fragment-name: #"range"
      fragment-function: <simple-variable-name-fragment>
        fragment-name: #"do"
  private-form-signature: <method-requires-signature-spec>
    private-spec-argument-next-variable-specs: <next-variable-spec>
      private-spec-variable-name: <simple-variable-name-fragment>
        fragment-name: #"next-method"
    private-spec-argument-required-variable-specs: <typed-required-variable-spec>
      private-spec-type-expression: <simple-variable-name-fragment>
        fragment-name: #"<integer>"
      private-spec-variable-name: <simple-variable-name-fragment>
        fragment-name: #"x"
  private-form-signature-and-body-fragment: <sequence-fragment>
    <parens-fragment>, <simple-variable-name-fragment>, <parens-fragment>, <semicolon-fragment>
  private-form-variable-name-or-names: <simple-variable-name-fragment>
    fragment-name: #"hello-world"
```

It is noteworthy that still no intra-library information is present, this is top-level Dylan code without any context. All macros are expanded.

## 4.5 Excursion into run-time and compile-time

NB: not sure whether this should be here or somewhere different.

Some objects are defined in the compiler, but are injected into the Dylan world. How does this happen?

So, in the Dylan library you see `// BOOTED`: comments here and there. The source location of well-known basic types and functions is `dylan:dylan-user:boot-dylan-definitions()`.

There is no definition of this specific method.

The method `dfmc-definitions:top-level-convert.dylan: boot-definitions-form?` checks exactly for this name. The method `top-level-convert-forms` behaves differently if `boot-definitions-form?` returns true, namely it calls `booted-source-sequence()`, which is defined in `boot-definitions.dylan`. This method grabs the `boot-record` and returns it sorted as a vector.

But what is a `boot-record` after all? Well, it's definition is all in `boot-definitions.dylan`, with the explanation “records the set of things that must be inserted into a Dylan world at the very start. Some of things are core definitions, such as converters and macros, and these are booted at the definition level. The rest are expressed as source to be fed to the compiler.”

The constant `*boot-record*` is filled by `do-define-core-*`. These are called by `dfmc-modeling`. Namely, primitives (which names and signatures are installed), macros, modules, libraries, classes.

Be aware that the actual implementation of the primitives is in the runtime (either `c-run-time.c` or the `runtime-generator` generates a `runtime.o` containing those definitions), but some crucial bits, like the adjectives (`side-effect-free`, `dynamic-extent`, `stateless` and `opposited`) are in `dfmc-modeling` and used in the optimization!

The core classes are emitted from modeling with actual constructors (be aware that the runtime layout is also recorded in `run-time.h`).

The dylan library and module definitions are in `modeling/namespaces.dylan`.

A noteworthy comment is that a compiler (`comp-0`, generation 0) loads the Dylan library (`dylan-0`), which contains the definitions (`defs-0`). When compiling itself (`comp-1`), first a fresh Dylan library (`dylan-1`) is built, which contains still the old booted definitions (`defs-0`). It emits new definitions (`defs-1`) and a new `boot-record` when dumping `dfmc-definitions`. Now the next generation compiler (`comp-1`) will use these new definitions in the next Dylan (`dylan-2`) library. Beware of dragons.

## 4.6 dfmc-macro-expander

The deep magic happens here.

## 4.7 dfmc-convert

Converts definition objects to model objects. In order to fulfill this task, it looks up bindings to objects from other libraries. Also converts the bodies of definitions to a flow graph. Does some initial evaluation, for example `limited(<vector>, of: <string>)` gets converted to a `<&limited-vector-type>` instance. Thus, it contains a poor-mans eval.

Also, creates `init-expressions`, which may be needed for the runtime, since everything can be dynamic, each `top-level-form` may need initializing which are called when the library is loaded.

Also sets up a lexical environment for the definitions, and checks bindings.

Here, type variables are now recorded into the lexical environment, the type variables are passed around while the signature is checked.

After Dylan code is converted, it is in a representation which can be passed to a backend to generate code. Modeling objects have corresponding compile and run time objects, and are prefixed with an ampersand (<object>).

## 4.8 dfmc-modeling

Contains modeling of runtime and compile time objects. Since some calls are tried to be done at compile time rather than at runtime, it provides these compile time methods with a mechanism to override the runtime methods (`define &override-function`). An example for this is `^instance?`, compile time methods are prefixed with a `^`, while compile and runtime class definitions are prefixed with `&`, like `define &class <type>`.

Also, DOOD (a persistent object store) models and proxies for compile time definitions are available in this library, in order to load definitions of dependent libraries.

## 4.9 dfmc-flow-graph

The flow graph consists of instances of the `<computation>` class, like `<if>`, `<loop-call>`, `<assignment>`, `<merge>`. The flow graph is in a (pseudo) single state assignment form. Every time any algorithm alters the flow graph, it disconnects the deprecated computation and inserts new computations. New temporaries are introduced if a binding is assigned to a new value. Subclasses of `<computation>` model control flow, `<temporary>` (as well as `<referenced-object>`) data flow.

Computations are a doubly-linked list, with special cases for merge nodes, loops, if, bind-exit and unwind-protect. Every computation may have computation-type field, which is bound to a `<type-variable>`. It also may have a temporary slot, which is its return value. Several cases, single and multiple return values, are supported. The temporary has a link to its generator, a list of users and a reference to its value.

Additional (data flow) information is kept in special slots, test in `<if>`, arguments of a `<call>`, etc. These are all `<referenced-object>`, or more specially `<value-reference>`, `<object-reference>`, etc. `<object-reference>` contains a binding to its actual value.

`<temporary>` and `<environment>` classes are defined in this library.

`join-2x1` etc. are the operations on the flow graph.

## 4.10 dfmc-typist

This library contains runtime type algebra as well as a type inference algorithm.

Main entry point is `type-estimate`, which calls `type-estimate-in-cache`. Each library contains a type-cache, mapping from method definitions, etc. to type-variables.

Type variables contain an actual type estimate as well as justifications (supporters and supportees), used for propagation of types.

converts types to `<type-estimate>` objects

`type-estimate-function-from-signature` calls `type-estimate-body` if available (instead of using types of the signature), call chain is `type-estimate-call-from-site -> type-estimate-call-stupidly-from-fn -> function-valtype`

contains hard-coded hacks for `make`, `element`, `element-setter` (in `type-estimate-call-from-site`)

**typist/typist-inference.dylan:poor-mans-check-type-intersection** if `#f` (the temp), optimizer has determined that type check is superfluous

**dfmc/typist-protocol.dylan:151 - does not look sane!**

```
define function type-estimate=?(te1 :: <type-estimate>, te2 :: <type-estimate>)
```

```
  => (e? :: <boolean>, known? :: <boolean>) // Dylan Torah, p. 48: te1 = te2 iff te1 <= te2 &
    te2 <= te1 let (sub?-1, known?-1) = type-estimate-subtype?(te1, te2); let (sub?-2, known?-
    2) = type-estimate-subtype?(te1, te2);
```

## 4.11 dfmc-optimization

This library contains several optimizations: dead code removal, constant folding, common subexpression elimination, inlining, dispatch upgrading and tail call analyzation.

Main entry point from management is `really-run-compilation-passes`. This loops over all lambdas in the given code fragment, converts assigned variables to a `<cell>` representation, renames temporaries in conditionals, then runs the “optimizer”. This builds an optimization queue, initially containing all computations. It calls `do-optimize` on each element of the optimization-queue, as long as it returns `#f` (protocol is, that, if an optimization was successful, it returns `#t`, if it was not successful, `#f`). For different types of computations different optimizations are run. Default optimizations are deletion of useless computations and constant folding. `<bind>` is skipped, for `<function-call>` additionally upgrade (analyzes the call, tries to get rid of `gf` dispatch) and inlining is done. `<primitive-call>` are optimized by `analyze-calls`.

**constant folds (constant-folding.dylan):** // The following is because we seem to have a bogus class hierarchy // here 8( // We mustn't propagate a constraint type above its station, since // the constraint is typically local (true within a particular // branch, say). & `~instance?(c, <constrain-type>)`

optimization/dispatch.dylan: `gf` dispatch optimization

optimization/assignment: here happens the “occurence typing” (type inference for `instance?`)... `<constrain-type>` is only for the `instance?` and conditionals hack



# OPEN DYLAN COMPILER DESIGN

## 5.1 Adding a DFM computation

What you have to do to add a new node class to the DFM:

- Add it to `flow-graph/computation.dylan`, and ensure that you export it from `flow-graph/flow-graph-library`.
- Create the converters to generate it. Likely in conversion, but some nodes are only created by optimizations.
- Make sure all the back ends handle it. This includes, at least:
  - c-back-end
  - debug-back-end – the printer
  - all native back ends
- In addition, it would be good to add any invariant checks to `flow-graph/checker.dylan`.

## 5.2 DFM block constructs

### 5.2.1 bind-exit

First, let's look at an example of bind-exit.

```
block (exit) exit(42); 13 end; =>
  [BIND]
  t2 := [BIND-EXIT entry-state: &t1 body: L1 exit-to: L0]
  L1:
  t4 := ^42
  t12 := exit entry-state: &t1 value: t4
  t6 := ^13
  end-exit-block entry-state: &t1
  L0:
  t7 := [MERGE t2 t6]
  return t7
```

(That's before register assignment, to make the difference in the temporaries used in the merge node clear.)

The `<bind-exit>` node establishes the place the exit jumps to, an `<entry-state>`. This is communicated to `<exit>` and `<end-exit-block>` through the temporary `t1`. The temporary returned by the `<bind-exit>` is set by the exit procedure.

(The printing code shows up one inconsistency: the temporary generated by the <bind-exit> node is actually not live after that point. It's live only if the exit procedure is taken. On the other hand, the entry-state is live after that point. Perhaps which temporary is the generated one from a <bind-exit> node should be exchanged.)

The merge node combines the two temporaries that could contain the result of the <merge> node – t2 by exiting, t6 by falling through. The <end-exit-block> node exists for at least two purposes: to possibly bash the exit procedure or entry state in order to prevent calls outside of its dynamic scope and to stop a thread in the execution engine. It references the entry state in order that it can be found from the <bind-exit> node.

Before we see the compiled code, here's the DFM code after register allocation:

```
block (exit) exit(42); 13 end; =>
  [BIND]
  t2 := [BIND-EXIT entry-state: &t0 body: L1 exit-to: L0]
  L1:
  t1 := ^42
  t3 := exit entry-state: &t0 value: t1
  t2 := ^13
  end-exit-block entry-state: &t0
  L0:
  t2 := [MERGE t2 t2]
  return t2
```

And this is the C code:

```
block (exit) exit(42); 13 end; =>
  D L4988I () {
    D T0;
    D T2;
    D T1;
    D T3;

    T0 = dNprimitive_make_bind_exit_frame();
    if (setjmp(dNprimitive_frame_destination(T0))) {
      T2 = dNprimitive_frame_return_value(T0);
      goto L0;
    }
  L1:
    T1 = I(42);
    dNprimitive_nlx(T0, T1);
  L0:
    return(T2);
  }
```

The only gotcha (other than how setjmp works in C) is that the emission engine knows that there's no point in generating code for the stuff that follows an <exit> node; it's a primitive form of dead code elimination. So that's why the `t2 := ^13` and <end-exit-block> nodes are not emitted.

The call to `dNprimitive_nlx` unwinds all <unwind-protect> frames on the way back to the entry state marked by T0. Eventually, (unless some cleanup calls another exit procedure) it will longjmp to the site of the setjmp. The second argument to `dNprimitive_nlx` is shoved into the `dNprimitive_frame_return_value` of the entry state.

On the other hand, if we omit the call to the exit procedure (or if there's some control flow path which falls through, or if it isn't inlined, as it was above), the generated code is:

```
block (exit) 13 end; =>
  D L1502I () {
    D T0;
    D T1;
```

```

T0 = dNprimitive_make_bind_exit_frame();
if (setjmp(dNprimitive_frame_destination(T0))) {
    T1 = dNprimitive_frame_return_value(T0);
    goto L0;
}
L1:
T1 = I(13);
/* invalidate T0 */
L0:
return(T1);
}

```

Note that the call just falls through from the assignment to T1 to the return; no jump need take place.

The comment about invalidating reflects something I think we should do, but haven't done yet, which is ensure that the exit procedure is bashed when we leave the block. Bashing a single slot should be sufficient.

## 5.2.2 unwind-protect

Now, let's consider the DFM code for an unwind-protect:

```

block () xxx() cleanup yyy() end; =>
  [BIND]
  [UNWIND-PROTECT entry-state: t0 body: L1 cleanup: L2 next: L0]
  L1:
  t1 := ^xxx
  t2 := [CALLx t1()]
  end-protected-block entry-state: t0
  L0:
  return t2
  L2:
  t3 := ^yyy
  [CALLx t3()]
  end-cleanup-block entry-state: t0

```

I think this code is pretty straight-forward, at least in terms of the data flow graph. Note that t2 is live in the code outside the block statement.

```

block () xxx() cleanup yyy() end; =>
  D L2437I () {
    D T0;
    D T1;
    D T2;
    D T3;

    T0 = dNprimitive_make_unwind_protect_frame();
    if (setjmp(dNprimitive_frame_destination(T0)))
      goto L2;
  L1:
    T1 = dNxxx;
    T2 = CALL0(T1);
  L2:
    T3 = dNyyy;
    CALL0(T3);
    dNprimitive_continue_unwind();
  L0:
    return(T2);
  }

```

The `dNprimitive_continue_unwind` just returns in this case. If the cleanup clause were invoked by an exit procedure, it would have set a flag in the frame indicating that it continues non-local-exiting. The important thing to see is that the decision about whether to fall through from the cleanup clause into the code outside the block is made by `dNprimitive_continue_unwind`, based on dynamic information.

### 5.2.3 Final notes

Finally, note that a block with both an exit procedure (`bind-exit`) and a cleanup clause (`unwind-protect`) is simply a `bind-exit` wrapped around an `unwind-protect`.

### 5.2.4 Optimizations

Lots of optimizations can be done. Off the top of my head:

- Code following an `<exit>` is dead; it should be dead-code eliminated in the DFM.
- If an `<exit>` is inlined and there are no `<unwind-protect>`s between it and the `<bind-exit>`, it can be turned into a control transfer.
- If there are no `<exit>`s for a given `<entry-state>`, the `<bind-exit>` node can be removed.

An invalid optimization that had been suggested was to merge nested `<unwind-protect>`s without intervening `<bind-exit>`s with a test in the merged cleanup to determine whether the inner cleanup is still active. This isn't valid because then the inner cleanup is no longer protected by the outer cleanup.

## 5.3 DFM local assignment

We really want the DFM to be a *single assignment* form. That is, all temporaries should be defined and then never mutated. We want this because it makes many optimizations (common sub-expression elimination, inlining, etc) significantly easier. See the usual set of SSA papers for details; I can dig up references.

On the other hand, Dylan has assignment to locals, and we model locals with temporaries. Since the DFM doesn't have cycles (loops), we could replace assignments *to variables which aren't closed over* with new temporaries, in the same way as SSA code is usually generated. But all the interesting cases in Dylan are when assigned variables are closed over, especially because they're assigned to in loop bodies.

Instead, based on Keith's suggestion, I map our Dylan-esque DFM into one that matches how ML, at the language level, with references (mutable variables): all temporaries which are assigned to are replaced with temporaries referring to boxed values.

The current approach:

I introduced three primitives:

```
make-box t => box           // create a box, containing t
get-box-value box => t      // return the value inside the box
set-box-value! box t => t   // set the value inside the box
```

There is a new compiler pass (`eliminate-assignments`) which traverses a DFM graph and does the rewriting.

Here's an example of what happens:

```
begin let a = 13; a := 42; a end; => // before
[BIND]
t0 := ^13
t1 := ^42
```

```

@a := t1
return t0

begin let a = 13; a := 42; a end; => // after
[BIND]
t0 := ^13
t1 := [PRIMOP primitive-make-box(t0)]
t2 := ^42
[PRIMOP primitive-set-box-value!(t1, t2)]
t3 := [PRIMOP primitive-get-box-value(t1)] // tail call
return t3

```

The eliminate-assignments pass should happen before any of the *interesting* optimizations, and should never need to be done twice on the same piece of code.

What remains to be done:

We probably want to turn these primitives into DFM computations before trying to do any optimizations on them.

make-box currently allocates the boxed cell in the heap. It should really allocate the cell either a closure or stack frame, depending on whether the box has dynamic extent. If the temporary the box is bound to (t1 in the example above) is only used as with get-box-value and set-box-value!, then we know that the box has the same extent as that temporary. I don't think that all optimizations will preserve that property, but it will probably be maintained most of the time.

When we have temporaries which aren't closed over, most of the time we should be able to do SSA-like elimination of assignments, rewriting them by introducing new temporaries. For example, assignment inside a conditional can produce something like this

```

begin let a = 1; if (p?) a := 2 else end; a end; =>
[BIND]
t2 := ^1
t8 := [PRIMOP primitive-make-box(t2)]
t9 := ^p?
if (t9) goto L1 else goto L2
L1:
t13 := ^2
t11 := [PRIMOP primitive-set-box-value!(t8, t13)]
L0:
[MERGE t11 t14]
t10 := [PRIMOP primitive-get-box-value(t8)] // tail call
return t10
L2:
t14 := ^&#f
goto L0

```

but that should be easy to turn into

```

[BIND]
t1 := ^p?
if (t1) goto L1 else goto L2
L1:
t2 := ^2
L0:
t4 := [MERGE t2 t3]
return t4
L2:
t3 := ^1
goto L0

```

This sort of optimization, in the absence of cycles, is pretty easy. It may be more work making it happen for loops built up from tail calls, but still not as bad as SSA conversion in general.

## 5.4 DFM multiple values

To represent multiple values, there's a new temporary class in the DFM, `<multiple-value-temporary>`. Multiple values temporaries are not interchangeable with other temporaries; maybe we should introduce a `<simple-temporary>` class for non-multiple-value temporaries, but we can do that later. In the debugging print code, MV temporaries print with a `*` in front of them.

A multiple value temporary is the result of any computation which can produce multiple values, notably a call.

In order to produce efficient code, we have imposed the requirement that at most one MV temporary is live at a time (per thread). This allows us to allocate space for all MV temporaries ahead of time, as part of the calling convention, in the *multiple value area*. It is generally best to think of the multiple value area, which is used to pass multiple values across calls, as a single multiple valued register, which we allocate to the live MV temporary.

When there really is more than one live MV temporary, we must spill and unspill uses. One of the important optimizations is to reduce these spills when the number of values in a MV temporary is known, by extracting them into normal temporaries and repackaging them as an MV temporary when needed as one.

A multiple value temporary has slots which describe the number of required values and whether there are rest values. Types need to be incorporated here, just as with other temporaries. There's also a slot for a normal temporary, which is used when spilling the multiple value temporary.

To manipulate multiple values, there are five new computation classes:

`<values>`

super: `<computation>` slots: fixed-values, rest-value

Creates a `<multiple-value-temporary>` from a set of single value temporaries. For now, a `<values>` node comes from a converter for the *function macro* values; in the future, there should be only one `<values>` node created directly, and the rest created by inlining the function values from the Dylan library. (A similar change needs to be made for `<apply>`.)

```
values(1, 2, 3) =>
  [BIND]
  t0 := ^1
  t1 := ^2
  t2 := ^3
  *t3 := [VALUES t0 t1 t2]
  return *t3
```

`<extract-single-value>`

super: `<computation>` slots: multiple-values, index, rest-vector?

Produces a single-valued temporary from an MV temporary. The index is used to select which multiple value is extracted; the indices are numbered from 0. If `rest-vector?` is true, a vector of the values from index on is returned, rather than just the index. (Perhaps that should be a different `<computation>` class.)

These very commonly follow calls, extracting the single value. They should also appear based on optimizations of let bindings.

```
f(g()) =>
  [BIND]
  t0 := ^f
```

```

t1 := ^g
*t2 := [CALLx t1()]
t3 := *t2 [0]
*t4 := [CALLx t0(t3)] // tail call
return *t4

```

**<multiple-value-call>**

super: <function-call>

Like an <apply> with no fixed arguments and a MV temporary as the single (last) argument. Constructed from `let` declarations which bind multiple values. (This could be used for all lets, but I wanted to wait with that until the multiple value optimizations were in place.)

The most important optimization with these nodes is to upgrade the calls to <simple-call> or <apply> with the shape of the MV temporary argument is known. If it's not known, the simplest code generation strategy is to extract all of the temporary values and transform the call into an <apply>.

```

begin let (a, b) = f(); g(a, b) end =>
  [BIND]
  t3 := ^[XEP lambda 741 [743] (a, b)
  [BIND]
  t0 := ^g
  *t1 := [CALLx t0(a, b)] // tail call
  return *t1
end lambda]
t0 := ^f
*t1 := [CALLx t0()]
*t2 := [MV-CALLx t3(*t1)] // tail call
return *t2

```

**<multiple-value-spill> <multiple-value-unspill>**

super: <temporary-transfer>

These instructions turn an MV temporary into a single-value temporary and vice-versa, for the purpose of maintaining the property that a single MV temporary is live at a time. As much as possible, we should try to avoid these instructions in generated code, which can be done when we know we're dealing with a fixed number of values.

These computations are only generated by the mandatory compiler pass `spill-multiple-values`, which should run after all optimizations have happened. (The reason that it should run afterwards is the spill code can defeat other optimizations and other optimizations can get rid of the need to spill.)

```

block () f() afterwards g() end =>
  [BIND]
  t0 := ^f
  *t1 := [CALLx t0()]
  t3 := [MV-SPIll *t1]
  t2 := ^g
  [CALLx t2()]
  *t4 := [MV-UNSPILL t3]
  return *t4

```

The reason the spill is needed is that the call to `g` tramples over the multiple value area.

In the C run time, there's an extra data structure, MV, as follows:

```
typedef struct _mv {
    int count;
    D value[VALUES_MAX];
} MV;
```

There's one global such thing (Preturn\_values), and one per bind-exit or unwind-protect frame, used for the return value that's being passed around. The ones that live in those frames should probably be shortened to some small number of values (2? 4? 8?) and evacuate to the heap if more multiple values are stored; it's pretty rare, I expect, for a large number of values to appear in an unwind-protect frame, or to be passed back with an exit procedure.

The C code generated for all of these is pretty stupid right now, calling out to primitives in all cases, so I won't bother to present it. I want to get to the task of optimizing multiple values soon. I think that a little bit of optimization will go a long way here.

In the native run-time, we'll pass the first few multiple values and (if there is one) the count in registers. Tony can describe that far better than I can.

## 5.5 define compilation-pass macro

NOTE: this is currently not used at all - it had been dropped before going open source, but in general I (hannes) believe it is a good idea (and plan to revive it), thus I keep the documentation.

I've now replaced the old mechanism for specifying compilation passes in the DFM compiler (setting the vector *compilation-passes* in *compile.dylan*) with a declarative system, based around a macro, *define compilation-pass*.

The macro is exported by *dfmc-common*, so every module should have it. The basic idea is that you put a compilation-pass definition in the same place as you define the main entry point for a compiler-pass; the definition includes things about the pass, such as when its run, how it is called, and if it should cause other passes to run.

First, a simple example:

```
define compilation-pass eliminate-assignments,
  visit: functions,
  mandatory?: #t,
  before: analyze-calls;
```

This defines a pass named *eliminate-assignments*, which runs before *analyze-calls* is run; it is possible to use arbitrarily many *before:* options. The *mandatory* option declares that the pass is part of optimization level 0; that is, it's always run.

The *visit: functions* option says that the function is called for every function in the form being compiled. The default is *visit: top-level-forms*, which corresponds to the previous behavior.

```
define compilation-pass try-inlining,
  visit: computations,
  optimization: medium,
  after: analyze-calls,
  before: single-value-propagation,
  triggered-by: analyze-calls,
  trigger: analyze-calls;
```

The *visit: computations* option says that every computation (in the top-level and all nested lambdas) is passed to the pass's function. The *after:* option is like *before:* in reverse.

The *trigger:* option runs the named pass if the pass being defined reports that it changed anything. If the triggered pass has already run, then it is queued to run again; if the triggered pass is disabled or of a higher optimization level than currently being used, it's not run. *Triggered-by:* is *trigger:* in reverse.

A pass function reports that it changed something by returning any non-false value.

Full catalog of options:

**visit: What things to pass to the pass's function:** top-level-forms Just the top-level function. functions Every function. computations Every computation in every function.

**optimization: What level of optimization to run this pass for?** (Choices: mandatory, low, medium, high.)

mandatory?: Always run this pass; overrides optimization:.

before: Run this pass before the named one. after: Run this pass after the named one.

trigger: If this pass changed something, run the named pass. triggered-by: If the named pass changes something, run this pass.

print-before?: Print the DFM code before calling the pass. print-after?: Print the DFM code after the pass is done. print?: Same as print-before?: #t and print-after?: #t.

check-before?: Call ensure-invariants before calling the pass. check-after?: Call ensure-invariants after the pass is done. check?: Same as check-before?: #t and check-after?: #t.

back-end: Turn pass on for the named back end. (Default: all) exclude-back-end: Turn pass off for the named back end. (Default: none.)

disabled?: Turn pass off; overrides everything else.

Convenience functions:

trace-pass(pass-name) untrace-pass(pass-name)

Turns on (or off) printing and checking (both before and after) for the pass.

untrace-passes()

Calls untrace-pass for all traced passes.

Global state:

The fluid-variable *optimization-level* is meant to be a gross control of how much optimization is done. The constants

```
define constant $optimization-mandatory = 0;
define constant $optimization-low      = 1;
define constant $optimization-medium   = 2;
define constant $optimization-high     = 3;

define constant $optimization-default  = $optimization-medium;
```

are defined and correspond to the optimization: option in the define compilation-pass macro.

The fluid-variable *back-end* is used with the options back-end: and exclude-back-end:.

The fluid-variable *trace-compilation-passes* will print a message about each pass as it runs, and report when one pass triggers another.



# OPEN DYLAN RUNTIME DESIGN

Tony Mann, Jonathan Bachrach  
Harlequin, Ltd.  
4 December 1995

## 6.1 Warning

Parts of this document were originally written for the GLUE project, and were aimed at a target audience which is not Dylan literate. There are vestigial patronizing references in some of these sections.

## 6.2 The Dylan Implementation Model

### 6.2.1 Object Representation

Harlequin's implementation achieves dynamic typing of Dylan objects by associating the type with an object based on tagging.

In many circumstances, the Dylan compiler can statically determine the type of an object. This knowledge can be used to select an alternative representation which is more efficient than the canonical representation. For example, the canonical representation of a double float object in Dylan is as a pointer to heap-allocated storage which contains the IEEE bit pattern of the double float in addition to a reference to the Dylan class object `<double-float>`. The compiler may choose to represent the value as a direct bit pattern, wherever this does not violate the semantics of the program.

#### Tagging Scheme

All Dylan values are represented as data of the same size, the size of a pointer. The bit pattern of these values contains tag bits which indicate whether the value is actually a pointer, or whether it is a direct value. Since there are three major groups (integers, characters and everything else), the representation for all platforms is to use two bits.

Tag Bits	Type
00	Heap Allocated
01	Integers
10	Characters
11	Unused

## Integers and Characters

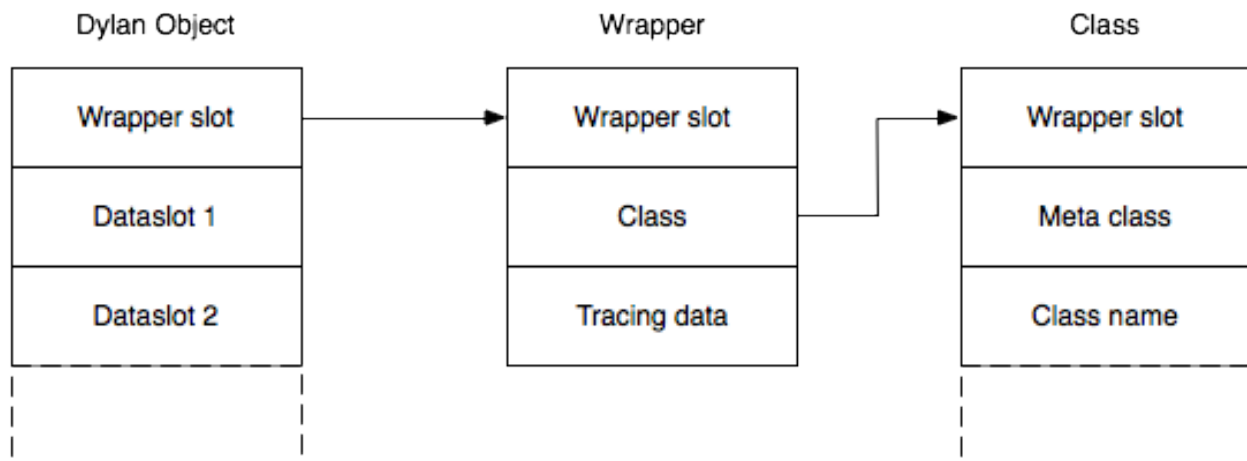
Integers and characters are represented as direct values, using the tag bits as the only indication of type. The tagging scheme uses the least significant two bits. With this scheme, a character or integer is converted to its untagged representation by arithmetic right shifting by two bits. Similarly the conversion from an untagged to a tagged representation is to shift left and add in the tag bits.

Operations on these values (e.g., addition, or other arithmetic operations) are always performed on the untagged representation. This is sub-optimal, because it is possible to perform arithmetic operations directly on the tagged values. It is planned to improve this mechanism at a later date, along with a revision of the tagging scheme.

## Boxed Objects

Apart from integers and characters, all Dylan objects are indirectly represented as *boxed* values (that is, they are pointers to heap allocated boxes). The runtime system is responsible for ensuring that these boxed values are appropriately tagged, because the runtime system provides the allocation service, and must ensure appropriate alignment.

Boxed objects are dynamically identified by their first slot, which is an identification wrapper. This identification wrapper (itself a boxed Dylan object) contains a pointer to the class of the object it is wrapping, as well as some encoded information for the garbage collector about which slots should be traced.



## Boxed Objects

Note that two indirections are necessary to find the class of an object. In practice, this is a rare operation, because almost all dynamic class testing within Dylan is implicit, and the implementation can use the wrapper for these implicit tests. Note that there is potentially a many-to-one correspondence between wrapper objects and class objects.

The Dylan compiler builds literal boxed objects statically whenever it can. In practice, this will include most function objects apart from closures, virtually all wrappers, and most class objects, as well as strings, symbols, and literal vectors and lists.

## Variably Sized Objects

Variably sized objects, such as strings, vectors and arrays, are boxed objects which contain a *repeated slot*. The repeated slot is implemented as a variably sized data area preceded by a normal slot containing the size of the variably sized data represented as a tagged integer. The size slot is used at the Dylan language level to determine the size of

the array. There is also a special encoding for it in the tracing data of the wrapper so that the memory manager knows how to trace the repeated data. For example, an instance of the `<byte-string>` `"foo"` is represented as in:

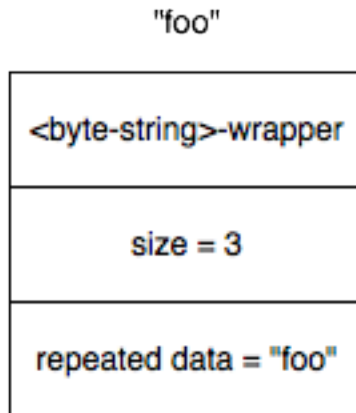


Figure 6.1: An Instance of `<byte-string>`

## Function Objects

Dylan provides two built-in classes of functions: `<generic-function>` and `<method>`. These both obey the same general purpose calling convention, but also support specialized calling conventions (described below) which the compiler may use depending on the detail of its knowledge about the function being called, and the circumstances. Slots in the function object point to the code which implements each convention.

All functions also have a slot which encodes the number of required parameters the function accepts, and whether the function accepts optional or keyword parameters. Another slot in each function object contains a vector of the types which are acceptable for each required parameter. These slots are used for consistency checking of the arguments.

Generic functions have further slots which support the method dispatching process — including a slot which contains a vector of all the methods belonging to the generic function, and a slot which contains a cache of sorted applicable methods for combinations of arguments which have been processed before.

Methods may be closures, in which case a slot in the method object contains the environment for the method, which is represented as a vector of closed-over variables. If the variable is known actually to be constant, then the constant value is stored directly in the vector. Alternatively, if there is any possibility of an assignment to the variable, then the value is stored with an extra indirection to a *value cell*, which may be shared between many closures with related environments.

## 6.2.2 Calling Convention

### Some terminology

Arguments passed to a function at the implementation level fall into 2 different groups. *Language parameters* correspond to the explicit arguments in the source code. *Implementation parameters* correspond to the house-keeping information used by the implementation.

The overall calling convention consists of several specific conventions with different properties, described below. Each convention is implemented by a separate *entry point*. There are partial orderings between the entry points for these conventions, depending on how specific each one is. The code which implements a control flow from one entry point to the next may be obliged to rearrange parameters (e.g. on the stack). This process is called *stack fixing*.

## The register model

Three registers are used within the calling convention to support the passing of *implementation parameters*: Note that for the C backend, global (or thread-local) variables might be used instead of real registers to pass these parameters.

Register	Purpose
arg-count	number of args passed
function	the <code>&lt;function&gt;</code> object being called
mlist	the next-method list ( <code>#f</code> for direct-entry)

## The argument passing conventions

For each of the conventions, arguments are pushed onto the stack in reverse order (i.e. the rightmost argument is pushed first). The leftmost (or leftmost few) arguments are passed in registers. This has a possible disadvantage from the opposite ordering in terms of the need for temporary variables to hold interim results for order-of-evaluation reasons. In practice, the disadvantages will be small because:

- Many arguments to functions are expected to be simple expressions (like constants or variable references) - so order of evaluation does not normally matter.
- On a RISC implementation, we won't want to push each argument anyway - instead it will be more efficient to allocate enough stack space for the call, and store each argument when it's available. This works well with a conservative GC - but it might be poor with a total GC.

This calling convention has the following advantages:

- required arguments can always be found at a known offset from a stack or frame pointer for any of the calling conventions
- optional arguments appear in the same order in memory as they would if vectored up as `#rest` parameters
- Stack allocating the optional arguments as vectors is almost trivial.

For the native code implementation, the callee is responsible for popping any arguments from the stack. This is always possible (even with dynamically sized optional args), because the `argcount` is available to say how many arguments were passed. This is not possible for the C backend - and this is the only substantial difference from the C arg passing convention.

## Calling Convention Goals

*Internal entry points* should be as efficient as possible. I.e. there should not be any constraints on them because Dylan is a dynamic language.

1. There must be a consistent convention for all functions at the *external entry point*, so that functions can be called without the caller having any knowledge of what they are.
2. The code which is executed at external entry points should be shared by all functions with similar properties / lambda-lists.
3. The design should make the path from the external entry point to the internal entry point as simple as is reasonably possible.

## The External Entry Point Convention

All Dylan function objects support the *external* convention. Each function object has an *XEP* slot containing the code to support this convention. External entry points are used for all unoptimized, normal calls to functions. This includes

direct calls to methods and generic functions. Of course, whenever the compiler can use a more efficient entry point instead, then it will.

The registers are used as follows:

Register	Purpose
argcount	number of arguments
function	the function object
mlist	not used

If the function has a complex lambda list (with `#rest` or `#key`), then the external entry code will be one of a standard set of stack fixing functions. This stack fixer will make use of information in the function register to determine which keys to look for, whether the arg-count is legal, whether the arguments have appropriate types etc. The stack fixer will then tail jump to the internal entry point (again, found from the function object). This mechanism requires 2 transfers of control (caller -> stack-fixer -> callee).

For example, consider the following Dylan code:

```
define method func1 (a, b, #rest optionals, #key key1, key2)
end method;
func1(1, 2, key2: 99);
```

For the call to `func1`, above, the parameters are described in the following table:

XEP Parameters for the Call to `func1`

XEP Parameters	Values
language parameters	1, 2, #"key2", 99
<i>argcount</i>	4
<i>function</i>	generic function <code>func1</code>

## Internal Entry Point Convention

The IEP convention uses a fixed number of language parameters, corresponding to each of the parameters of the function (5 in the case of `func1`, above, corresponding to `a`, `b`, `optionals`, `key1`, `key2`). In addition, there are two implementation parameters:

- *mlist*, a list of the next applicable methods to call if the function is a method called from a generic function (this parameter is used to support calls to *next-method*). If the function is not being called from a generic function, the value is `#f` (false).
- *function*, the Dylan function object being called (as for the XEP).

The implementation parameters are not obligatory for all IEP code. It is only necessary to pass *mlist* if the function contains a call to *next-method*. It is only necessary to pass *function* if the function is a closure (because the value is used by the IEP code to locate the environment of the closure). If the IEP is called from the XEP code, both the implementation parameters will always be set, even though they may not be necessary. For the same call to `func1`, above, the parameters are described in '[<runtime.htm#12946>](#)'.

IEP Parameters for the Call to `func1`

IEP Parameters	Values
language parameters	1, 2, optionals, #f, 99
<i>mlist</i>	#f
<i>function</i>	generic function <code>func1</code>

Note that the language parameters now correspond to the formal parameters of the function, whereas, for the XEP, they corresponded to the supplied arguments.

The value of *optionals* in the set of language parameters is the Dylan vector `#[#"key2", 99]` which corresponds to all the optional arguments. The language parameter corresponding to `key1` is `#f`, because the keyword `"key1"` was not supplied. However, the language parameter corresponding to `key2` is `99`, because `"key2"` was supplied with that value.

### The Method Entry Point Convention

All `<method>` objects support the *method entry point* convention. Each method object has an *MEP* slot containing the code to support this convention. When a method is called by a generic function (or via `next method`), the caller uses a dedicated entry point (available from the function object). If the method accepts `#key` or `#rest` parameters, then the method is called with a (possibly stack-allocated) vector representing the optional args. This vector appears as a single extra required argument.

If the method accepts `#key` parameters, then the method entry point will process the supplied keywords - stack fixing them so that they appear as required arguments. It will then tail-call the internal entry point.

If the method does not accept `#key`, then the method entry point is the same as the internal entry point.

## 6.2.3 Special Features

### Introduction to `bind-exit` and `unwind-protect`

The following sections describe the implementation for the native code compiler, only.

`bind-exit` and `unwind-protect` are represented on the stack as frames which contain information about how to invoke the relevant continuation. `unwind-protect` frames are also chained together, and the current environment of existing `unwind-protects` is available in `%current-unwind-protect-frame`.

There are primitives to build each type of frame, and also to remove `unwind-protect` frames (`bind-exit` frames just have to be popped - so that is done inline). The primitive which removes `unwind-protect` frames in the fall-through case is also responsible for invoking the cleanup code (which is called as a sub-function in the same function frame as its parent).

There are also primitives to do non-local exits (*NLX*). These are passed the address of the `bind-exit` frame for the destination, and also the multiple values to be returned. As part of the *NLX*, any intervening `unwind-protects` are invoked and their frames are removed. Multiple-values are saved around the `unwind-protects` in the `bind-exit` frame of the destination.

### `unwind-protect`

An *unwind-protect* frame (*UPF*) looks as follows:

Offset	Value
8	address of start of cleanup code
4	frame pointer
0	previous <code>unwind-protect</code> frame

The compiler compiles `unwind-protect` as follows:

```
let frame = primitive-build-unwind-protect-frame(tag1);
do-the-protected-forms-setting-results-as-for-a-return();
primitive-unwind-protect-cleanup();
goto(tag-finished);
tag1:
  do-the-cleanup-forms();
```

```
end-cleanup(); // inlined as a return instruction
tag-finished:
```

If the protected body exits normally, then *primitive-unwind-protect-cleanup* is called (in the runtime system). This causes the unwind-protect frame to be unlinked from the chain, and the cleanup code to be invoked, as a subroutine call within the same function frame as the protected body. The cleanup code finishes by executing a return instruction. The runtime system ensures that any multiple values are restored, and returns control to the compiled code, which then executes the code following the unwind-protect.

If the cleanup code is invoked because of an NLX, then the runtime function finds the ultimate destination *bind exit frame (BEF)* from the UPF. The runtime function then passes this BEF to another runtime function (as for *bind-exit*) to test whether there are any further intervening cleanups, or to transfer control to the ultimate destination if not.

## bind-exit

A *bind-exit* frame (*BEF*) looks as follows:

Offset	Value
52	continuation address
48	frame pointer
44	current unwind-protect frame
4	space for stack-allocated vector for up to 8 multiple values
0	pointer to saved multiple values as a vector

The compiler compiles *bind-exit* as follows:

```
let frame = primitive-build-bind-exit-frame(tag1);
let closure = make-bind-exit-closure(frame);
do-the-bind-exit-body-setting-results-as-for-a-return();
tag1:
```

During an NLX, multiple-values will be saved in the frame if an intervening unwind-protect is active. The frame itself contains space for 8 values. If more values are present, then they will be heap allocated.

When an NLX occurs, the transfer of control is implemented by a call into the runtime system, passing the pointer to the BEF as a parameter. The runtime function first checks whether there is an intervening cleanup, by testing whether the target dynamic environment in the BEF matches the current global dynamic environment. If there is no intervening cleanup, then control is transferred to the destination of the BEF. Alternatively, if there is an intervening cleanup, then the ultimate destination field of the current UPF is set to the destination BEF, and the cleanup code is invoked within a loop which repeatedly tests for further intervening unwind-protect frames until no more are found.

## Multiple Values

The current implementation of multiple values supports Common Lisp semantics. It is about to be replaced by a new version which which support the new Dylan semantics.

Harlequin's current implementation uses a register to return a single Dylan value, as this is the only value that is used by almost all callers. In addition, each function returns a count of the number of values being returned. This count can be examined by the caller, if required, to determine how many values were returned. If a function is returning more than one value, the additional values are stored in a global (thread-local) area, where the caller may retrieve then, if desired. On RISC architectures, the multiple value count is returned in a register. For the x86 architecture, the *direction flag* register is set / unset to specify whether single / multiple values are being returned, respectively. If multiple values are being returned, then the count of the values is stored in a global (thread-local) location.

Documentation for the new version will be available shortly. Until then, here's an overview:

Functions which return a fixed number of return values just return those values, without returning a count. The first few values will be returned in registers (an architecture-specific number), and remaining values will be returned in a thread-local overflow area. If a function always returns zero values, then no code need be executed to indicate this fact.

Functions which return a dynamically-sized number of values return their values as above, but also return a count of the number being returned in a register. If a function dynamically happens to return zero values, then the return count will be set to zero, but the value `*#f*` will be returned as if it were the first return value.

If the caller of a function can statically determine the number of return values (i.e. at compile-time), then it need perform no checks. However, if the caller has no knowledge of the function being called, then it must check the properties of the callee function object to determine whether the static or dynamic convention is being used, and may then need to read either the dynamic return value count, or the static count in the properties of the function object.

This design has some interesting implications for tail-call optimization. A function can simply tail another function only if both the following rules apply:

1. The callee is known to return at least as many values as the caller, and they have appropriate types.
2. If the caller returns a dynamically-sized number of values, then the callee must too.

## 6.2.4 Name Mangling

In Dylan, unlike C, identifier names are case insensitive. Dylan also permits additional characters to appear in names. As a further complication, Dylan provides multiple namespaces, and the namespaces are controlled within a two-tier hierarchy of modules and libraries.

In order to make it possible to link Dylan code with tools designed to support more traditional languages, the Dylan compiler transforms the names which appear in Dylan programs to C compatible names, via a process called *mangling*.

The library, module and identifier names are each processed, according to the following rules:

1. All uppercase characters are converted to lowercase.
2. Any character which appears on the left-hand side of the table is mapped to the new character sequence accordingly.

Old	New	Comment
-	_	dash
!	_E_	exclamation
\$	_D_	dollar
*	_T_	times
/	_S_	slash
<	_L_	less
>	_G_	greater
?	_Q_	question mark
+	_PL_	plus
&	_AP_	ampersand
^	_CR_	caret
_	_UB_	underbar
~	_SG_	squiggle
	_SP_	space

Finally, the fully mangled name is created by concatenating the processed library, module, and identifier names respectively, separated by X.

For example, the Dylan identifier `add-new!` in module `internal` of library `dylan` would be mangled as `dylanXinternalXadd_new_E_`.

## 6.3 In-line Call Caches

No documentation available here about this yet.

## 6.4 Static Booting

No documentation available here about this yet.

## 6.5 FFI

No documentation available here about this yet.

## 6.6 Allocation

No documentation available here about this yet.

## 6.7 HARP instruction set

No documentation available here about this yet.

## 6.8 Compiler Support for Threads

### 6.8.1 Dylan Portability Interface

The Simple Threads Library is designed for implementation using different threads APIs from common operating systems, including Unix and Windows. Harlequin's implementation of the library is designed to be directly portable onto these operating systems. This portability is achieved by using primitive operations defined within our runtime system. Each primitive operation must be implemented specially for each operating system.

The set of portable primitive operations is collectively called the *portability layer*. The Dylan compiler has special knowledge of the portability layer via primitive function definitions and some specialized emit methods for flow-graph node types which are specific to threads.

### Portability and Runtime Layers

The design assumes that each of the concrete classes of the Simple Threads Library (`<thread>`, `<simple-lock>`, `<recursive-lock>`, `<semaphore>` and `<notification>`) corresponds with an equivalent lower-level feature provided directly by either the operating system or the runtime system. The Dylan objects which are instances of these classes are implemented as *containers* for handles corresponding to low-level (non-Dylan) objects. The Dylan objects contain normal Dylan slots too, and these are directly manipulated by the Dylan library. However, the slots containing the low-level handles may only be manipulated via primitive function calls. For each of the classes, primitive functions are defined to both create and destroy the low-level handles, as well as to perform the basic functions of the class, such as *wait-for* and *release*. The platform-specific implementation of these primitive functions is free to

choose any representation for these handles, provided that it is the same shape as a Dylan slot (which is equivalent to C's *void \**).

As with all Dylan objects, the container objects defined by the threads library are subject to automatic memory management, and possible relocation by the garbage collector. The contents of the container slots will be copied during such a relocation — but the values they contain will not be subject to garbage collection or relocation themselves.

The portability layer provides no direct support for the *fluid-bind* operation. The library implements a *fluid-variable* as a thread-local variable, and uses the high-level Dylan construct *unwind-protect* [also called *cleanup* in Dylan's infix syntax] to manage the creation and deletion of new bindings.

The portability layer includes support for conditional update of atomic variables, as well as assignment. The implementation mechanism for these is not defined, but it is hoped that many platforms will provide direct hardware support for this operation. Where hardware support is not available, the low-level implementation may choose to use a lock to protect conditional updates and assignments, as a fall back option. It is assumed that atomic variables may always be read as normal variables.

[Implementations of Dylan Thread Interfaces](#) shows the expected mapping between the concrete Dylan classes and low-level operating system features, for three of the most popular general-purpose operating systems.

### Implementations of Dylan Thread Interfaces

Dylan Interface	Unix Implementation	Win32 Implementation
<thread>	thread	thread
<simple-lock>	mutex	critical region
<recursive-lock>	mutex	critical region
<semaphore>	semaphore	semaphore
<notification>	condition variable	event
fluid-variable	thread-local variable	thread-local variable
conditional-update!	mutex	exchange instruction (using a guard value as a lock);

### Dylan Types for Threads Portability

Three Dylan types merit discussion for their use with portability primitives: <thread>, <portable-container>, and <optional-name>. Objects that are instances of the <thread> and <portable-container> classes have slots which contain lower-level objects that are specific to the Dylan runtime or operating system. The <optional-name> type allows an object, such as a lock, to have a name represented as a string or, if no name is supplied, as the Boolean false value #f.

<thread>

[Class]

A Dylan object of class <thread> contains two OS handles. One of these represents the underlying OS thread, and the other may be used by implementations to contain the current status of the thread, as an aid to the implementation of the join state.

<portable-container>

[Class]

The <portable-container> class is used by the implementation as a superclass for all the concrete synchronization classes (<simple-lock>, <recursive-lock>, <semaphore>, and <notification>). Each <portable-container> object contains an OS handle, which is available to the runtime for storing any OS-specific data. Subclasses may provide additional slots.

<optional-name>

## [Type]

This is a union type which is used to represent names of synchronization objects. Values of the type are either strings (of class `<byte-string>`) or false (`#f`).

Various classes of Dylan objects are passed through the portability interface, and hence require description in terms of lower level languages. [Correspondence Between Dylan Types and C Types](#) maps the layout of these Dylan objects onto their C equivalents, which are used by runtime-specific implementations of the portability layer.

In general, all Dylan types can be thought of as equivalent to the C type `D`, which is in turn equivalent to the C type `void*`. Of course, runtime-specific implementations of the portability layer must have access to relevant fields of the Dylan objects on which they operate. The type definitions in [Correspondence Between Dylan Types and C Types](#) give implementations access to fields needed for specific types. These definitions are not necessarily complete descriptions of the Dylan objects, however. The objects may contain additional fields that are not of interest to the portability layer, and subclasses may add additional fields of their own.

## Correspondence Between Dylan Types and C Types

Dylan Type	C Type	C Type Definition
<code>&lt;object&gt;</code>	<code>D</code>	<code>typedef void* D;</code>
<code>&lt;small-integer&gt;</code>	<code>DINT</code>	<i>platform specific (size of void*)</i>
<code>&lt;function&gt;</code>	<code>DFN</code>	<code>typedef D&gt;(*DFN)(D, int, ...);</code>
<code>&lt;simple-object-vector&gt;</code>	<code>SOV*</code>	<code>typedef struct _sov { **D class; **DINT size; D data[ ]; } SOV;</code>
<code>&lt;byte-string&gt;</code>	<code>B_STRING*</code>	<code>typedef struct _bst { **D class; **DINT size; char data[ ]; } B_STRING;</code>
<code>&lt;optional-name&gt;</code>	<code>D_NAME</code>	<code>typedef void* D_NAME;</code>
<code>&lt;portable-container&gt;</code>	<code>CONTAINER*</code>	<code>typedef struct _ctr { **D class; **void* handle; } CONTAINER;</code>
<code>&lt;thread&gt;</code>	<code>D_THREAD*</code>	<code>typedef struct _dth { **D class; **void* handle1; void* handle2; } D_THREAD;</code>

## 6.8.2 Compiler Support for the Portability Interface

### The Compiler Flow Graph

The front end of the compiler parses Dylan source code and produces an intermediate representation, the Implicit Continuation Representation (ICR). The ICR is a directed acyclic graph (DAG) of Dylan objects. A *leaf* in the ICR represents a basic computational object, such as a variable (of class `<variable-leaf>`) or a function (of class `<function-leaf>`). A *node* in the ICR represents an operation such as assignment (class `<assignment>`), conditional execution (class `<if>`), or a reference to a leaf (class `<reference>`).

In mapping Dylan code to the ICR, the compiler uses a set of *converters*, which perform syntactic pattern matching against fragments of Dylan code and generate the ICR corresponding to the matched code. For example, when the compiler encounters a top-level variable definition (introduced by the Dylan *define variable* construct), the converter for *define variable* creates a new instance of `<global-variable-leaf>` in the ICR to represent this variable and to record data such as its name, initial value, and typing information.

The back end of the compiler traverses the flow graph and emits code in the target language for compiler output. Methods in the back end specialize on node and leaf classes to enable them to produce the appropriate output.

## Compiler Support for Atomic and Fluid Variables

The portability layer provides support for atomic variable access and for Dylan fluid variables (implemented as thread-local variables). Atomic variables and thread variables are directly represented in the flow graph, where they are subject to dataflow analysis. The variables themselves appear as leaves in the graph.

Because both atomic and fluid variables need special treatment when they are accessed, the back end must emit output that is different from that for accessing other kinds of variables. The compiler defines two specialized classes of leaf for the ICR, `<atomic-global-variable-leaf>` (corresponding to atomic variables) and `<fluid-global-variable-leaf>` (corresponding to fluid variables). These are subclasses of `<global-variable-leaf>` and therefore inherit general characteristics of leaves that represent variables.

ICR leaves representing both atomic and fluid variables are created by the converter for `define variable`. When the compiler encounters a definition of an atomic variable (introduced by the `define atomic-variable construct`), the converter for `define variable` creates an instance of `<atomic-global-variable-leaf>` in the ICR. When the compiler encounters a definition of a fluid variable (introduced by the `define fluid-variable construct`), the converter creates an instance of `<fluid-global-variable-leaf>`.

The operations of reading, writing, and conditionally updating atomic variables and of reading and writing fluid variables are not represented by primitive functions. Instead, they are represented directly in the flow graph. They are implemented by specializing methods on the leaf classes that represent atomic and fluid variables.

## Compiler Support for Primitives

When the compiler constructs the flow graph, it represents a function call as a node in the ICR. Just as the compiler distinguishes atomic and fluid variables by means of specialized leaf classes, so it distinguishes calls to primitive functions of the portability interface by means of a specialized node class.

A function call is an operation on several components: the function object, the arguments, and the destination for returned values. When the compiler encounters a regular Dylan call, which typically appears as a call to a generic function, it represents the call in the ICR as a node of class `<combination>`.

However, the compiler contains a table of the primitive functions in the portability interface. Before creating an ICR node to represent a function call, the compiler looks up the function being called in the table of primitives. If the function appears in the table, the compiler creates an ICR node of class `<primitive-combination>`.

When the back end traverses the flow graph, methods specialized on the node class `<primitive-combination>` emit calls to primitive functions.

## 6.8.3 Support for Dylan Language Features

### Interfacing to Foreign Code

It is intended that threads created by the Dylan library may inter-operate with code written in other languages with no special constraints. Dylan is interfaced with other languages via a Foreign Language Interface (*FLI*), which acts as a barrier between Dylan conventions and the *neutral* conventions of the platform. The FLI is responsible for:

1. mapping between Dylan and foreign data types,
2. converting between Dylan and foreign calling conventions
3. maintaining the Dylan dynamic environment
4. maintaining any support necessary for garbage collection (such as ensuring that all Dylan values can be traced).

The first and second of these require no significant extensions to support multiple threads, since these are inherently computations which have no effect on any thread other than the one performing the computation.

There is a requirement that the dynamic environment for each thread is stored in a thread-local variable. Since the environment is stored in this way, its value is preserved across calls into foreign code, and it will still be valid if the foreign code calls back into Dylan. The techniques described in [MG95] for maintaining the dynamic environment across foreign calls are therefore directly appropriate to a multi-threaded implementation too.

If an object is passed to foreign code with dynamic extent, then it is sufficient to ensure that the object is referenced from the current stack, which the garbage collector will scan conservatively. In a multi-threaded implementation, the garbage collector will scan all the stacks conservatively, so there is no requirement to maintain a thread-global data structure.

If an object is passed with indefinite extent, then it must be recorded in a table. The table may be maintained by the runtime system, by means of suitable primitive functions to add and remove references. There are potentially synchronization problems associated with multiple threads manipulating a global data structure — but the runtime system implementation is free to choose whether to have separate tables for each thread, or whether to have a global table with an associated lock to guard accesses. Either technique is possible — but Harlequin have not yet implemented this feature.

One further consideration is the interaction of the Dylan threads library itself with foreign components:

If foreign code is not designed for multiple threads (for instance, because it uses global data structures, and doesn't synchronize updates), then the code may fail if it is invoked from multiple Dylan threads. However, this problem is not related to the Dylan implementation, since it would fail if called from multiple threads created by any means. The solution is to modify the foreign component to make it thread safe.

If foreign code is designed for use with multiple threads, then it is valid for it to use the synchronization facilities of the Dylan library (by calling back into Dylan, to invoke the Simple Threads Library synchronization functions). Alternatively, it may use its own methods for synchronization, provided that these are not incompatible with the methods provided by the operating system. This is valid whenever it has been possible to implement the runtime system support for threads directly in terms of operating system features, and it is anticipated that this will always be true if the operating system supports threads. Typically, foreign code is expected to make direct use of operating system threads facilities.

However, a problem may arise if a thread is created in foreign code, and the new thread then calls back into Dylan. In this case, the Dylan thread library itself will not be able to find an existing `<thread>` object corresponding to the current thread, and the fluid variables for the current thread will not have been correctly initialized. Worse still, the garbage collector may not have enough information to locate the roots of the thread. Harlequin have not yet allowed for this in their implementation, but they have an anticipated solution.

It is possible to detect that a thread has never been executing on the Dylan side of the FLI before because it will have an uninitialized (zero) value for its thread-local dynamic environment variable. This can be checked at a call-in in the stub function which implements the FLI. Once such a thread has been detected, appropriate initialization steps can be taken. A function in the runtime system can be called to register the stack of the thread for root tracing; the dynamic environment can be set to point to a suitable value on the stack; finally a new Dylan `<thread>` object can be allocated and initialized with `primitive-initialize-current-thread` (as for the first thread).

## Finalization

As has been discussed, the Dylan synchronization objects are implemented as wrappers around lower-level operating system structures. The Dylan objects are subject to garbage collection, and their memory will be automatically freed by the garbage collector at an undefined point in the program. But the low-level structures are not Dylan objects and must be explicitly freed when the Dylan container is collected (primitive functions are provided for this purpose). However, the core language of Dylan provides no *finalization* mechanism to invoke cleanup code when objects are reclaimed. Harlequin's implementation of the Simple Threads Library strictly requires this, but it is not yet implemented. It is intended to provide finalization support for Dylan with a new garbage collector which is currently under development.

## 6.9 Runtime System Functions

### 6.9.1 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>, name :: <optional-name>, priority :: <small-integer>, function :: <function>) => ()

Arguments

*thread* A Dylan thread object.

*name* The name of the thread (as a <byte-string>) or #f.

*priority* The priority at which the thread is to run.

*function* The initial function to run after the thread is created.

Description

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>) => ()

Arguments

*thread* A Dylan thread object.

Description

Frees any runtime-allocated memory associated with the thread.

primitive-initialize-current-thread

[Primitive]

Signature

(thread :: <thread>) => ()

Arguments

*thread* A Dylan thread object.

Description

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 :: <small-integer>)

Arguments

*thread* A Dylan thread object.

Values

*error-code* 0 = ok, anything else is an error, corresponding to a multiple join.

Description

The calling thread blocks (if necessary) until the specified thread has terminated.

primitive-thread-join-multiple

[Primitive]

Signature

(thread-vector :: <simple-object-vector>) => (result)

Arguments

*thread-vector* A <simple-object-vector> containing <thread> objects

Values

*result* The <thread> that was joined, if the join was successful; otherwise, a <small-integer> indicating the error.

Description

The calling thread blocks (if necessary) until one of the specified threads has terminated.

primitive-thread-yield

[Primitive]

Signature

() => ()

Description

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

Description

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 :: <optional-name>) => ()

Arguments

*lock* A Dylan `<simple-lock>` object.

*name* The name of the lock (as a `<byte-string>`) or `#f`.

Description

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>) => ()

Arguments

*lock* A Dylan `<simple-lock>` object.

Description

Frees any runtime-allocated memory associated with the lock.

primitive-wait-for-simple-lock

[Primitive]

Signature

(lock :: <portable-container>) => (error-code :: <small-integer>)

Arguments

*lock* A Dylan `<simple-lock>` object.

Values

*error-code* 0 = ok

Description

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 :: <small-integer>) => (error-code :: <small-integer>)

### Arguments

*lock* A Dylan `<simple-lock>` object.

*milliseconds* Timeout period in milliseconds

### Values

*error-code* 0 = ok, 1 = timeout expired

### Description

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 :: <small-integer>)

### Arguments

*lock* A Dylan `<simple-lock>` object.

### Values

*error-code* 0 = ok, 2 = not locked

### Description

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 :: <small-integer>)

### Arguments

*lock* A Dylan `<simple-lock>` object.

### Values

*owned* 0= not owned, 1 = owned

### Description

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 :: <optional-name>) => ()

### Arguments

*lock* A Dylan `<recursive-lock>` object.

*name* The name of the lock (as a <byte-string>) or #f.

### Description

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>) => ()

### Arguments

*lock* A Dylan '<recursive-lock>' object.

### Description

Frees any runtime-allocated memory associated with the lock.

primitive-wait-for-recursive-lock

[Primitive]

### Signature

(lock :: <portable-container>) => (error-code :: <small-integer>)

### Arguments

*lock* A Dylan <recursive-lock> object.

### Values

*error-code* 0 = ok

### Description

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 :: <small-integer>) => (error-code :: <small-integer>)

### Arguments

*lock* A Dylan <recursive-lock> object.

*millisecs* Timeout period in milliseconds

### Values

*error-code* 0 = ok, 1 = timeout expired

### Description

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 :: <small-integer>)

Arguments

*lock* A Dylan “<recursive-lock>” object.

Values

*error-code* 0 = ok, 2 = not locked

Description

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 :: <small-integer>)

Arguments

*lock* A Dylan <recursive-lock> object.

Values

*owned* 0= not owned, 1 = owned

Description

Returns 1 if the specified lock is locked and owned by the calling thread.

## Semaphores

primitive-make-semaphore

[Primitive]

Signature

(lock :: <portable-container>, name :: <optional-name>,

initial :: <small-integer>, max :: <small-integer>) => ()

Arguments

*lock* A Dylan <semaphore> object.

*name* The name of the lock (as a <byte-string>) or #f.

*initial* The initial value for the semaphore count

Description

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>) => ()

Arguments

*lock* A Dylan <semaphore> object.

Description

Frees any runtime-allocated memory associated with the semaphore.

primitive-wait-for-semaphore

[Primitive]

Signature

(lock :: <portable-container>) => (error-code :: <small-integer>)

Arguments

*lock* A Dylan <semaphore> object.

Values

*error-code* 0 = ok

Description

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 :: <small-integer>)

=> (error-code :: <small-integer>)

Arguments

*lock* A Dylan <semaphore> object.

*millisecs* Timeout period in milliseconds

Values

*error-code* 0 = ok, 1 = timeout expired

Description

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 :: <small-integer>)

Arguments

*lock* A Dylan <semaphore> object.

Values

*error-code* 0 = ok, 3 = count exceeded

#### Description

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 :: <optional-name>) => ()

Arguments

*notification* A Dylan <notification> object.

*name* The name of the notification (as a <byte-string>) or *#f*.

Description

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>) => ()

Arguments

*notification* A Dylan <notification> object.

Description

Frees any runtime-allocated memory associated with the notification.

primitive-wait-for-notification

[Primitive]

Signature

(notification :: <portable-container>, lock :: <portable-container>)

=> (error-code :: <small-integer>)

Arguments

*notification* A Dylan <notification> object.

*lock* A Dylan <simple-lock> object.

Values

*error-code* 0 = ok, 2 = not locked, 3 = other error

Description

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>,  
milliseconds :: <small-integer>) => (error-code :: <small-integer>)

Arguments

*notification* A Dylan <notification> object.

*lock* A Dylan <simple-lock> object.

*milliseconds* Timeout period in milliseconds

Values

*error-code* 0 = ok, 1 = timeout, 2 = not locked, 3 = other error

Description

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 :: <small-integer>)

Arguments

*notification* A Dylan <notification> object.

*lock* A Dylan <simple-lock> object.

Values

*error-code* 0 = ok, 2 = not locked

Description

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 :: <small-integer>)

### Arguments

*notification* A Dylan <notification> object.

*lock* A Dylan <simple-lock> object.

### Values

*error-code* 0 = ok, 2 = not locked

### Description

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* :: <small-integer>) => ()

### Arguments

*millisecs* Time interval in milliseconds

### Description

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*)

### Arguments

*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.

### Description

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.

## 6.9.2 Simple Runtime Primitives

### D `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.

### D `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 `primitive_fill_E_` (D *storage[]*, int *size*, D *value*)

(The odd name is a result of name mangling from `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 *size*.

### D `primitive_replace_E_` (D *dst[]*, D *src[]*, int *size*)

(See `primitive_fill_E_()` re. name). This copies from the source vector into the destination vector as many values as are specified in the *size* parameter.

### D `primitive_replace_vector_E_` (SOV\* *dest*, SOV\* *source*)

This is related to `primitive_replace_E_()`, except that the two arguments are guaranteed to be simple object vectors, and they are self-sizing. It takes two parameters, 'dest', and 'source', and the data from 'source' is copied into 'dest'. 'Dest' is returned.

### D `primitive_allocate_vector` (int *size*)

This is related to `primitive_allocate()`, except that it takes a '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 'size' field shows how big it is.

### D `primitive_copy_vector` (D *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 `primitive_initialize_vector_from_buffer` (SOV \* *vector*, int *size*, D\* *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 '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 `primitive_make_string` (char \* *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 `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 **primitive\_nlx** (Bind\_exit\_frame\* *target*, SOV\* *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 `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 **primitive\_inlined\_nlx** (Bind\_exit\_frame\* *target*, D *first\_argument*)

This is similar to `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\* **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\* **primitive\_make\_environment** (int *size*, ...)

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.

## 6.9.3 Entry Point Functions

D **xep\_0** (FN\* *function*, int *argument\_count*)

D **xep\_1** (FN\* *function*, int *argument\_count*)

D **xep\_2** (FN\* *function*, int *argument\_count*)

D **xep\_3** (FN\* *function*, int *argument\_count*)

D **xep\_4** (FN\* *function*, int *argument\_count*)

D **xep\_5** (FN\* *function*, int *argument\_count*)

D **xep\_6** (FN\* *function*, int *argument\_count*)

D **xep\_7** (FN\* *function*, int *argument\_count*)

D **xep\_8** (FN\* *function*, int *argument\_count*)

D **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 `xep0`; if it takes a single required parameter it will be `xep1`, and so on. There are values available for `xep0` to `xep9`. For more than nine required parameters, the `xep()` function is used.

**xep** (FN\* *function*, int *argument\_count*, ...)

If the function takes more than nine required parameters, then the function will simply be called `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 **optional\_xep** (FN\* *function*, int *argument\_count*, ...)

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.

D **gf\_xep\_0** (FN\* *function*, int *argument\_count*)

D **gf\_xep\_1** (FN\* *function*, int *argument\_count*)

D **gf\_xep\_2** (FN\* *function*, int *argument\_count*)

D **gf\_xep\_3** (FN\* *function*, int *argument\_count*)

D **gf\_xep\_4** (FN\* *function*, int *argument\_count*)

D **gf\_xep\_5** (FN\* *function*, int *argument\_count*)

D **gf\_xep\_6** (FN\* *function*, int *argument\_count*)

D **gf\_xep\_7** (FN\* *function*, int *argument\_count*)

D **gf\_xep\_8** (FN\* *function*, int *argument\_count*)

D **gf\_xep\_9** (FN\* *function*, int *argument\_count*)

These primitives are similar to `xep_0()` through `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.

D **gf\_xep** (FN\* *function*, int *argument\_count*, ...)

This primitive is similar to `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.

D **gf\_optional\_xep** (FN\* *function*, int *argument\_count*, ...)

This is used for all generic functions which take optional arguments. This function calls the internal entry point.

D **primitive\_basic\_iep\_apply** (FN\* *f*, int *argument\_count*, D *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 it 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 `primitive_basic_iep_apply()`, it has to make sure that any necessary 'function' or 'mlist' parameters have been set up.

D **primitive\_iep\_apply** (FN\* *f*, int *argument\_count*, D *a*[])

This is closely related to `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.

D **primitive\_xep\_apply** (FN\* *f*, int *argument\_count*, D *a*[])

This is a more usual usage of `apply`, i.e., the standard Dylan calling convention being invoked by `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.

## 6.10 Compiler Primitives

### 6.10.1 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>

## 6.10.2 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>

Description

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>

Description

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>

rimitive-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\*>

### 6.10.3 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 <i>type</i> in Primitive Name <i>primitive-type-operator</i>
Small Integer	<raw-small-integer>	<i>small-integer</i>
Big Integer	<raw-big-integer>	<i>big-integer</i>
Machine Integer	<raw-machine-integer>	<i>machine-integer</i>
Unsigned Machine Integer	<raw-unsigned-machine-integer>	<i>unsigned-machine-integer</i>

## 6.10.4 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>

Description

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>

Description

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>

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>

### 6.10.5 Accessor Primitives

primitive-element

[Primitive]

Signature

(array :: <object>, index :: <raw-small-integer>) => <object>

Description

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>

#### Description

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>

#### Description

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>

#### Description

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]

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]

Signature

(address :: <raw-pointer>) => <raw-unsigned-64-bit-integer>

primitive-unsigned-64-bit-integer-at-setter

[Primitive]

Signature

**(new-value :: <raw-unsigned-64-bit-integer>, address :: <raw-pointer>) => <raw-unsigned-64-bit-integer>**

primitive-ieee-single-float-at

[Primitive]

Signature

(address :: <raw-pointer>) => <raw-ieee-single-float>

primitive-ieee-single-float-at-setter

[Primitive]

Signature

(new-value :: <raw-ieee-single-float>, address :: <raw-pointer>) => <raw-ieee-single-float>

primitive-ieee-double-float-at

[Primitive]

Signature

(address :: <raw-pointer>) => <raw-ieee-double-float>

primitive-ieee-double-float-at-setter

[Primitive]

Signature

**(new-value :: <raw-ieee-double-float>, address :: <raw-pointer>) => <raw-ieee-double-float>**

primitive-ieee-extended-float-at

[Primitive]

Signature

(address :: <raw-pointer>) => <raw-ieee-extended-float>

primitive-ieee-extended-float-at-setter

[Primitive]

Signature

(new-value :: <raw-ieee-extended-float>, address :: <raw-pointer>) => <raw-ieee-extended-float>

# LIBRARY DOCUMENTATION

We are working on a tool to automatically generate skeletal documentation from source code, but until then, we are documenting the Open Dylan libraries manually using [Sphinx](#) to build the HTML pages. Sphinx uses reStructuredText markup with some extensions of its own, and we have created additional extensions to document Dylan language entities.

The documentation — a number of RST files — is in the “documentation” directory in the “opendylan” repository. Consult the Sphinx web-site for details about reStructuredText markup and Sphinx extensions to it, and see the “dylandomain/reference.rst” file in the “sphinx-extensions” repository for details about the Dylan language extensions. (You may use the `rst2html` tool to generate an HTML page from an `.rst` file.)

## 7.1 Example documentation

Here is an example of documentation to get a feel for how it works. Use the “Show Source” link at the bottom of the page to see it in RST markup form.

### 7.1.1 Skip Lists

A skip-list is a data type equivalent to a balanced (binary) tree.

#### **skip-list Library**

The skip list library may be found in the “[skip-list](#)” repository.

#### **skip-list Module**

The skip-list module exports a number of symbols. Two new symbols are of interest:

- `<skip-list>`
- `element-sequence`

The skip-list module also adds to several generic methods. One of the new methods is:

- `element`

#### **<skip-list> Open Primary Class**

A skip-list is a data type equivalent to a balanced (binary) tree. All keys must be comparable by some kind of ordering function, e.g., `<`.

**Superclasses** `<stretchy-collection>`, `<mutable-explicit-key-collection>`

#### **Init-Keywords**

- **key-test** – The collection’s key-test function; should return `#t` if two keys should be considered equal. Defaults to `==`.

- **key-order** – A function that accepts two keys and returns `#t` if the first key sorts before the second. Defaults to `<`.
- **size** – Preallocates enough memory to hold this number of objects. Optional.
- **capacity** – Sets the maximum capacity of the skip list. Optional.
- **probability** – The probability to create a new level of the list. Equivalent to the fan-out of a tree. Defaults to 0.25.
- **max-level** – The list will not grow beyond this number of levels. Defaults to a value based on the `size` and `capacity` keywords.
- **level** – The list starts with this number of levels. Defaults to a value based on the `size` and `capacity` keywords.

In general, a skip list operates like a stretchy mutable key collection. But a skip list can also act as an *ordered* stretchy mutable key collection where the iteration order is the key order. To take advantage of this, the library defines `forward-by-key-iteration-protocol`, `element-sequence`, and `element-sequence-setter`.

### `element-sequence` Generic function

#### Parameters

- **list** – A skip list.

#### Values

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

One of the useful features of skip lists is that they can be ordered. However, most of the useful operations that can be performed on ordered collections, such as `sort`, are only defined for sequences. To solve this problem, I add `element-sequence` and `element-sequence-setter`. The client may call the former to obtain a sequence, operate on it, and call the latter to fix the results in the skip list. The setter ensures that no elements have been added or removed from the skip list, only reordered.

### `element (<skip-list>)` Method

A specialization of `element`.

#### Parameters

- **collection** – An instance of `<skip-list>`.
- **key** – The key of an element. An instance of `<object>`.
- **default (#key)** – A value to return if the element is not found. If omitted and element not found, signals an error.

#### Values

- **object** – The element associated with the key.

# GLOSSARY

**canonical sources** only meaningful in reference to a particular compiler database or compilation context. The set of sources from which the information in the database was derived.

**compilation context** the in-memory representation of an open compiler database. Consists of a connection to a disk database, a reference to the owning project, and a collection of caches for pre-loaded/pre-computed information.

**compiler database** a file or set of files containing information derived from compiling a project. A project can have multiple compiler databases, corresponding to different target machines, compiler settings, or even different versions of the project sources. The project manager is responsible for managing the collection of project databases and telling the compilation system which database to use.

**component** a native DLL or EXE file.

**DFMC** the Dylan Flow Machine Compiler, the intermediate representation on which type inference and optimization (dispatch, inline, dynamic extent, common subexpression elimination, constant folding, dead code removal) is done. Source of data and control flow elements is in *dfmc/flow-graph*.

**execution context** the compiler-derived information about a process, such as the namespaces of known runtime components, installed definitions, etc. Initialized from the compilation contexts of the preloaded runtime components and subsequently updated by interactive execution.

**HARP** Harlequin Abstract RISC Machine, the native back-end of Open Dylan. On this representation register allocation etc is done: input DFM, output: assembly. Source is in *harp* subdirectory.

**interactive execution** a mechanism for exploratory programming by which the user can execute Dylan forms in an existing process. May allow “out of language” operations such as addition of new variables to existing modules, redefinition of classes, constants, overriding sealing restrictions, etc. Forms to be executed can come from a project or from an interactor.

**project** a development environment object representing a Dylan program under development. It corresponds to a Dylan library, but it exists before the compiler is ever invoked. It is used by the compiler only to identify a library in interactions with the project manager.

**project manager** the part of the development environment charged with managing projects. The project manager is a client of the compilation system, i.e. it is the project manager which is expected to invoke many of the functions in this API.

**runtime component** a runtime manager object representing a component loaded into a tethered process.

**interactor** a mechanism by which a user can type in source records for interactive execution without modifying project sources.

**runtime manager** the part of the development environment charged with controlling the runtime. The compilation system is a client of the runtime manager, i.e. the compilation system will invoke runtime manager functions as needed to effect interactive execution.

**source record** smallest unit of source suitable for compilation. Consists of a stream of characters and a module name. Contains complete top-level forms (i.e. top-level forms may not be split across source records).

**tether** a runtime manager object representing a debuggable process on the runtime. Sometimes referred to as an “access-path”, but I’m staying away from that term because it seems to be used differently in different documents.

# INDICES AND TABLES

- *genindex*
- *search*



# API INDEX

## E

`element` (<skip-list>) (*method*), 72  
`element-sequence` (*generic function*), 72

## S

<skip-list> (*class*), 71  
`skip-list` (*library*), 71  
`skip-list:skip-list` (*module*), 71



# INDEX

## Symbols

<skip-list>, 71

## E

element

    element(<skip-list>), 72

element-sequence, 72

## G

gf\_optional\_xep (C function), 56

gf\_xep (C function), 56

gf\_xep\_0 (C function), 56

gf\_xep\_1 (C function), 56

gf\_xep\_2 (C function), 56

gf\_xep\_3 (C function), 56

gf\_xep\_4 (C function), 56

gf\_xep\_5 (C function), 56

gf\_xep\_6 (C function), 56

gf\_xep\_7 (C function), 56

gf\_xep\_8 (C function), 56

gf\_xep\_9 (C function), 56

## O

optional\_xep (C function), 56

## P

primitive\_allocate (C function), 54

primitive\_allocate\_vector (C function), 54

primitive\_basic\_iep\_apply (C function), 56

primitive\_byte\_allocate (C function), 54

primitive\_continue\_unwind (C function), 54

primitive\_copy\_vector (C function), 54

primitive\_fill\_E\_ (C function), 54

primitive\_iep\_apply (C function), 56

primitive\_initialize\_vector\_from\_buffer (C function), 54

primitive\_inlined\_nlx (C function), 55

primitive\_make\_box (C function), 55

primitive\_make\_environment (C function), 55

primitive\_make\_string (C function), 54

primitive\_nlx (C function), 55

primitive\_replace\_E\_ (C function), 54

primitive\_replace\_vector\_E\_ (C function), 54

primitive\_xep\_apply (C function), 57

## S

skip-list, 71

## X

xep (C function), 56

xep\_0 (C function), 55

xep\_1 (C function), 55

xep\_2 (C function), 55

xep\_3 (C function), 55

xep\_4 (C function), 55

xep\_5 (C function), 55

xep\_6 (C function), 55

xep\_7 (C function), 55

xep\_8 (C function), 55

xep\_9 (C function), 55