An Introduction to Dylan

Release 1.0

Dylan Hackers

November 22, 2016

CONTENTS

1 Why Dylan? 3
1.1 Dynamic vs. Static Languages . . . 3
1.2 Functional Languages . . . 3
1.3 Algebraic Infix Syntax . . . 4
1.4 Object Orientation . . . 4
1.5 Garbage Collection . . . 4
1.6 Why Not Dylan? . . . 5

2 Expressions & Variables 7
2.1 Naming Conventions . . . 7
2.2 True and False . . . 7
2.3 The Nature of Variables . . . 7
2.4 Assignment, Equality and Identity . . . 8
2.5 Parallel Values . . . 9
2.6 Type Declarations . . . 9
2.7 Module Variables and Constants . . . 9

3 Methods & Generic Functions 11
3.1 Parameters & Parameter Lists . . . 11
3.2 Return Values . . . 12
3.3 Bare Methods . . . 12
3.4 Local Methods . . . 13
3.5 Generic Functions . . . 13
3.6 Keyword Arguments . . . 14
3.7 Rest Arguments . . . 15
3.8 Parameter Lists and Generic Functions . . . 15

4 Objects 19
4.1 Built-In Classes . . . 19
4.2 Slots . . . 20
4.3 Getters and Setters . . . 20
4.4 Generic functions and Objects . . . 21
4.5 Initializers . . . 22
4.6 Abstract Classes and Overriding Make . . . 22

5 Multiple Dispatch 25
5.1 Dispatching on Specific Objects . . . 26

6 Modules & Libraries 27
6.1 Simple Modules . . . 27
6.2 Import Options . . . 27

6.3 Libraries 28
6.4 Sealing 28

7 Conditions **31**
7.1 Signaling 31
7.2 Handlers 31
7.3 Recovery 32
7.4 Blocks 33

8 Copyright **35**

This document introduces the Dylan programming language. Dylan is an object-oriented dynamic language designed for efficient compilation. It uses an algebraic infix syntax similar to Pascal or C, but supports an object model not unlike the Common Lisp Object System (CLOS).

This tutorial is written primarily for those with solid programming experience in C++ or another object-oriented static language. It provides a gentler introduction to Dylan than does the DRM, although it refers to the latter book frequently.

CHAPTER

ONE

WHY DYLAN?

What earthly reason could there be for learning yet *another* computer language? And why should that language be Dylan?

Dylan has an interesting combination of features. It is a dynamic language, but is designed to perform nearly as well as a static language. It is a *functional* language – like Scheme or TCL – but uses an algebraic infix syntax similar to C's. Dylan is object-oriented from the ground up, supports multiple inheritance and exceptions, implements *multiple dispatch*, and *collects garbage*.

1.1 Dynamic vs. Static Languages

Static languages need to know the type of every variable at compile time. Examples of static languages include C++, Java, and Go. Code written in static languages typically compiles efficiently, and strong type-checking at compile-time reduces the risk of errors.

Dynamic languages allow the programmer to create variables without explicitly specifying the type of information they contain. This simplifies prototyping and cleans up certain kinds of object oriented code. Typical dynamic languages include Common Lisp, Javascript, and Python.

Dylan provides a good balance between the advantages of static and dynamic languages. The programmer may choose to specify or omit type declarations as desired. Code using explicit variable types can be compiled to very efficient code, and type mismatch errors can be caught at compile time. Code omitting those type declarations gains the flexibility of a dynamic language.

1.2 Functional Languages

Functional languages, such as Common Lisp, Scheme and to a large extent TCL, view an entire program as one large function to be evaluated. Expressions, statements and even control structures all return values, which may in turn be used as arguments elsewhere.

Dylan is a functional language, permitting programmers to write functions like the following:

```
define method shoe-size (person :: <string>)
  if (person = "Larry")
    14
  else
    11
  end if
end method;
```

The function `shoe-size` has one argument, a string, and an untyped return value. (If this function didn't link against external code, the compiler could easily infer the return type.) If `person` equals `"Larry"`, then the `if` statement

returns 14, otherwise it returns 11. Since no other statements follow the `if`, its return value is used as the return value of the entire function.

The same function could also have been written as follows, in a more *imperative* idiom:

```
define method shoe-size (person :: <string>)
  let the-size = 11;
  if (person = "Joe")
    the-size := 14;
  end if;
  the-size
end method;
```

1.3 Algebraic Infix Syntax

Languages based on LISP typically use a notation called *fully-parenthesized prefix syntax* (also known as s-expressions). This consists of nested parentheses, as seen in the following Scheme version of the `shoe-size` function:

```
(define (shoe-size person)
  (if (equal? person "Joe")
    14
    11))
```

This has a certain elegance, but takes some time to learn to read. Dylan, as shown in *the previous section*, uses a syntax similar to those of C and Pascal.

1.4 Object Orientation

Unlike many other object-oriented languages, Dylan uses objects for every data value. Integers and strings are objects, as are functions and classes themselves.

Dylan's design makes this reasonably efficient. Compile-time analysis and explicit *type declarations* allow the compiler to optimize away most of the overhead. Other language features permit the programmer to mark certain classes as *sealed*, that is, ineligible for further subclassing. This allows for further compile-time optimizations.

Dylan's object model, detailed in the following sections of this tutorial, differs from that of C++ in several important respects. Multiple inheritance may be used freely, without concern for *object slicing*, erroneous down-casting or a whole host of other gotchas familiar to C++ programmers. Methods are separate from class declarations, allowing a programmer to write new polymorphic functions without editing the relevant base class. Methods may also dispatch polymorphically on more than one parameter, a powerful technique known as *multiple dispatch*. All of these features will be explained in greater detail later on.

1.5 Garbage Collection

Languages with *garbage collection* have no need of a `free` or `delete` operator, because unused heap memory gets reclaimed automatically by the language runtime. This reduces the complexity of source code, eliminates the need of keeping reference counts for shared objects, and prevents most memory allocation bugs and a major source of memory leaks.

1.6 Why Not Dylan?

Dylan's greatest weaknesses are its lack of a battle-hardened compiler and IDE, and a large user base (and hence a large set of libraries). However, the compiler and IDE themselves are written in Dylan so there *are* several hundred thousand lines of Dylan code. You probably want to consider very carefully before using Open Dylan for mission critical code.

CHAPTER

TWO

EXPRESSIONS & VARIABLES

Dylan identifiers may contain a greater variety of characters than those of C++ or Java. Specifically, variable names may contain all alphanumeric characters, plus the symbols `! & * < = > | ^ $ % @ _ - + ~ ? /`. Identifiers may not begin with the symbols `- + ~ ? /`, although identifiers may begin with numbers, provided they contain at least two alphabetic characters in a row. Variable names are not case sensitive.

This means that `(a - b)` subtracts one variable from another, whereas `(a-b)` simply returns the value of the hyphenated variable named `a-b`. Because of this, infix operators, such as addition, subtraction and equality, must be surrounded by whitespace.

As in C++, Dylan infix operators may also be referred to as functions. In C++, `(a + b)` could also be written as `operator+(a, b)`. In Dylan, the same expression could be written `\+(a, b)`. In both languages, programmers can use this flexibility to define operators for custom numeric classes.

2.1 Naming Conventions

Dylan uses the extra characters permitted in variable names to support a number of standard naming conventions, as shown in this table:

`<string>`	a class
`insert!`	mutative function (modifies argument destructively)
`empty?`	predicate function (tests one or more arguments and returns either true or false)
`write-line`	a two word name
`$name`	constant
`*name*`	module-level variable

2.2 True and False

Dylan represents true as `#t` and false as `#f`. When evaluated in a Boolean context, all values other than `#f` are considered true. Thus, the number zero – and other common "false" values – evaluate as true in Dylan.

2.3 The Nature of Variables

Dylan variables differ from those found in C and Pascal. Instead of *holding* their values, Dylan variables *refer* to them. Conceptually, they resemble a cross between pointers and C++ references. Like references, Dylan variables may be evaluated without any indirection. Like pointers, they may be set to point to new objects whenever the programmer desires.

Furthermore, there's only one of any given numeric value in a Dylan program, at least from the programmer's point of view. All variables which refer to the integer 2 – or, in Dylan-speak, are *bound* to the integer 2 – point to the exact same thing.

```
let x = 2;  // creates x and binds it to 2
x := 3;    // rebinds x to the value 3
let y = x;  // creates y, and binds it to whatever x is bound to
```

If two variables are bound to one object with internal structure, the results may surprise C and Pascal programmers.

```
let car1 = make(<car>);  // bind car1 to a new car object
car1.odometer := 10000;       // set odometer
let car2 = car1;              // bind new name
car2.odometer := 0;           // reset odometer
car1.odometer;                // evaluates to 0
```

As long as one or more variables refer to an object, it continues to exist. However, as soon as the last reference either goes out of scope or gets rebound, the object becomes *garbage*. Since there's no way that the program could ever refer to the object again, the *garbage collector* feels free to reuse the memory which once held it.

Note that Dylan variables *must* be bound to a particular value when they are declared. In the name of type safety and implementation efficiency, every variable must refer to some well-defined object.

2.4 Assignment, Equality and Identity

Dylan uses all three of the "equals" operators found in C and Pascal, albeit in a different fashion. The assignment operator, `:=`, rebinds Dylan variable names to new values. The equality operator, =, tests for equality in Dylan and also appears in some language constructs such as `let`. (Two Dylan objects are equal, generally, if they belong to the same class and have equal substructure.)

The C++ equality operator, ==, acts as the *identity* operator in Dylan. Two variables are *identical* if and only if they are bound to the exact same object. For example, the following three expressions mean roughly the same thing:

```
(a == b)   // in Dylan or Java
(&a == &b) // in C or C++
```

The following piece of source code demonstrates all three operators in actual use.

```
let car1 = make(<car>);
let car2 = make(<car>);
let car3 = car2;

car2 = car3;  // #t
car1 = car2;  // ??? (see below)
car2 == car3;  // #t
car1 == car2;  // #f

car2 := car1;  // rebind
car1 == car2;  // #t

let x = 2;
let y = 2;

x = y;  // #t
x == y;  // #t (there is only one 2!)
```

Two of the examples merit further explanation. First, we don't know if make creates each car with the same serial number, driver and other information as previous cars, or whether there is a method defined on `\=(<car>, <car>)`

that compares cars slot-by-slot.

Second, `x == y` because every variable bound to a given number refers to the exact same instance of that number, at least from the programmer's perspective. (The compiler will normally do something more useful and efficient when generating the actual machine code.) Strings behave in a fashion different from numbers – instances of strings are stored separately, and two equal strings are not necessarily the same string.

2.5 Parallel Values

It's possible to bind more than one variable at a time in Dylan. For example, a single `let` statement could bind `x` to 2, `y` to 3 and `z` to 4.

```
let (x, y, z) = values(2, 3, 4);
```

In Perl, the equivalent statement would assign a vector of values to a vector of variables. In Dylan, no actual vectors or lists are used. All three values are assigned directly, using some implementation-dependent mechanism.

2.6 Type Declarations

Dylan variables may have explicit types. This allows the compiler to generate better code and to catch type-mismatch errors at compile time. To take advantage of this feature, use the `::` operator:

```
let x :: <integer> = 2;
let vehicle :: <vehicle> = make(<car>);
let y :: <number> = 3;  // any numeric class
let z :: <integer> = vehicle;  // error!
```

As seen in the example, a variable may be bound to values of its declared type or to values of subclasses of its declared type. Type mismatch errors should be caught at compile time. In general, the compiler may infer the types of variables at when generating machine code. If a local variable never gets rebound to anything other than an integer, for example, the compiler can rely on this fact to optimize the resulting code.

2.7 Module Variables and Constants

Dylan supports *module-level* variables, which serve roughly the same purpose as C's global variables. Although the `let` function may only be used within *methods* (Dylan-speak for regular functions), the forms `define variable` and `define constant` may be used at module top level.

```
define variable *x* :: <integer> = 3;
define variable *y* = 4;
define constant $hi = "Hi!";
```

Note that there's not much point in declaring types for constants. Any remotely decent compiler will be able to figure that information out on its own.

CHAPTER

THREE

METHODS & GENERIC FUNCTIONS

Dylan *methods* correspond roughly to the functions found in C++. They take zero or more named parameters, but also return zero or more named return values. A minimal Dylan method might look like the following:

```
define method hello-world ()
  format-out("Hello, world!");
end;
```

This method has no parameters and an unspecified return value. It could return any number of values of any type. In order to make the above code more clear, the function could be rewritten as follows:

```
define method hello-world () => ()
  format-out("Hello, world!");
end method;
```

There have been two changes. The function now officially returns no values whatsoever. Also note that `end` has been replaced by `end method` which could in turn be rewritten as `end method hello-world`. In general, Dylan permits all the obvious combinations of keywords and labels to follow an end statement.

3.1 Parameters & Parameter Lists

Dylan methods declare parameters in fashion similar to that of conventional languages, except for the fact that parameters may optionally be untyped. Both of the following methods are legal:

```
define method foo (x :: <integer>, y) end;
define method bar (m, s :: <string>) end;
```

Both `foo` and `bar` have one typed and one untyped parameter, but neither has a well-defined return value (or actually does anything). As in C, each typed parameter must have its own type declaration; there's no syntax for saying "the last three parameters are all integers".

Functions with variable numbers of parameters include the `#rest` keyword in their parameter lists. Thus, the declaration for C's `printf` function would appear something like the following in Dylan:

```
define method printf (format-string :: <string>, #rest arguments) => ()
  // Print the format string, extracting one at a time from "arguments".
  // Note that Dylan actually allows us to verify the types of variables,
  // preventing those nasty printf errors, such as using %d instead of %ld.
  // ...
end method printf;
```

Note that Dylan makes no provision for passing variables by reference in the Pascal sense, or for passing pointers to variables. parameter names are simply bound to whatever values are passed, and may be rebound like regular variables.

This means that there's no way to write a `swap` function in Dylan. (It may be done using macros). However, the following function works just fine, because it modifies the *internal state* of another object:

```
define method sell (car :: <car>, new-owner :: <string>) => ()
  if (credit-check(new-owner))
    car.owner := new-owner;
  else
    error("Bad credit!");
  end;
end;
```

If this sounds unclear, reread the chapter on *variables and expressions*.

3.2 Return Values

Because Dylan methods can't have "output" parameters, they're allowed considerably more flexibility when it comes to return values. Methods may return more than one value. As with parameters, these values may be typed or untyped. All return values *must* be named.

A Dylan method – or any other control construct – returns the value of the last expression in its body.

```
define method foo () => (sample :: <string>)
  "Sample string."    // return string
end;

define method bar () => (my-untyped-value)
  if (weekend-day?(today()))
    "Let's party!"  // return string
  else
    make(<excuse>)  // return object
  end if
end method;

define method moby () => (sample :: <string>, my-untyped-value)
  values(foo(), bar())    // return both!
end;

define method baz () => ()
  let (x,y) = moby();  // assign both
end;
```

3.3 Bare Methods

Nameless methods may be declared inline. Such *bare methods* are typically used as parameters to other methods. For example, the following code fragment squares each element of a list using the built in `map` function and a bare method:

```
define method square-list (numbers :: <list>) => (out :: <list>)
  map(method(x) x * x end, numbers);
end;
```

The `map` function takes each element of the list `numbers` and applies the anonymous method. It then builds a new list using the resulting values and returns it. The method `square-list` might be invoked as follows:

```
square-list(#(1, 2, 3, 4));
=> #(1, 4, 9, 16)
```

3.4 Local Methods

Local methods resemble bare methods but have names. They are declared within other methods, often as private utility routines.

```
define method sum-squares (in :: <list>) => (sum-of-element-squares :: <integer>)
  local method square (x)
          x * x
        end,
        method sum (list :: <list>)
          reduce1(\+, list)
        end;
  sum(map(square, in))
end;
```

Local methods can outlive the invocation of the function which created them. parameters of the parent function remain bound in a local method, allowing some interesting techniques:

```
define method build-put (string :: <string>) => (res :: <function>)
  local method string-putter()
          format-out(string);
        end;
  string-putter   // return local method
end;

define method print-hello () => ()
  let f = build-put("Hello!");
  f()  // print "Hello!"
end;
```

Local functions which contain references to local variables that are outside of the local function's own scope are known as *closures*. In the above example, `string-putter` "closes over" (or captures the binding of) the variable named `string`.

3.5 Generic Functions

A *generic function* represents zero or more similar methods. Every method created by means of `define method` is automatically *contained* within the generic function of the same name. For example, a programmer could define three methods named `display`, each of which acted on a different data type:

```
define method display (i :: <integer>)
  do-display-integer(i);
end;

define method display (s :: <string>)
  do-display-string(s);
end;

define method display (f :: <float>)
  do-display-float(f);
end;
```

When a program calls `display`, Dylan examines all three methods. Depending on the type of the argument to `display`, Dylan invokes one of the above methods. If no methods match the actual parameters, an error occurs.

In C++, this process occurs only at compile time. (It's called operator overloading.) In Dylan, calls to `display` may

be resolved either at compile time or while the program is actually executing. This makes it possible to define methods like:

```
define method display (c :: <collection>)
  for (item in c)
    display(item);  // runtime dispatch
  end;
end;
```

This method extracts objects of unknown type from a collection, and attempts to invoke the generic function `display` on each of them. Since there's no way for the compiler to know what type of objects the collection actually contains, it must generate code to identify and invoke the proper method at runtime. If no applicable method can be found, the Dylan runtime environment throws an exception.

Generic functions may also be declared explicitly, allowing the programmer to exercise control over what sort of methods get added. For example, the following declaration limits all `display` methods to single parameter and no return values:

```
define generic display (thing :: <object>) => ()
```

Generic functions are explained in greater detail in the chapter on *multiple dispatch*.

3.6 Keyword Arguments

Functions may accept *keyword arguments*, extra parameters which are identified by a label rather than by their position in the argument list. Keyword arguments are often used in a fashion similar to *default parameter values* in C++, and they are always optional.

The following hypothetical method might print records to an output device:

```
define method print-records
    (records :: <collection>, #key init-codes = "", lines-per-page = 66)
 => ()
  send-init-codes(init-codes)
  // ...print the records
end method;
```

The arguments following `#key` are keyword arguments. You could call this method in several ways:

```
print-records(recs);
print-records(recs, lines-per-page: 65);
print-records(recs, lines-per-page: 120, init-codes: "***42\n");
```

The first line calls the method without using any of the keyword arguments. The second line uses one of the keyword arguments and the third uses both. Note that the order of the keyword arguments does not matter.

With all three calls, the `init-codes` and `lines-per-page` variables are available in the body of the method, even though keyword arguments are omitted in two of the calls. When a keyword argument is omitted, it is given the default value specified in the method definition. Therefore, in the first call, the `lines-per-page` variable has the value `66`, and in the first and second calls, the `init-codes` variable has the value `""`.

Programmers have quite a bit of flexibility in specifying keyword arguments.

- The default value specifier (e.g. the `= 66` above) may be omitted, in which case `#f` is used.
- The type of the keyword argument may be specified or omitted, just as with regular arguments.
- The keyword name can be different from the variable name used in the body of the method—a handy tool for preventing name conflicts.

- The default value specifier can be a complex expression, and it can even use earlier parameters.
- The keyword arguments allowed or required by each method can be specified by the generic function. For more on this, see Parameter Lists and Generic Functions below.

The following method uses some of these features:

```
define method subseq
    (seq :: <sequence>, #key start :: <integer> = 0, end: _end :: <integer> = seq.size)
  assert(start <= _end, "start is after end");
  ...
end;
```

Firstly, the `start:` and `end:` keyword arguments are both specialized as `<integer>`. The caller can only supply integers for these parameters. Secondly, the `start:` keyword argument is associated with the `start` variable in the body of the method as usual, but because the Dylan language does not allow a variable named `end`, that keyword argument is instead associated with the `_end` variable. Finally, if the `end:` keyword argument were omitted, the value of the `_end` variable would be the size of the `seq` argument.

3.7 Rest Arguments

An argument list can also include `#rest`, which is used with a variable name:

```
define method format (format-string, #rest format-parameters)
  ...
end method
```

Any extra arguments are passed to the body of the method as a `<sequence>` in the specified variable. For example, if the above method were called like so:

```
format("Today will be %s with a high of %d.", "cloudy", 52);
```

The `format-parameters` variable in the body of the method would have the value `#["cloudy", 52]`.

3.8 Parameter Lists and Generic Functions

A generic function restricts the parameter lists of its methods, but methods can expand on the generic function's parameter list if the generic function allows it. This section describes how that works. It is a little more advanced than rest of this introduction, so you may want to skip this section for now and refer back to it later.

We described the `#key` and `#rest` parameter list tokens above. The `#key` token may also be used by itself, e.g., `define method foo (arg, #key)`. And there is a third parameter list token, `#all-keys`, that indicates that a method permits other keyword arguments than those listed. These features are only useful when working with a generic function and its family of methods. When used together, these tokens must appear in the order `#rest`, `#key`, `#all-keys`.

The table below shows the different kinds of parameter lists that a generic function can have, and what effect each has on the parameter lists of the methods that it contains.

Generic function's parameter list	Methods' parameter lists			
	`#key`	`#key a, b`	`#all-keys`	`#rest`
`(x)`	Forbidden	Forbidden	Forbidden	Forbidden
`(x, #key)`	Required	Allowed	Allowed	Allowed
`(x, #key a, b)`	Required	Required	Allowed	Allowed
`(x, #key, #all-keys)`	Required	Allowed	Automatic	Allowed
`(x, #key a, b, #all-keys)`	Required	Required	Automatic	Allowed
`(x, #rest r)`	Forbidden	Forbidden	Forbidden	Required

Required: Each method must have this element in its parameter list.

Allowed: Each method may have this element in its parameter list, but is not required to.

Forbidden: No method may have this element in its parameter list.

Automatic: Each method effectively has `#all-keys` in its parameter list, even if it is not present.

This table shows the different kinds of parameter lists that a method can have, what the `r` variable contains for each, and which keywords are permitted by each. It is a run-time error to call a method with a keyword argument that it does not permit.

Method's parameter list	Contents of `r`	Permits `a:` and `b:`	Permits other keywords
`(x)`	—	No	No
`(x, #key)`	—	If next method permits	If next method permits
`(x, #key a, b)`	—	Yes	If next method permits
`(x, #key, #all-keys)`	—	Yes	Yes
`(x, #key a, b, #all-keys)`	—	Yes	Yes
`(x, #rest r)`	Extra arguments	No	No
`(x, #rest r, #key)`	Key-words/values	If next method permits	If next method permits
`(x, #rest r, #key a, b)`	Key-words/values	Yes	If next method permits
`(x, #rest r, #key, #all-keys)`	Key-words/values	Yes	Yes
`(x, #rest r, #key a, b, #all-keys)`	Key-words/values	Yes	Yes

Extra arguments: The local variable `r` is set to a `<sequence>` containing all the arguments passed to the method beyond the required arguments (i.e., the sequence will not contain `x`).

Keywords/values: The local variable `r` is set to a `<sequence>` containing all the keywords and values passed to the method. The first element of the sequence is one of the keywords, the second is the corresponding value, the third is another keyword, the fourth is its corresponding value, etc.

If next method permits: The method only permits a keyword if some other applicable method permits it. In other words, it permits all the keywords in the `next-method` chain, effectively inheriting them. This rule is handy when you want to allow for future keywords that make sense within a particular family of related classes but you do not want to be overly permissive.

To illustrate the "next method" rule, say we have the following definitions:

```
define class <shape> (<object>) ... end;
define generic draw (s :: <shape>, #key);

define class <polygon> (<shape>) ... end;
```

```
define class <triangle> (<polygon>) ... end;

define class <ellipse> (<shape>) ... end;
define class <circle> (<ellipse>) ... end;

define method draw (s :: <polygon>, #key sides) ... end;
define method draw (s :: <triangle>, #key) ... end;

define method draw (s :: <ellipse>, #key) ... end;
define method draw (s :: <circle>, #key radius) ... end;
```

The `draw` methods for `<polygon>` and `<triangle>` permit the `sides:` keyword. The method for `<triangle>` permits `sides:` because the method for `<polygon>` objects also applies to `<triangle>` objects and that method permits `sides:`.

However, the `draw` method for `<circle>` only permits the `radius:` keyword, because the `draw` method for `<polygon>` does not apply to `<circle>` objects — the two classes branch off separately from `<shape>`.

Finally, the method for `<ellipse>` does not permit the `radius:` keyword because, while a circle is a kind of ellipse, an ellipse is *not* a kind of circle. `<circle>` does not inherit from `<ellipse>` and the `draw` method for `<circle>` objects does not apply to `<ellipse>` objects.

For more information on keyword arguments, especially their use with *generic functions*, see the DRM.

CHAPTER

FOUR

OBJECTS

The features of Dylan's object system don't map directly onto the features found in C++. Dylan handles access control using *modules*, not `private` declarations within individual classes. Standard Dylan has no destructors, but instead relies upon the garbage collector to recover memory and on block/cleanup to recover lexically scoped resources. Dylan objects don't even have real member functions.

Dylan's object system is at least as powerful as that of C++. Multiple inheritance works smoothly, constructors are rarely needed and there's no such thing as object slicing. Alternate constructs replace the missing C++ features. Quick and dirty classes can be turned into clean classes with little editing of existing code.

Before starting, temporarily set aside any low-level expertise in C++. Dylan differs enough that such knowledge can actually interfere with the initial learning process.

4.1 Built-In Classes

Dylan has a large variety of built-in classes. Several of these represent primitive data types, such as `<integer>` and `<character>`. A few represent actual language-level entities, such as `<class>` and `<function>`. Most of the others implement collection classes, similar to those found in C++'s Standard Template Library. A few of the most important classes are shown here:

Table 4.1: Several Standard Dylan Classes

Primitive Types	Collections
`<character>`	`<string>`
`<integer>`	`<list>`
`<boolean>`	`<vector>`
`<single-float>`	`<table>`
`<double-float>`	`<pair>`
`<symbol>`	`<deque>`

The built-in collection classes include a number of common data structures. Arrays, tables, vectors, ranges and deques should be provided by all Dylan implementations. The language specification also standardizes strings and byte-strings.

Not all the built-in classes may be subclassed. This allows the compiler to heavily optimize code dealing with basic numeric types and certain common collections. The programmer may also mark classes as *sealed*, restricting how and where they may be subclassed. See Sealing for details.

4.2 Slots

Objects have *slots*, which resemble the data members in C++ or fields in Java. Like variables, slots are bound to values; they don't actually contain their data. A simple Dylan class shows how slots are declared:

```
define class <vehicle> (<object>)
  slot serial-number;
  slot owner;
end;
```

The above code would be quick and convenient to write while building a prototype, but it could be improved. The slots have no declared types so they default to `<object>`, and they don't specify default values so they default to `#f`. The following snippet fixes both problems:

```
define class <vehicle> (<object>)
  slot serial-number :: <integer>,
    required-init-keyword: sn:;
  slot owner :: <string>,
    init-keyword: owner:,  // optional
    init-value: "Northern Motors";
end class <vehicle>;
```

The type declarations work just like type declarations anywhere else in Dylan; they limit a binding to objects of a given class or of one of its subclasses, and they let the compiler optimize. The new keywords describe how the slots get their initial values. (The keyword `init-function` may also be used; it must be followed by a function with no arguments and the appropriate return type.)

To create a vehicle object using the new class declaration, a programmer could write one of the following:

```
make(<vehicle>, sn: 1000000)
make(<vehicle>, sn: 2000000, owner: "Sal")
```

In the first example, `make` returns a vehicle with the specified serial number and the default owner. In the second example, `make` sets both slots using the keyword arguments.

Only one of `required-init-keyword`, `init-value` and `init-function` may be specified. However, `init-keyword` may be paired with either of the latter two if desired. More than one slot may be initialized by a given keyword.

Dylan also provides for the equivalent of C++ `static` members, plus several other useful allocation schemes. See the DRM for the full details.

4.3 Getters and Setters

An object's slots are accessed using two functions: a getter and a setter. By default, the getter function has the same name as the slot, and the setter function appends "`-setter`". These functions may be invoked as follows:

```
owner(sample-vehicle);  // returns owner
owner-setter("Faisal", sample-vehicle);
```

Dylan also provides some convenient "syntactic sugar" for these two functions. They may also be written as:

```
sample-vehicle.owner;  // returns owner
sample-vehicle.owner := "Faisal";
```

4.4 Generic functions and Objects

Generic functions, introduced in *Methods and Generic functions*, provide the equivalent of C++ member functions. In the simplest case, just declare a generic function which dispatches on the first parameter.

```
define generic tax (v :: <vehicle>) => (tax-in-dollars :: <float>);

define method tax (v :: <vehicle>) => (tax-in-dollars :: <float>)
  100.00
end;

//=== Two new subclasses of vehicle

define class <car> (<vehicle>)
end;

define class <truck> (<vehicle>)
  slot capacity, required-init-keyword: tons:;
end;

//=== Two new "tax" methods

define method tax (c :: <car> ) => (tax-in-dollars :: <float>)
  50.00
end method;

define method tax (t :: <truck>) => (tax-in-dollars :: <float>)
  // standard vehicle tax plus $10/ton
  next-method() + t.capacity * 10.00
end method;
```

The function `tax` could be invoked as `tax(v)` or `v.tax`, because it only has one argument. Generic functions with two or more arguments must be invoked in the usual Dylan fashion; no syntactic sugar exists to make them look like C++ member functions.

The version of tax for `<truck>` objects calls a special function named `next-method`. This function invokes the next most specific method of a generic function; in this case, the method for `<vehicle>` objects. Parameters to the current method get passed along automatically.

Technically, `next-method` is a special parameter to a method, and may be passed explicitly using `#next`.

```
define method tax
    (t :: <truck>, #next next-method) => (tax-in-dollars :: <float>)
  // standard vehicle tax plus $10/ton
  next-method() + t.capacity * 10.00
end method;
```

Dylan's separation of classes and generic functions provides some interesting design ideas. Classes no longer need to "contain" their member functions; it's possible to write a new generic function without touching the class definition. For example, a module handling traffic simulations and one handling municipal taxes could each have many generic functions involving vehicles, but both could use the same vehicle class.

Slots in Dylan may also be replaced by programmer-defined accessor functions, all without modifying existing clients of the class. The DRM describes numerous ways to accomplish the change; several should be apparent from the preceding discussion. This flexibility frees programmers from creating functions like `GetOwnerName` and `SetOwnerName`, not to mention the corresponding private member variables and constructor code.

For even more creative uses of generic functions and the Dylan object model, see the chapter on *Multiple Dispatch*.

4.5 Initializers

The `make` function handles much of the drudgery of object construction. It processes keywords and initializes slots. Programmers may, however, customize this process by adding methods to the generic function `initialize`. For example, if vehicle serial numbers must be at least seven digits:

```
define method initialize (v :: <vehicle>, #key)
  next-method();
  if (v.serial-number < 1000000)
    error("Bad serial number!");
  end if;
end method;
```

`initialize` methods get called after regular slot initialization. They typically perform error checking or calculate derived slot values. Initialize methods must specify `#key` in their parameter lists.

It's possible to access the values of slot keywords from `initialize` methods, and even to specify additional keywords in the class declaration. See the DRM for further details.

4.6 Abstract Classes and Overriding Make

Abstract classes define the interface, not the implementation, of an object. There are no direct instances of an abstract class. Concrete classes actually implement their interfaces. Every abstract class will typically have one or more concrete subclasses. For example, if plain vanilla vehicles shouldn't exist, `<vehicle>` could be defined as follows:

```
define abstract class <vehicle> (<object>)
  // ...as before
end;
```

The above modification prevents the creation of direct instances of `<vehicle>`. At the moment, calling `make` on this class would result in an error. However, a programmer could add a method to make which allowed the intelligent creation of vehicles based on some criteria, thus making `<vehicle>` an *instantiable abstract class*:

```
define method make
    (class == <vehicle>, #rest keys, #key big?)
 => (vehicle :: <vehicle>)
  if (big?)
    make(<truck>, keys, tons: 2)
  else
    make(<car>, keys)
  end
end method make;
```

A number of new features appear in the parameter list. The expression "`class == <vehicle>`" specifies a *singleton* dispatch, meaning this method will be called only if `class` is exactly `<vehicle>`, not a subclass such as `<car>`. Singleton dispatch is discussed in the chapter on *Multiple Dispatch*. The use of `#rest` and `#key` in the same parameter list means all keyword arguments will be stored in the `keys` parameter but if `big?` is passed it will be bound to the variable by the same name. The new make method could be invoked in any of the following fashions:

```
let x = 1000000;
make(<vehicle>, sn: x, big?: #f); //=> car
make(<vehicle>, sn: x, big?: #t); //=> truck
make(<vehicle>, sn: x);           //=> car
```

Methods added to `make` don't actually need to create new objects. Dylan officially allows them to return existing objects. This can be used to manage lightweight shared objects, such as the "flyweights" or "singletons" described by

Gamma, et al., in Design Patterns.

CHAPTER

FIVE

MULTIPLE DISPATCH

Multiple dispatch is one of the most powerful and elegant features of Dylan. As explained in the section on *generic functions and objects*, Dylan methods are declared separately from the classes upon which they act. *Polymorphism*, the specialization of methods for use with particular classes, can be implemented by declaring several methods with different parameters and attaching them to one generic function:

```
define generic inspect-vehicle (v :: <vehicle>, i :: <inspector>) => ();

define method inspect-vehicle (v :: <vehicle>, i :: <inspector>) => ()
  look-for-rust(v);
end;

define method inspect-vehicle (car :: <car>, i :: <inspector>) => ()
  next-method();  // perform vehicle inspection
  check-seat-belts(car);
end;

define method inspect-vehicle (truck :: <truck>, i :: <inspector>) => ()
  next-method();  // perform vehicle inspection
  check-cargo-attachments(truck);
end;
```

However, different types of vehicle inspectors may have different policies. A state inspector, in addition to the usual procedures, will also typically check a car's insurance policy. To implement this, add another method to the generic function `inspect-vehicle`:

```
define method inspect-vehicle (car :: <car>, i :: <state-inspector>) => ()
  next-method();  // perform car inspection
  check-insurance(car);
end;

let inspector = make(<state-inspector>);
let car = make(<car>);
inspect-vehicle(car, inspector);
```

Calling the generic function `inspect-vehicle` with these arguments performs three separate tasks: `look-for-rust`, `check-seat-belts` and `check-insurance`. The most specific method on `inspect-vehicle` – the one for the classes `<car>` and `<state-inspector>` – is invoked first and calls `next-method` to invoke the less-specific methods in turn.

For an exact definition of "specific", see the DRM.

5.1 Dispatching on Specific Objects

Dylan also allows functions to dispatch on specific objects. For example, state inspectors might pass the governor's car without actually looking at it. Dylan expresses this situation using *singletons*, objects which are treated as though they were in a class of their own. For example:

```
define constant $governors-car = make(<car>);

define method inspect-vehicle
    (car == $governors-car, i :: <state-inspector>) => ()
  wave-through(car);
end;
```

(In this example, none of the usual inspection methods will be invoked since the above code doesn't call `next-method`.)

CHAPTER

SIX

MODULES & LIBRARIES

Modules and libraries provide the structure of a Dylan program. Modules represent namespaces and control access to objects and functions. Libraries contain modules, and act as units of compilation in a finished Dylan program.

6.1 Simple Modules

Modules import the symbols of other modules and export their own. The dependencies between modules must form a directed, acyclic graph. Two modules may not use each other, and no circular dependencies may exist.

Modules only export variables. Since the names of classes and generic functions are actually stored in variables, this represents no hardship. A sample module containing the vehicle classes from earlier chapters might resemble:

```
define module vehicles
  use dylan;
  export
    <vehicle>,
      serial-number,
      owner, owner-setter,
      tax,
    <car>,
    <truck>,
      capacity;
end module;
```

Like all normal modules, this one uses the `dylan` module, which contains all of the standard built-in functions and classes. In turn, the `vehicles` module exports all three of the vehicle classes, the generic function `tax`, several getter functions and a single setter function.

To control access to a slot, export some combination of its getter and setter functions. To make a slot public, export both. To make it read-only, export just the getter function. To make it private, export neither. In the above example, the slot `serial-number` is read-only, while the slot `owner` is public.

Note that when some module adds a method to a generic function, the change affects all modules using that function. The new method actually gets added *into* the variable representing the generic function. Since the variable has been previously exported, all clients can access the new value.

6.2 Import Options

Dylan allows very precise control over how symbols are imported from other modules. For example, individual symbols may be imported by name. They may be renamed, either one at a time, or by adding a prefix to all of a module's symbols at once. Some or all of them may be re-exported immediately. See the DRM for specific examples.

Dylan's import system has a number of advantages. Name conflicts occur rarely. Programmers don't need to define or maintain function prototypes. There's no need for header files. Modules may also provide different interfaces to the same objects – one module exports a complete interface, which another module imports, redefines and re-exports.

6.3 Libraries

Libraries contain modules. For example, the `dylan` library contains the `dylan` module described earlier, the `extensions` module, and possibly several other implementation-dependent modules. Note that a library and a module may share the same name. Modules with the same name may also appear in more than one library.

By default, a Dylan environment provides a library called `dylan-user` for the convenience of the programmer. This is typically used for short, single library programs which depend only on modules found in the Dylan library.

Additionally, every library contains an implicit module, also known as `dylan-user`, which imports all of the modules found in the `dylan` library. This may be used for single module programs. Many Dylan environments, however, use it to bootstrap new library definitions. The vehicle library, for example, might be defined as follows in a `dylan-user` module:

```
define library vehicles
  use dylan;            // This is the library!
  export                // These are modules.
    vehicles,           // (Defined above.)
    traffic-simulation,
    crash-testing,
    inspection;         // (Hypothetical.)
end library vehicles;
```

This library could in turn be imported by another library:

```
define library vehicle-application
  use dylan;
  use my-gui-classes;
  use vehicles;
end;
```

Libraries import other libraries and export modules, whereas modules import other modules and export variables. In general, a module may import any module found in its own library or exported from a library imported by its own library. The following module, for example, could belong to the `vehicle-application` library.

```
define module sample-module
  // module name         source library
  use dylan;          // dylan
  use extensions;     // dylan
  use menus;          // my-gui-classes
  use vehicles;       // vehicles
  use inspection;     // vehicles
end module;
```

6.4 Sealing

Classes and generic functions may be *sealed* using a number of Dylan forms. This prevents code in other libraries from subclassing objects or adding methods to generic functions, and lets the compiler optimize more effectively. Both classes and generic functions are sealed by default.

To allow code in other libraries to subclass a given class, declare it as `open`:

```
define open class <sample> (<object>) end;
```

To allow other libraries to add methods to a generic function, use a similar syntax:

```
define open generic sample-function (o :: <object>) => ();
```

A third form, `define sealed domain`, partially seals a generic function, disallowing only some additions from outside a library.

For more information on sealing, see the chapter "Controlling Dynamism" in the DRM.

CHAPTER

SEVEN

CONDITIONS

Dylan offers sophisticated exception handling, allowing programs to recover smoothly from error conditions. Like C++, Dylan represents errors with objects. Dylan also supports advisory warnings and potentially correctable errors.

When something unusual happens, a program can *signal* a *condition*. *Handlers* specify how to react to various sorts of conditions.

7.1 Signaling

Unlike the exceptions of C++ or Java, signaling a condition does *not* itself cause the current function or block to exit. Instead, calling the `signal` function is just like calling any other function. The `signal` function just locates an appropriate handler and calls it normally.

One consequence of this is that a handler can signal another condition in a very straightforward manner. For example, imagine a program that searches for a person by name, and if it cannot find one, it searches for a pet by the same name, and if it cannot find the pet either, it breaks into the debugger. Given an unknown name, you might see the following backtrace:

1. `break({condition <key-not-found-error>: "Toby"})`
2. `handle-no-pet-found({condition <key-not-found-error>: "Toby"})`
3. `signal({condition <key-not-found-error>: "Toby"})`
4. `element(*pets*, "Toby")`
5. `find-pet("Toby")`
6. `handle-no-person-found({condition <key-not-found-error>: "Toby"})`
7. `signal({condition <key-not-found-error>: "Toby"})`
8. `element(*people*, "Toby")`
9. `find-person-or-pet("Toby")`

Here you can see that each failure signals a new condition, but the program never backs out of a function call; it just keeps going, leaving the history of conditions for you to examine.

7.2 Handlers

A function *establishes a handler* with the `let handler` statement. The handler remains in effect until the function exits. Other functions called by the first can establish new handlers. When the `signal` function looks for a handler, it looks for the most recently established handler that fits the condition.

In the example above, there are two handlers: `handle-no-person-found` and `handle-no-pet-found`. Both handlers are for the `<key-not-found-error>` condition. Let us assume that the `find-person-or-pet` function established the `handle-no-person-found` handler and that the `find-pet` function established the `handle-no-pet-found` handler. Since `handle-no-pet-found` was established later, it was the one chosen and called by `signal` in frame 3.

The code to establish the handlers may have looked like this:

```
let handler <key-not-found-error> = handle-no-pet-found;
```

A handler can be a normal function, but it can also be a local method or bare method, complete with access to local variables.

7.3 Recovery

Dylan's condition system allows it to offer a couple of useful error recovery techniques.

7.3.1 Returning from `signal`

Because a `signal` call is just like any other function call, it can return values. It returns whatever values the handler function returns. In the above example, `signal` never returns because we break into the debugger, and the `element` function wouldn't do anything with the value if it did return, but your own code could call `signal` and handle any return values appropriately.

This technique allows you to use conditions as a sort of callback. You can establish a condition handler that returns a rarely-needed value, and another deeply nested function could retrieve that value if needed by signaling that condition and then taking the return value of the `signal` function.

7.3.2 Restart handlers

You can recover from a problem by returning a fall-back value from the `signal` function, but that technique has limitations. It does not provide much encapsulation or allow for complicated recovery information, and the recovery information has to be processed locally.

Another way to return recovery information is through the use of a *restart*. A restart is a condition that includes recovery information. But unlike most conditions, this condition provides a solution instead of indicating a problem. A restart handler — which may be established anywhere useful — can use the information included in the restart to work around the problem.

For example, if the `find-pet` function above does not succeed, the `handle-no-pet-found` function could create a new goldfish object and signal a `<possible-new-pet>` restart, returning the goldfish. The callers of `find-pet` would establish a handler for that restart. The restart handler established by the `find-person-or-pet` function would probably ignore the goldfish and signal a different condition instead, but other callers may establish different restart handlers with the appropriate behavior.

Regardless, when the restart handler finishes, it returns, and then its caller returns, and so on until the original `signal` function returns, at which point the program resumes work where it left off. You cannot use restart handlers or conditions to escape the program's normal flow of control. For that, Dylan offers blocks.

7.4 Blocks

A *block* is a group of statements. As with other control structures, it may return a value. A simple block might appear as follows:

```
block ()
  1 + 1
end; // returns 2
```

But in addition to returning a value normally, a block can use a *nonlocal exit*. This allows the block to exit at any time, optionally returning a value. In some ways, it is similar to the `goto` statement, the `break` statement, or the POSIX `longjmp` function. To use a nonlocal exit, specify a name in the parentheses following a `block` statement. Dylan binds this name to an *exit function* which can be called from anywhere within the block or the functions it calls. The following block returns either `"Weird!"` or `"All's well."`, depending on the color of the sky.

```
block (finished)
  if (sky-is-green())
    finished("Weird!");
  end;
  "All's well."
end block;
```

Many programs need to dispose of resources or perform other cleanup work when exiting a block. The block may contain optional `afterwards` and `cleanup` clauses. Neither affects the block's return value. The `afterwards` clause executes if the block ends normally without using its nonlocal exit, and the `cleanup` clause executes when the block ends whether it ends normally or via nonlocal exit.

```
let fd = open-input-file();
block (return)
  let (errorcode, data) = read-data(fd);
  if (errorcode)
    return(errorcode);
  end if;
  process-data(data);
afterwards
  report-success();
cleanup
  close(fd);
end;
```

7.4.1 Blocks and conditions

In addition to the `afterwards` and `cleanup` clauses, a block may also contain any number of `exception` clauses. The exception clauses establish handlers for a condition much like the `let handler` statement, but before they run the handler calls the block's exit procedure and takes a nonlocal exit. In other words, it takes a short cut out of the normal flow of control. The `signal` function that signaled the condition never returns to its caller. Instead, the program resumes execution after the block.

The end result is similar to the `try...catch...finally` statements of C++ or Java:

```
let fd = open-input-file();
block ()
  let data = read-data(fd);
  process-data(data);
cleanup
  close(fd);
exception (error :: <file-error>)
```

```
  report-problem(error);
end;
```

You can use a block with a restart to abort some work entirely and fall back to the data supplied by the restart object, neatly circumventing the problem mentioned at the end of the Restart handlers section above:

```
let fd = open-input-file();
block ()
  let data = read-data(fd);
  process-data(data);
cleanup
  close(fd);
exception (restart :: <fallback-data-restart>)
  process-data(restart.fallback-data);
end;
```

CHAPTER

EIGHT

COPYRIGHT

Copyright © 1995, 1996, 1998, 1999 Eric Kidd

Copyright © 2002, 2003, 2004, 2011, 2012 The 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.