Overview

This chapter describes the syntax and structure of a Dylan program, from the outside in. This is one of the two defining characteristics of Dylan. The other is the set of objects on which a Dylan program operates; objects and their types are discussed in the following chapters. This section is only an overview; language constructs briefly mentioned here are explained in detail in later sections. A formal specification of Dylan syntax appears in Appendix A, "BNF."

Libraries and Modules

A complete Dylan **program** consists of one or more **libraries**. Some of these libraries are written by the programmer, others are supplied by other programmers or by the Dylan implementation. A library is Dylan's unit of separate compilation and optimization. The libraries that compose a program can be linked together as early as during compilation or as late as while the program is running. Program structure inside of a library is static and does not change after compilation. However, many Dylan implementations provide an incremental compilation feature which allows a library under development to be modified, while the program is running, by modifying and recompiling portions of the library.

A library contains one or more **modules**. A module is Dylan's unit of global name scoping, and thus of modularity and information hiding. A module can be exported from its library; otherwise it is internal to that library. A library can import modules from other libraries. Only an exported module can be imported.

A module contains zero or more source records and a set of bindings.

A **source record** is an implementation-defined unit of source program text. For example, in a file-based development environment each source file would be one source record. As another example, in an interactive Dylan interpreter each executable unit of programmer input would be a source record. The source program text in a source record is a body, a grammatical element used in several places in Dylan.

Bindings

A **binding** is an association of a name with a value. The bindings in a module persist for the life of the program execution. The scope of such a binding is its module. That is, the binding is visible to all source-records in the module. A module can export bindings and can import bindings from other modules. Only an exported binding can be imported. A binding is visible to all source-records in a module that imports it.

A binding may be **specialized**. This restricts the types of values that may be held in the binding. An error will be signaled on any attempt to initialize or assign the binding to a value that is not of the correct type.

A binding is either **constant** or **variable**. A constant (or read-only) binding always has the same value. In contrast, a variable (or writable) binding can have its value changed, using the assignment operator `:=`. Most bindings in a typical Dylan module are constant.

Macros

A **macro** is an extension to the core language that can be defined by the programmer, by the implementation, or as part of the Dylan language specification. Much of the grammatical structure of Dylan is built with macros. A macro defines the meaning of one construct in terms of another construct. The original construct is the call to the macro. The replacement construct is the expansion of the macro. The compiler processes the expansion in place of the call.

Portions of the call to a macro are substituted into part of the macro definition to create the expansion. This substitution preserves the meanings of names. In other words, each name inserted into the expansion from the macro call refers to the same binding that it referred to in the call, and each name inserted into the expansion from the macro definition refers to the same binding that it referred to in the definition.

A macro is named by a binding and thus is available for use wherever that binding is visible. There are three kinds of macros: defining macros, which extend the available set of definitions; statement macros, which extend the available set of statements; and function macros, which look syntactically like function calls but have more flexible semantics.

Bodies

A **body** is a sequence of zero or more constituents. When multiple constituents are present, they are separated by semicolons. When at least one constituent is present, the last constituent can optionally be followed by a semicolon; this allows programmers to regard the semicolon as either a terminator or a separator, according to their preferred programming style.

A **constituent** is either a definition, a local declaration, or an expression. Definitions and local declarations form the structure of a program and do not return values. In contrast, expressions are executed for the values they return and/or the side-effects that they perform.

Definitions

A **definition** is either a call to a user-defined defining macro, a call to a built-in defining macro, or a special definition. Typically, a definition defines a binding in the module containing the definition. Some definitions define more than one binding, and some do not define any bindings.

A **user-defined defining macro** is a macro that defines a definition in terms of other constructs. A call to a user-defined defining macro always begins with the word `define` and includes the name of the defining macro. This name when suffixed by “`-definer`” is the name of a visible binding whose value is the defining macro. The rest of the syntax of a call to a user-defined defining macro is determined by the particular macro. Some definitions include a body. Advanced programmers often define new defining macros as part of structuring a program in a readable and modular way.

A **built-in defining macro** is like a user-defined defining macro but is specified as part of the Dylan language. There are eight built-in defining macros:

`define class`, `define constant`, `define generic`, `define inert`, `define library`, `define method`, `define module`, and `define variable`.

A **special definition** is a definition construct that is built into the grammar of Dylan. There is only one special definition: `define macro`.

An implementation can add new kinds of definitions as language extensions. Such definitions may be implemented as special definitions. However, they will more commonly take the form of user-defined definition macros that are the values of bindings exported by implementation-defined modules.

Local Declarations

A **local declaration** is a construct that establishes local bindings or condition handlers whose scope is the remainder of the body following the local declaration.

Unlike module bindings, local bindings are established during program execution, each time the local declaration is executed. They persist for as long as code in their scope is active. Local bindings persist after the body containing them returns if they are referenced by a method created inside the body and a reference to the method escapes from the body, so that it could be called after the body returns. Unlike module bindings, local bindings are always variable. However, since a local binding has a limited scope, if there is no assignment within that scope, the local binding is effectively constant.

A local binding shadows any module binding with the same name and any surrounding local binding with the same name. The innermost binding is the one referenced.

The name of a local binding cannot be the name of a macro.

There are three kinds of local declaration: local value bindings (`let`), local method bindings (`local`), and condition handler establishment (`let handler`).

The **local value bindings** construct, `let`, executes an expression and locally binds names to the values returned by that expression.

The **local method bindings** construct, `local`, locally binds names to bare methods. These bindings are visible in the remainder of the body and also inside the methods, permitting recursion.

The **condition handler establishing** construct, `let handler`, establishes a function to be called if a condition of a given type is signaled during the execution of the remainder of the body or anything the body calls. The handler is disestablished as soon as the body returns. Unlike the other two kinds of local declaration, `let handler` does not establish any bindings.

Expressions

An **expression** is a construct that is executed for the values it returns and/or the side-effects that it performs. The "active" portions of a Dylan program are expressions. An expression is either a literal constant, a named value reference, a function call, a unary operator call, a binary operator call, an element reference, a slot reference, a parenthesized expression, or a statement.

An **operand** is a restricted expression: it cannot be a unary or binary operator call nor a symbol literal. The other seven forms of expression are allowed. Operands appear in situations in the grammar where an expression is desirable but the full generality of expressions would make the grammar ambiguous.

A **literal constant** directly represents an object. Literal constants are available for numbers, characters, strings, symbols, boolean values, pairs, lists, and vectors. For example:

number	`123, 1.5, -4.0, #x1f4e`
character	`'a', '\n'`
string	`"foo", "line 1\nline 2"`
symbol	`test:, #"red"`
boolean value	`#t, #f`
pair	`#(1 . "one")`
list	`#(1, 2, 3)`
vector	`#[1, 2, 3]`

Literal constants are immutable. Attempting to modify an immutable object has undefined consequences. Immutable objects may share structure. Literal constants that are equal may or may not be identical.

A symbol can be indicated in two ways: as a keyword (for example, `test:`) or as a unique string (for example, `#"red"`). The difference is purely syntactic; the choice is provided to promote program readability.

A string literal can be broken across lines by writing two string literals in a row, separated only by whitespace; they are automatically concatenated (without a newline character).

A **named value reference** returns the value of a visible binding given its name. For example, `foo`. The referenced binding can be a module binding (either constant or variable) or a local binding established by a local declaration or by a parameter list. The value of the binding must not be a macro.

A **reserved word** is a syntactic token that has the form of a name but is reserved by the Dylan language and so cannot be given a binding and cannot be used as a named value reference. There are seven reserved words in Dylan: `define`, `end`, `handler`, `let`, `local`, `macro`, and `otherwise`.

A **function call** applies a function to arguments, and returns whatever values the function returns. The function is indicated by an operand and can be a generic function, a method, or a function macro. The arguments are indicated by expressions separated by commas and enclosed in parentheses. For example, `f(x, y)`. For readability, the comma can be omitted between the two arguments in a keyword/value pair, for example `element(c, k, default: d)` is a function call with four arguments.

A **unary operator call** consists of an operand preceded by one of the two unary operators – (arithmetic negation) or ~ (logical negation). For example, `- x`. This is actually an abbreviated notation for a function call.

A **binary operator call** consists of two expressions separated by one of the binary operators + (addition), – (subtraction), `*` (multiplication), `/` (division), ^ (exponentiation), = (equality), == (identity), < (less than), > (greater than), <= (less than or equal), >= (greater than or equal), ~= (not equal), ~== (not identical), `&` (logical and), `|` (logical or), or `:=` (assignment). When binary operator calls are chained together, they are grouped by rules of precedence and associativity and by parentheses. For example, `(a - b) * x + c * x ^ 2`. Except for the last three operators, a binary operator call is actually an abbreviated notation for a function call. The last three operators (`&`, `|`, and `:=`) are treated specially be the compiler.

An **element reference** consists of an operand that indicates a collection and an expression in square brackets that indicates a key. Instead of a key, there can be multiple expressions separated by commas that indicate array indices. For example, `c[k]` or `a[i, j]`. This is actually an abbreviated notation for a function call.

A **slot reference** is another abbreviated notation for a function call. It consists of an operand that indicates an object, a period, and a named value reference that indicates a one-argument function to apply to the object. Typically the function is a slot getter but this is not required. For example, `airplane.wingspan`.

A **parenthesized expression** is any expression inside parentheses. The parentheses have no significance except to group the arguments of an operator or to turn a general expression into an operand. For example, `(a + b) * c`.

Statements

A **statement** is a call to a statement macro. It begins with the name of a visible binding whose value is a statement macro. The statement ends with the word `end` optionally followed by the same name that began the statement. In between is a program fragment whose syntax is determined by the macro definition. Typically this fragment includes an optional body. For example, `if (ship.ready?) embark(passenger, ship) end if`.

A **statement macro** can be built-in or user-defined.

A **user-defined statement macro** is a macro that defines how to implement a statement in terms of other constructs. Advanced programmers often define new statement macros as part of structuring a program in a readable and modular way.

A **built-in statement macro** is like a user-defined statement macro but is specified as part of the Dylan language. There are nine built-in statement macros: `begin`, `block`, `case`, `for`, `if`, `select`, `unless`, `until`, and `while`.

An implementation can add new kinds of statements as language extensions. Such a statement takes the form of a user-defined statement macro that is the value of a binding exported by an implementation-defined module.

Parameter Lists

Several Dylan constructs contain a **parameter list**, which describes the arguments expected by a function and the values returned by that function. The description includes names, types, keyword arguments, fixed or variable number of arguments, and fixed or variable number of values. The argument names specified are locally bound to the values of the arguments when the function is called. The value names specified are only for documentation.

The syntactic details of parameter lists are described in “Methods” on page 412.

Lexical Syntax

Dylan source code is a sequence of tokens. Whitespace is required between tokens if the tokens would otherwise blend together. Whitespace is optional between self-delimiting tokens. Alphabetic case is not significant except within character and string literals.

Whitespace can be a space character, a tab character, a newline character, or a comment. Implementations can define additional whitespace characters.

A **comment** can be single-line or delimited. Although comments count as whitespace, the beginning of a comment can blend with a preceding token, so in general comments should be surrounded by genuine whitespace.

A **single-line comment** consists of two slash characters in a row, followed by any number of characters up to and including the first newline character or the end of the source record. For example, `// This line is a kludge!`.

A **delimited comment** consists of a slash character immediately followed by a star character, any number of characters including balanced slash-star / star-slash pairs, and finally a star character immediately followed by a slash character. For example, `/* set x to 3 */`.

A single-line comment may appear within a delimited comment; occurances of slash-star or star-slash within the single line comment are ignored.

A **token** is a name, a #-word, an operator, a number, a character literal, a string literal, a symbol literal, or punctuation.

A **name** is one of the following four possibilities:

- An alphabetic character followed by zero or more name characters.
- A numeric character followed by two or more name characters including at least two alphabetic characters in a row.
- A graphic character followed by one or more name characters including at least one alphabetic character.
- A "\" (backslash) followed by a function operator.

An **alphabetic character** is any of the 26 letters of the Roman alphabet in upper and lower case.

A **numeric character** is any of the 10 digits.

A **graphic character** is one of the following:

```
! & * < = > | ^ $ % @ _
```

A **name character** is an alphabetic character, a numeric character, a graphic character, or one of the following:

```
- + ~ ? /
```

The rich set of name characters means that name and operator tokens can blend. Thus Dylan programs usually set off operators with whitespace.

Implementations can add additional characters but programs using them will not be portable.

A **#-word** is one of `#t`, `#f`, `#next`, `#rest`, `#key`, or `#all-keys`. The first two are literal constants, the others are used in parameter lists. Implementations can add additional implementation-defined #-words, but programmers cannot add their own #-words.

An **operator** is one of the following:

+	addition
–	subtraction and negation
*	multiplication
/	division
^	exponentiation
=	equality
==	identity
<	less than
>	greater than
<=	less than or equal
>=	greater than or equal
~=	not equal
~==	not identical
&	logical and
|	logical or
:=	assignment
~	logical negation

Programmers cannot add their own operators.

A **number** is a decimal integer with an optional leading sign, a binary integer, an octal integer, a hexadecimal integer, a ratio of two decimal integers with an optional leading sign, or a floating-point number. The complete syntax of numbers is given in "Numbers" on page 406.

A **character literal** is a printing character (including space, but not ' nor \) or a backslash escape sequence enclosed in a pair of single-quote characters '.

A **string literal** is a sequence of printing characters (including space, but not " nor \) and backslash escape sequences enclosed in a pair of double-quote characters ".

The backslash escape sequences used in character and string literals allow "quoting" of the special characters ', ", and \, provide names for "control" characters such as newline, and allow Unicode characters to be specified by their hexadecimal codes.

A **symbol literal** is a keyword or a unique string. A **keyword** is a name followed immediately by a colon character ":". A **unique string** is a sharp sign "#" followed immediately by a string literal.

Punctuation is one of the following:

`( )`	parentheses
`[ ]`	square brackets
`{, }`	curly brackets
`,`	comma
`.`	period
`;`	semicolon
`=`	defaulting/initialization
`::`	type specialization
`==`	singleton specialization
`=>`	arrow
`#(`	list/pair literal
`#[`	vector literal
`?, ??`	macro pattern variables
`...`	macro ellipsis

Note that some tokens are both punctuation and operators. This ambiguity is resolved by grammatical context.

Note that some punctuation tokens (for example period and equal sign) are capable of blending into some other tokens. Where this can occur, whitespace must be inserted to delimit the token boundary.

Special Treatment of Names

Escaping Names

The escape character (\ ) followed by any name or operator-name has the same meaning as that name or operator-name, except that it is stripped of special syntactic properties. If it would otherwise be a reserved word or operator, it is not recognized as such.

For example, `\if` and `if` are names for the same binding, but `\if` is treated syntactically as a named value reference, while `if` is the beginning of a statement. Similarly, \+ and + refer to the same binding, but the former is treated syntactically as a named value reference, and the latter as an operator.

For reserved words, this allows the names of statement macros to be exported and imported from modules. It does not allow them to be used as the names of local bindings, nor does it allow them to be executed. (That is, they cannot be used as bindings to runtime values.)

For operators, it allows the operator to be used where a named value reference is required, for example as the name in a method definition, as an argument to a function, or in a `define module` export clause. This feature can only be used for operators which provide a shorthand for a function call. It cannot be used for special operators.

Function Call Shorthand

Dylan provides convenient syntax for calling a number of functions. These include the operators which are not special operators, the array reference syntax, and the singleton syntax.

In all cases, the syntax is equivalent to using the name of the function in the current environment. The syntax does not automatically refer to a binding in the Dylan module.

Top-Level Definitions

Dylan's built-in defining macros can only be used at top level. When the expansion of a user-defined macro includes a call to a built-in defining macro, the user-defined macro also can only be used at top level.

A constituent is **at top level** if and only if it is a direct constituent of a body, no preceding constituent of that body is a local declaration, and the body is either the body of a source record or the body of a `begin` statement that is itself a constituent at top level. When a constituent appears inside a call to a macro, whether that constituent is at top level must be determined after macro expansion.

The effect of the above rule is that a constituent at top level is not in the scope of any local declarations, is not subject to any condition handlers other than default handlers, and is not affected by any flow of control constructs such as conditionals and iterations. This restriction enhances the static nature of definitions.

Dylan Interchange Format

The **Dylan interchange format** is a standard file format for publishing Dylan source code. Such a file has two parts, the **file header** and the **code body**. The file header comes before the code body.

The code body consists of a source record.

The file header consists of one or more keyword-value pairs, as follows:

- A keyword is a letter, followed by zero or more letters, digits, and hyphens, followed by a colon, contains only characters from the ISO 646 7-bit character set, and is case-independent.
- A keyword begins on a new line, and cannot be preceded by whitespace.
- All text (excluding whitespace) between the keyword and the next newline is considered to be the value. Additional lines can be added by having the additional lines start with whitespace. Leading whitespace is ignored on all lines.
- The meaning of the value is determined by the keyword.
- Implementations must recognize and handle standardized keywords properly, unless the specification for a keyword explicitly states that it can be ignored.
- When importing a file, implementations are free to ignore any non-standard keyword-value pairs that they do not recognize.
- When exporting a file, implementations must use standard keywords properly. Implementations are free to use non-standard keywords.
- The definition of a keyword may specify that the keyword may occur more than once in a single file header. If it does not, then it is an error for the keyword to occur more than once. If it does, it should specify the meaning of multiple occurances.

The file header cannot contain comments, or other Dylan source code.

Blank lines may not appear in the file header. A blank line defines the end of the file header and the beginning of the code body. The blank line is not part of the code body. (A "blank line" is a line consisting of zero or more space or tab characters, ending in a newline character.)

The following standard keywords are defined:

`language:` ***language-name*** **[Header keyword]**

The source record in the file is written in the named language. The only portable value for this keyword is `infix-dylan`.

`module:` ***module-name*** **[Header keyword]**

The source record in the file is associated with the named module. This keyword is required.

`author:` ***text*** **[Header keyword]**
`copyright:` ***text*** **[Header keyword]**
`version:` ***text*** **[Header keyword]**

These are provided for standardization. They are optional, and can be ignored by the implementation.

A typical Dylan source file might look like this:

```
module: quickdraw
author: J. Random Rect
        Linear Wheels, Inc., "Where quality is a slogan!"
        rect@linear.com
copyright: (c) 1995 Linear Wheels, Inc., All rights reserved
version: 1.3 alpha (not fully tested)

define constant $black-color = ...
```

Naming Conventions

Several conventions for naming module bindings help programmers identify the purposes of bindings. In general, the names of bindings do not affect the semantics of a program, but are simply used to improve readability. (The exceptions to this rule are the "`-definer`" suffix used by definition macros, and the "`-setter`" suffix, described below.)

■ Module bindings used to hold types begin and end with angle brackets.

```
<window>
<object>
<character>
<number>
<stream>
<list>
```

■ Variable module bindings begin and end with asterisks.

```
*parse-level*
*incremental-search-string*
*machine-state*
*window-count*
```

■ Program constants begin with a dollar sign.

```
$pi
$end-of-file
```

■ The names of most predicate functions end with a question mark. Predicates are functions which return a true or false value.

```
subclass?
even?
instance?
```

■ Operations that return a value similar to one of their arguments and which also destructively modify the argument end in a `!`. (It will often also be the case that destructive and non-destructive variations of the function exist.) `!` isn't a universal warning that an operation is destructive. Destructive functions that return other values (like `-setter` functions and `pop`) don't need to use the `!` convention.

```
reverse!
sort!
```

- Operations that retrieve a value from a location are called **getters**. Operations that store into a location are called **setters**. In general, getters and setters come in pairs. Setter binding names are derived by appending "`-setter`" to the corresponding getter binding name. This convention is used to generate setter names automatically, and it is used by **`:=`**, the assignment operator, to find the setter that corresponds to a given getter.

`element`	`element-setter`
`size`	`size-setter`
`color`	`color-setter`