Overview

A **macro** is an extension to the core language that can be defined by the user, 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 compiler substitutes the new construct for the original. The purpose of macros is to allow programmers to extend the Dylan language, for example by creating new control structures or new definitions. Unlike C, Dylan does not intend macros to be used to optimize code by inlining. Other parts of the language, such as sealing and `define constant`, address that need.

Throughout this chapter, *italic font* and SMALL CAPS are used to indicate references to the formal grammar given in Appendix A, "BNF."

Compilation and Macro Processing

Compilation consists of six conceptual phases:

1. Parsing a stream of characters into tokens, according to the lexical grammar in Appendix A, "BNF."
2. Parsing a stream of tokens into a program, according to the phrase grammar in Appendix A, "BNF."
3. Macro expansion, which translates the program to a core language.
4. Definition processing, which recognizes special and built-in definitions and builds a compile-time model of the static structure of the program.
5. Optimization, which rewrites the program for improved performance.
6. Code generation, which translates the program to executable form.

Portions of a program can be macro calls. Macro expansion replaces a macro call with another construct, which can itself be a macro call or contain macro calls. This expansion process repeats until there are no macro calls remaining in the program, thus macros have no space or speed cost at run time. Of course, expanding macros affects the speed and space cost of compilation.

This document was created with FrameMaker 4.0.2

A macro definition describes both the syntax of a macro call and the process for creating a new construct to replace the macro call. Typically the new construct contains portions of the old one, which can be regarded as arguments to the macro. A macro definition consists of a sequence of rewrite rules. The left-hand side of each rule is a pattern that matches a macro call. The right-hand side is a template for the expansion of a matching call. Pattern variables appearing in the left-hand side act as names for macro arguments. Pattern variables appearing in the right-hand side substitute arguments into the expansion. Macro arguments can be constrained to match specified elements of the Dylan grammar. Auxiliary rule sets enhance the rewrite rule notation with named subrules.

Some implementations and a future version of the Dylan language specification might allow macro expansions to be produced by compile-time computation using the full Dylan language and an object-oriented representation for programs. Such a "procedural macro" facility is not part of Dylan at this time.

The input to, and output from, macro expansion is a fragment, which is a sequence of elementary fragments. An elementary fragment is one of the following:

- A token: the output of the lexical grammar. The bracket tokens `(`, `)`, `[`, `]`, `{`, `}`, `#(`, and `#[` are not allowed. Core reserved words (except `otherwise`), BEGIN-WORDS, and FUNCTION-WORDS are not allowed unless quoted with backslash.
- A bracketed fragment: balanced brackets (`()`, `[]`, or `{}`) enclosing a fragment.
- A macro call fragment: a macro call.
- A parsed fragment: a single unit that is not decomposable into its component tokens. It has been fully parsed by the phrase grammar. A parsed fragment is either a function call, a list constant, a vector constant, a definition, or a local declaration.

The second and third phases of compilation (parsing and macro expansion) are interleaved, not sequential. The parsing phase of the compiler parses a macro call just enough to find its end. See *definition-macro-call*, *statement*, *function-macro-call*, *body-fragment*, *list-fragment*, and *basic-fragment* in Appendix A, "BNF." This process of parsing a macro call also parses any macro calls nested inside it. The result is a macro call fragment.

This loose grammar for macro calls gives users a lot of flexibility to choose the grammar that their macros will accept. For example, the grammar of macro calls doesn't care whether a bracketed fragment will be interpreted as an argument list, a parameter list, a set of `for` clauses, or a module import list.

The compiler delays computing the expansion of a macro call fragment until it is needed. If an argument to a macro is a macro call, the outer macro call is always expanded first. When the compiler computes the expansion of a macro call fragment, it obeys the macro's definition. Constraints on pattern variables can cause reparsing of portions of the macro call.

A *constituent*, *operand*, or *leaf* that is a macro call expands the macro during or before the definition processing and optimization phases. The compiler brackets the expansion in `begin` … `end`, using the standard binding of `begin` in the Dylan module, and then reparses it as a *statement*. This reparsing may discover more macro calls. A parse error while reparsing a macro expansion could indicate an invalid macro definition or an incorrect call to the macro that was not detected during pattern matching. Once the cycle of macro expansion and reparsing has been completed, no tokens, bracketed fragments, or macro call fragments remain and the entire source record has become one parsed fragment.

This `begin` … `end` bracketing ensures that the expansion of a macro call will not be broken apart by operator precedence rules when the macro call is a subexpression. Similarly it ensures that the scopes of local declarations introduced by a macro will not extend outside that macro expansion when the macro call is a statement in a body.

The fragment produced by parsing a macro call, which is the input to macro expansion, is as follows:

- Local declarations and special definitions are parsed fragments.
- Calls to macros are macro call fragments.
- List constants and vector constants are parsed fragments.
- Anything in brackets is a bracketed fragment.
- If the macro call was not the result of macro expansion, everything else is represented as sequences of tokens. There are a few restrictions on the tokens, for example semicolons must appear in certain places and bare brackets cannot appear; for details see the definition of *body-fragment* and *list-fragment* in Appendix A, “BNF.”

- In a macro call that is the result of macro expansion, additional items can be parsed fragments, due to pattern-variable substitution.
- Many built-in macros expand into implementation-specific parsed fragments.

The fragment produced by parsing an expression is as follows:

- An expression consisting of a single token returns a one-token fragment. This will be a variable-name, non-collection literal, or SYMBOL.
- An expression consisting of just a string-literal returns a one-token fragment. If the string-literal consists of multiple STRING tokens, they are concatenated into a single STRING token.
- An expression consisting of just a list constant or a vector constant returns a list constant or vector constant fragment.
- An expression consisting of just a statement or function-macro-call returns a macro call fragment.
- An operator call, slot reference, or element reference that calls a function macro returns a macro call fragment.
- A function call, operator call, slot reference, or element reference that calls something other than a function macro returns a function call fragment.
- Enclosing an expression in parentheses does not change how it parses.

The term "parsed expression fragment" refers to any of the above.

The parser recognizes parsed fragments as well as raw tokens. The nonterminals *definition* and *local-declaration* in the phrase grammar accept parsed fragments of the same kind. The nonterminal *operand* accepts parsed function call fragments and macro call fragments. The nonterminal *literal* accepts list constant and vector constant fragments. The nonterminal *simple-fragment* accepts parsed function call fragments and macro call fragments. The nonterminal *macro* accepts macro call fragments. The parser expands bracketed fragments into their constituent tokens before parsing them.

Extensible Grammar

There are three kinds of macros: definition macros, which extend the available set of definitions; statement macros, which extend the available set of statements; and function macros, which syntactically resemble function calls but are more flexible. Named value references and local declarations cannot be macro calls. Only statements, function calls, and definitions are extensible.

Definition Macros

A definition macro extends the *definition-macro-call* production of the Dylan phrase grammar to recognize additional constructs as valid definitions, by creating a new DEFINE-BODY-WORD that is recognized by the following grammar line:

definition-macro-call:
`define` *modifiers*$_{opt}$ DEFINE-BODY-WORD *body-fragment*$_{opt}$ *definition-tail*

or by creating a new DEFINE-LIST-WORD that is recognized by the following grammar line:

definition-macro-call:
`define` *modifiers*$_{opt}$ DEFINE-LIST-WORD *list-fragment*$_{opt}$

This allows programmers to extend Dylan by defining new kinds of definitions. The syntax of the definition must be parseable by one of these two predefined grammar rules. The first handles body-style definitions like `define class`, `define method`, and `define module`, while the second handles list-style definitions like `define constant`. See Appendix A, "BNF," for the details.

The new DEFINE-BODY-WORD or DEFINE-LIST-WORD becomes a partially reserved word in each module where the macro definition is visible. In particular a DEFINE-BODY-WORD or DEFINE-LIST-WORD cannot be used as a modifier in a definition. It can still be used as a variable-name.

Statement Macros

A statement macro extends the *statement* production of the Dylan phrase grammar to recognize additional constructs as valid statements, by creating a new BEGIN-WORD that is recognized by the following grammar line:

statement:
BEGIN-WORD *body-fragment*$_{opt}$ *end-clause*

The new BEGIN-WORD becomes a reserved word in each module where the macro definition is visible. It can only be used at the beginning and end of this new statement.

Function Macros

A function macro extends the *function-macro-call* production of the Dylan phrase grammar to recognize additional constructs, by creating a new FUNCTION-WORD that is recognized by the following grammar line:

function-macro-call:
FUNCTION-WORD (*body-fragment*$_{opt}$)

In addition, a function macro can be invoked by any of the shorthand syntax constructs available for invoking functions. In this case, the arguments are always parsed expression fragments, as described on page 144.

The new FUNCTION-WORD becomes a reserved word in each module where the macro definition is visible. It can only be used at the beginning of a macro call.

Macro Names

A macro is named by a constant module binding. The macro is available to be called in any scope where this binding is accessible. Macro names can be exported and can be renamed during module importing just like any other module binding. Macro bindings are constant and cannot be changed by the assignment operator `:=`.

The name bound to a definition macro is the macro's DEFINE-BODY-WORD or DEFINE-LIST-WORD suffixed by "`-definer`". This suffixing convention is

analogous to the naming convention for setters and allows the DEFINE-BODY-WORD or DEFINE-LIST-WORD to be used for another purpose. The name bound to a statement macro is the macro's BEGIN-WORD. The name bound to a function macro is the macro's FUNCTION-WORD.

A named value reference is not allowed when the value of the binding is a macro, because macros are not run-time objects.

A macro cannot be named by a local binding. Macro definitions are always scoped to modules.

Attempting to create a local binding that shadows a binding to a macro is an error.

Reserved words created by a macro definition are reserved in any module where the binding that names the macro is accessible. In other modules, the same words are ordinary names. Each module has an associated syntax table which is used when parsing code associated with that module. The syntax table controls the lexical analyzer's assignment of names to the DEFINE-BODY-WORD, DEFINE-LIST-WORD, BEGIN-WORD, and FUNCTION-WORD categories. Importing a macro into a module makes the same modifications to that module's syntax table that would be made by defining that macro in the module. If a definition macro is renamed when it is imported, the DEFINE-BODY-WORD or DEFINE-LIST-WORD derives from the new name. If the new name does not end in "`-definer`", the imported macro cannot be called.

A NAME or UNRESERVED-NAME in the lexical grammar can be a backslash ('\') character followed by a word. This prevents the word from being recognized as a reserved word during parsing, but does not change which binding the word names. Quoting the name of a statement or function macro with a backslash allows the name to be mentioned without calling the macro, for example to export it from a module.

When a binding that names a macro is exported from a module that is exported from a library, clients of that library can call the macro. Information derived from the macro definition goes into the library export information part of the library description.

Rewrite Rules

The grammar of a macro definition is **`define macro`** *macro-definition*. For details see Appendix A, "BNF."

If the optional NAME at the end of a *macro-definition* is present, it must be the same NAME that appears at the beginning of the *macro-definition*.

The kind of macro being defined, and thus the Dylan grammar production that this macro extends, is determined by which kind of rules appear in the macro's *main-rule-set*.

The NAME preceding the *main-rule-set* is the name of the binding whose value is this macro. It must be consistent with each left-hand side of the *main-rule-set*. It can be any name, even a reserved word or backslash followed by an operator. For statement and function macros this NAME must be the same as the NAME that appears as the first token in each *main-rule-set* pattern. For definition macros this NAME must be the same as the NAME in the *xxx-style-definition-rule* with the suffix "`-definer`" added.

A NAME can belong to more than one of the lexical categories BEGIN-WORD, FUNCTION-WORD, DEFINE-BODY-WORD, and DEFINE-LIST-WORD. A NAME cannot belong to both BEGIN-WORD and FUNCTION-WORD. A NAME cannot belong to both DEFINE-BODY-WORD and DEFINE-LIST-WORD.

For simplicity of documentation, the *xxx-style-definition-rule* productions are written ambiguously. The NAME in the left-hand side of the rule must be the NAME immediately following `define macro` with the "`-definer`" suffix removed, not an arbitrary NAME, which would be ambiguous with *modifier*.

The general idea is that the *main-rule-set* is an ordered sequence of rewrite rules. Macro expansion tests the macro call against each left-hand side in turn until one matches. The corresponding right-hand side supplies the new construct to replace the macro call. The left- and right-hand sides can contain pattern variables. The portion of the macro call that matches a particular pattern variable on the left replaces each occurrence of that pattern variable on the right. It is an error for the right-hand side of a rule to contain a pattern variable that does not appear on the left-hand side of the same rule.

If none of the left-hand sides match, the macro call is invalid. If more than one left-hand side matches, the first matching rule is used. Note that (as described

in the next section) a pattern variable with a wildcard constraint can match an empty portion of the macro call. A comma or a semicolon followed by a pattern variable with a wildcard constraint also can match an empty portion of the macro call. Do not assume that only an empty pattern can match an empty input. In general when writing recursive rewrite rules it is better to put the base case first, before the inductive cases, in case an inductive case rewrite rule might match a base case input.

The punctuation marks `?,` `??,` and `?=` used in patterns and templates are customarily written without any whitespace following them.

Patterns

Approximately speaking, a pattern looks like the construct that it matches, but contains pattern variables that bind to portions of the construct. Hence a left-hand side in the *main-rule-set* looks like a macro call. However, the grammar of patterns is not the same as the grammar of programs, but contains just what is required to match the portions of the Dylan grammar that are extensible by macros. Patterns have a simple nested grammar, with semicolons, commas, and brackets used to indicate levels of nesting. See the definition of *pattern* in Appendix A, "BNF."

A pattern matches a fragment (a sequence of elementary fragments) by executing the following algorithm from left to right. It is easy to create patterns that are ambiguous when considered as grammars. This ambiguity is resolved by the left to right processing order and the specified try-shortest-first order for matching wildcards. Pattern matching succeeds only if all sub-patterns match. If pattern matching fails, the current rule fails and control passes to the next rule in the current rule set. If all patterns in a rule set fail to match, the macro call is invalid.

Multiple occurrences of the same pattern variable name in a single rule's left-hand side are not valid.

A *pattern* matches a fragment as follows:

- If the pattern consists of just one pattern-list, go to the next step. Otherwise divide the pattern into subpatterns and the fragment into subfragments at semicolons, and match subpatterns to subfragments individually in order. The subpatterns and subfragments do not include the semicolons that

separate them. Suppose the pattern consists of N + 1 pattern-lists separated by N semicolons. Locate the first N semicolons in the fragment (without looking inside of elementary fragments) and divide up the fragment into subfragments accordingly. The match fails if the fragment contains fewer than N - 1 semicolons. As a special case, if the fragment contains N - 1 semicolons, the match still succeeds and the last subfragment is empty. If the fragment contains more than N semicolons, the extra semicolons will be in the last subfragment.

A *pattern-list* matches a fragment as follows:

- If the pattern-list consists of just a pattern-sequence, go to the next step. If the pattern-list consists of just a property-list-pattern, go to that step. Otherwise divide the pattern-list into subpatterns and the fragment into subfragments at commas, and match subpatterns to subfragments individually in order. The subpatterns and subfragments do not include the commas that separate them. Suppose the pattern consists of N + 1 subpatterns separated by N commas. Locate the first N commas in the fragment (without looking inside of elementary fragments) and divide up the fragment into subfragments accordingly. The match fails if the fragment contains fewer than N - 1 commas. As a special case, if the fragment contains N - 1 commas, the match still succeeds and the last subfragment is empty. If the fragment contains more than N commas, the extra commas will be in the last subfragment. Note that the subdivision algorithms for commas and semicolons are identical.

A *pattern-sequence* matches a fragment as follows:

- Consider each simple-pattern in the pattern-sequence in turn from left to right. Each simple-pattern matches an initial subsequence of the fragment and consumes that subsequence, or fails. The entire pattern match fails if any simple-pattern fails, if the fragment is empty and the simple-pattern requires one or more elementary fragments, or if the fragment is not entirely consumed after all simple-patterns have been matched. There is a special backup and retry rule for wildcards, described below.

A *simple-pattern* matches a fragment as follows:

- A NAME or => consumes one elementary fragment, which must be identical to the *simple-pattern*. A NAME matches a name that is spelled the same, independent of modules, lexical scoping issues, alphabetic case, and backslash quoting. As a special case, after the word `otherwise`, an => is

optional in both the pattern and the fragment. Presence or absence of the arrow in either place makes no difference to matching.

- A *bracketed-pattern* matches and consumes a *bracketed-fragment*. If the enclosed *pattern* is omitted, the enclosed *body-fragment* must be empty, otherwise the enclosed *pattern* must match the enclosed *body-fragment* (which can be empty). The type of brackets (`()`, `[]`, or `{}`) in the *bracketed-fragment* must be the same as the type of brackets in the *bracketed-pattern*.

A *binding-pattern* matches a fragment as follows:

- *pattern-variable* **::** *pattern-variable* consumes as much of the fragment as can be parsed by the grammar for *variable.* It matches the first pattern-variable to the *variable-name* and the second to the *type*, a parsed expression fragment. If no specializer is present, it matches the second pattern-variable to a parsed expression fragment that is a named value reference to `<object>` in the Dylan module. This matching checks the constraints on the pattern variable, fails if the constraint is not satisfied, and binds the pattern variable to the fragment.
- *pattern-variable* **=** *pattern-variable* consumes as much of the fragment as can be parsed by the grammar for *variable* = *expression.* It matches the first pattern-variable to the *variable,* a fragment, and the second to the *expression*, a parsed expression fragment.
- *pattern-variable* **::** *pattern-variable* **=** *pattern-variable* consumes as much of the fragment as can be parsed by the grammar for *variable* = *expression.* It matches the first two pattern-variables the same as the first kind of *binding-pattern* and it matches the third pattern-variable the same as the second kind of *binding-pattern.*

A *pattern-variable* matches a fragment as follows:

- When the constraint is a wildcard constraint (see "Pattern Variable Constraints" on page 154), the pattern variable consumes some initial subsequence of the fragment, using a backup and retry algorithm. First, the wildcard consumes no elementary fragments, and matching continues with the next *simple-pattern* in the *pattern-sequence*. If any *simple-pattern* in the current *pattern-sequence* fails to match, back up to the wildcard, consume one more elementary fragment than before, and retry matching the rest of the *pattern-sequence*, starting one elementary fragment to the right of the previous start point. Once the entire *pattern-sequence* has successfully

matched, the pattern variable binds to a fragment consisting of the sequence of elementary fragments that it consumed.

- It is an error for more than one of the *simple-patterns* directly contained in a *pattern-sequence* to be a wildcard.
- When the constraint is other than a wildcard constraint, the pattern variable consumes as much of the fragment as can be parsed by the grammar specified for the constraint in "Pattern Variable Constraints" on page 154. If the parsing fails, the pattern match fails. The pattern variable binds to the fragment specified in "Pattern Variable Constraints." This can be a parsed fragment rather than the original sequence of elementary fragments.
- The ellipsis *pattern-variable*, `...`, can only be used in an auxiliary rule set. It represents a pattern variable with the same name as the current rule set and a wildcard constraint.

A *property-list-pattern* matches a fragment as follows:

- Parse the fragment using the grammar for $property\text{-}list_{opt}$. If the parsing fails or does not consume the entire fragment, the pattern match fails.
- If the *property-list-pattern* contains `#key` and does not contain `#all-keys`, the match fails if the SYMBOL part of any property is not the NAME in some *pattern-keyword* in the *property-list-pattern*. Comparison of a SYMBOL to a NAME is case-insensitive, ignores backslash quoting, and is unaffected by the lexical context of the NAME.
- If the *property-list-pattern* contains `#rest`, bind the pattern variable immediately following `#rest` to the entire fragment. If the pattern variable has a non-wildcard constraint, parse the *value* part of each property according to this constraint, fail if the parsing fails or does not consume the entire *value* part, and substitute the fragment specified in "Pattern Variable Constraints" on page 154 for the *value* part.
- Each *pattern-keyword* in the *property-list-pattern* binds a pattern variable as follows:
  - A single question mark finds the first property whose SYMBOL is the NAME of the *pattern-keyword* . Comparison of a SYMBOL to a NAME is case-insensitive, ignores backslash quoting, and is unaffected by the lexical context of the NAME. If the *pattern-keyword* has a non-wildcard constraint, parse the property's *value* according to this constraint, fail if the parsing fails or does not consume the entire *value* , and bind the pattern variable to the fragment specified in "Pattern Variable

Constraints" on page 154. If the *pattern-keyword* has a wildcard constraint, bind the pattern variable to the property's *value* .

- A double question mark finds every property with a matching SYMBOL, processes each property's *value* as for a single question mark, and binds the pattern variable to a sequence of the values, preserving the order of properties in the input fragment. This sequence can only be used with double question mark in a template. Constraint-directed parsing applies to each property *value* individually.

- If a single question mark *pattern-keyword* does not find any matching property, then if a *default* is present, the pattern variable binds to the default expression, otherwise the property is required so the pattern match fails.
- If a double question mark *pattern-keyword* does not find any matching property, then if a *default* is present, the pattern variable binds to a sequence of one element, the default expression, otherwise the pattern variable binds to an empty sequence.
- Note: the default expression in a *pattern-keyword* is not evaluated during macro expansion; it is a parsed expression fragment that is used instead of a fragment from the macro call. The default is not subject to a pattern variable constraint.

Special Rules for Definitions

A list-style definition parses as the core reserved word `define`, an optional sequence of modifiers, a DEFINE-LIST-WORD, and a possibly-empty *list-fragment*. The left-hand side of a *list-style-definition-rule* matches this by treating the *definition-head* as a *pattern-sequence* and matching it to the sequence of modifiers, and then matching the *pattern* to the *list-fragment*. If no *definition-head* is present, the sequence of modifiers must be empty. If no *pattern* is present, the *list-fragment* must be empty. The word `define` and the DEFINE-LIST-WORD do not participate in the pattern match because they were already used to identify the macro being called and because the spelling of the DEFINE-LIST-WORD might have been changed by renaming the macro during module importing.

A body-style definition parses as the core reserved word `define`, an optional sequence of modifiers, a DEFINE-BODY-WORD, a possibly-empty *body-fragment*, the core reserved word `end`, and optional repetitions of the DEFINE-BODY-WORD and the NAME (if any) that is the first token of the *body-fragment*. The left-hand

side of a *body-style-definition-rule* matches this by treating the *definition-head* as a *pattern-sequence* and matching it to the sequence of modifiers, and then matching the *pattern* to the *body-fragment*. If no *definition-head* is present, the sequence of modifiers must be empty. If no *pattern* is present, the *body-fragment* must be empty. If the *body-fragment* ends in a semicolon, this semicolon is removed before matching. The optional semicolon in the rule is just decoration and does not participate in the pattern match. The word `define` and the DEFINE-BODY-WORD do not participate in the pattern match because they were already used to identify the macro being called and because the spelling of the DEFINE-BODY-WORD might have been changed by renaming the macro during module importing. The word `end` and the two optional items following it in the macro call are checked during parsing, and so do not participate in the pattern match.

It is an error for a *definition-head* to contain more than one wildcard.

Special Rules for Statements

A statement parses as a BEGIN-WORD, a possibly-empty *body-fragment*, the core reserved word `end`, and an optional repetition of the BEGIN-WORD. The left-hand side of a *statement-rule* matches this by matching the *pattern* to the *body-fragment*. If the rule does not contain a *pattern*, the *body-fragment* must be empty. If the *body-fragment* ends in a semicolon, this semicolon is removed before matching. The optional semicolon in the rule is just decoration and does not participate in the pattern match. The BEGIN-WORD does not participate in the pattern match because it was already used to identify the macro being called and because its spelling might have been changed by renaming the macro during module importing. The word `end` and the optional item following it in the macro call are checked during parsing, and so do not participate in the pattern match.

Special Rules for Function Macros

A call to a function macro parses as a FUNCTION-WORD followed by a parenthesized, possibly-empty *body-fragment*. The left-hand side of a *function-rule* matches this by matching the *pattern* to the *body-fragment*. If the rule does not contain a *pattern*, the *body-fragment* must be empty. The FUNCTION-WORD does not participate in the pattern match because it was already used to identify the macro being called and because its spelling might

have been changed by renaming the macro during module importing. The parentheses in the rule are just decoration and do not participate in the pattern match.

A function macro can also be invoked by any of the shorthand syntax constructs available for invoking functions. In this case, the arguments are always parsed expression fragments, as described on page 144. However, the left-hand side of a function-rule has to use function-macro-call syntax even if the macro is intended to be called by operator, slot reference, or element reference syntax.

Pattern Variable Constraints

Each *pattern-variable* in the left-hand side of a rule in a macro definition has a constraint associated with it. This prevents the pattern from matching unless the fragment matched to the pattern-variable satisfies the constraint. In most cases it also controls how the matching fragment is parsed.

You specify a constraint in a *pattern-variable* by suffixing a colon and the constraint name to the pattern variable name. Intervening whitespace is not allowed. As an abbreviation, if a pattern variable has the same name as its constraint, the *pattern-variable* can be written `?:`*the-name* instead of `?`*the-name*`:`*the-name*.

The following constraints are available:

Table 10-1 Available constraints

Constraint name	Grammar accepted	Binds pattern variable to
`expression`	*expression*	parsed expression fragment(1)
`variable`	*variable*	fragment(2)
`name`	NAME	one-token fragment
`token`	TOKEN	one-token fragment
`body`	$body_{opt}$ (3)	parsed expression fragment (4)

Table 10-1 Available constraints

Constraint name	Grammar accepted	Binds pattern variable to
`case-body`	*case-body*$_{opt}$ (3)	fragment(2)
`macro`	*macro*	fragment(5)
`*`	(wildcard)	fragment

Notes:

1. Parsed expression fragments are described on page 144
2. Where *expression, operand, constituents* or *body* appears in the grammar that this constraint accepts, the bound fragment contains a parsed expression fragment, not the original elementary fragments.
3. Parsing stops at an intermediate word.
4. The body is wrapped in `begin` … `end` to make it an expression, using the standard binding of `begin` in the Dylan module. An empty body defaults to `#f`.
5. A pattern-variable with a `macro` constraint accepts exactly one elementary fragment, which must be a macro call fragment. It binds the pattern variable to the expansion of the macro.

Some implementations and a future version of the Dylan language specification might add more constraint choices to this table.

When a pattern variable has the same name as an auxiliary rule-set, its constraint defaults to wildcard and can be omitted. Otherwise a constraint must be specified in every *pattern-variable* and *pattern-keyword*.

A constraint applies only to the specific pattern variable occurrence to which it is attached. It does not constrain other pattern variable occurrences with the same name.

Intermediate Words

When a *pattern-variable* has a constraint of `body` or `case-body`, its parsing of the fragment stops before any token that is an intermediate word. This allows intermediate words to delimit clauses that have separate bodies, like `else` and

`elseif` in an `if` statement. The intermediate words of a macro are identified as follows:

- Define a body-variable to be a pattern variable that either has a constraint of `body` or `case-body`, or names an auxiliary rule-set where some left-hand side in that rule-set ends in a body-variable. This is a least fixed point, so a recursive auxiliary rule-set does not automatically make its name into a body-variable! Note that an ellipsis that stands for a pattern variable is a body-variable when that pattern variable is one.
- Define an intermediate-variable to be a pattern variable that either immediately follows a body-variable in a left-hand side, or appears at the beginning of a left-hand side in an auxiliary rule-set named by an intermediate-variable.
- An intermediate word is a NAME that either immediately follows a body-variable in a left-hand side, or occurs at the beginning of a left-hand side in an auxiliary rule-set named by an intermediate-variable. Intermediate words are not reserved, they are just used as delimiters during the parsing for a *pattern-variable* with a `body` or `case-body` constraint.

Templates

Approximately speaking, a template has the same structure as what it constructs, but contains pattern variables that will be replaced by fragments extracted from the macro call. Thus a template in the *main-rule-set* looks like the macro expansion.

However, templates do not have a full grammar. A template is essentially any sequence of tokens and *substitutions* in which all of Dylan's brackets are balanced: `()`, `[]`, `{}`, `#()`, and `#[]`. Substitution for pattern variables produces a sequence of tokens and other elementary fragments.

Note that using unparsed token sequences as templates allows a macro expansion to contain macro calls without creating any inter-dependencies between macros. Since the template is not parsed at macro definition time, any macros called in the template do not have to be defined first, and macros can be compiled independently of each other. This simplifies the implementation at the minor cost of deferring some error checking from when a macro is defined until the time when the macro is called.

The grammar for templates is the definition of *template* in "Templates" on page 415.

All *template-elements* other than *substitution* are copied directly into the macro expansion. The various kinds of *substitution* insert something else into the macro expansion, as follows:

`?` NAME — The fragment bound to the pattern variable named NAME.

name-prefix$_{opt}$ `?` *name-string-or-symbol name-suffix*$_{opt}$ — The fragment bound to the pattern variable named *name-string-or-symbol*, converted to a STRING or SYMBOL and/or concatenated with a prefix and/or suffix. Note that this rule applies only when the first rule does not. The fragment must be a NAME. Concatenate the prefix, if any, the characters of the fragment, and the suffix, if any. The alphabetic case of the characters of the fragment is unspecified. Convert this to the same grammatical type (NAME, STRING, or SYMBOL) as *name-string-or-symbol*. When the result is a NAME, its hygiene context is the same as that of the fragment.

`??` NAME *separator*$_{opt}$ `...` — The sequence of fragments bound to the pattern variable named NAME, with *separator* inserted between each pair of fragments. The pattern variable must have been bound by a `??` *pattern-keyword*. *Separator* can be a binary operator, comma, or semicolon. If the size of the sequence is 1 or *separator* is omitted, no separator is inserted. If the sequence is empty, nothing is inserted.

`...` — The fragment bound to the pattern variable that names this rule set; this is only valid in an auxiliary rule set.

`?=` NAME — A reference to NAME, in the lexical context where the macro was called.

It is an error for a single question-mark *substitution* to use a pattern variable that was bound by a double question-mark *pattern-keyword*.

It is an error for a double question-mark *substitution* to use a pattern variable that was bound by a single question-mark *pattern-variable* or *pattern-keyword*.

It is an error for a *substitution* to use a pattern variable that does not appear on the left-hand side of the same rule.

When a template contains a *separator* immediately followed by a *substitution*, and the fragment inserted into the macro expansion by the *substitution* is empty, the separator is removed from the macro expansion.

Auxiliary Rule Sets

Auxiliary rule sets are like subroutines for rewrite rules. An auxiliary rule set rewrites the value of a pattern variable after it is bound by a pattern and before it is substituted into a template. Auxiliary rule sets only come into play after a pattern has matched; the failure of all patterns in an auxiliary rule set to match causes the entire macro call to be declared invalid, rather than back-tracking and trying the next pattern in the calling rule set.

See the definition of *aux-rule-sets* in "Auxiliary Rule Sets" on page 416.

A SYMBOL flags the beginning of an auxiliary rule set. For readability it is generally written as `name:` rather than `#"name"`. The name of the symbol is the same as the name of the pattern variable that is rewritten by this auxiliary rule set. All occurrences of this pattern variable in all rule sets are rewritten. A pattern variable can occur in the very auxiliary rule set that rewrites that pattern variable; this is how you write recursive rewrite rules, which greatly expand the power of pattern-matching.

When an auxiliary rule set's pattern variable occurs in a double question-mark *pattern-keyword*, the auxiliary rule set rewrites each property value in the sequence individually.

The order of auxiliary rule sets in a macro definition is immaterial.

The ellipsis `...` in patterns and templates of an auxiliary rule set means exactly the same thing as the pattern variable that is rewritten by this auxiliary rule set. Using ellipsis instead of the pattern variable can make recursive rewrite rules more readable.

Hygiene

Dylan macros are always **hygienic**. The basic idea is that each named value reference in a macro expansion means the same thing as it meant at the place in

the original source code from which it was copied into the macro expansion. This is true whether that place was in the macro definition or in the macro call. Because a macro expansion can include macro calls that need further expansion, named value references in one final expansion can come from several different macro definitions and can come from several different macro calls, either to different macros or—in the case of recursion—distinct calls to the same macro.

(Sometimes the property that variable references copied from a macro call mean the same thing in the expansion is called "hygiene" and the property that variable references copied from a macro definition mean the same thing in the expansion is called "referential transparency." We include both properties in the term "hygiene.")

Specifically, a macro can bind temporary variables in its expansion without the risk of accidentally capturing references in the macro call to another binding with the same name. Furthermore, a macro can reference module bindings in its expansion without the risk of those references accidentally being captured by bindings of other variables with the same name that surround the macro call. A macro can reference module bindings in its expansion without worrying that the intended bindings might have different names in a module where the macro is called.

One way to implement this is for each *template-element* that is a NAME, UNARY-OPERATOR, or BINARY-OPERATOR to be replaced in the macro expansion by a special token which plays the same grammatical role as the NAME, UNARY-OPERATOR, or BINARY-OPERATOR but remembers three pieces of information:

- The original NAME. For an operator, this is the name listed in Table 4-1 on page 36.
- The lexical context where the macro was defined, which is just a module since macro definitions are only allowed at top level, not inside of bindings.
- The specific macro call occurrence. This could be an integer that is incremented each time a macro expansion occurs.

In general one cannot know until all macros are expanded whether a NAME is a bound variable reference, a module binding reference, a variable that is being bound, or something that is not a binding name at all, such as a definition *modifier* or an intermediate word. Similarly, one cannot know until all macros are expanded whether a UNARY-OPERATOR or BINARY-OPERATOR refers to a local binding or a module binding. Thus the information for each of those cases is

retained in the special token. A named value reference and a binding connect if and only if the original NAMES and the specific macro call occurrences are both the same. (In that case, the lexical contexts will also be the same, but this need not be checked.) A named value reference and a binding never connect if one originated in a template and the other originated in a macro call.

References in a macro expansion to `element` or `aref` created by using element reference syntax must receive similar treatment so the NAME `element` or `aref` gets looked up in the environment of the macro definition, not the environment of the macro call.

For purposes of hygiene, a *pattern-keyword default* is treated like part of a template, even though it is actually part of a pattern.

The mapping from getters to setters done by the `:=` operator is hygienic. In all cases the setter name is looked up in the same lexical context and macro call occurrence as the getter name.

Intentional Hygiene Violation

Sometimes it is necessary for a macro to violate the hygienic property, for example to include in a macro expansion a named value reference to be executed in the lexical context where the macro was called, not the lexical context where the macro was defined. Another example is creating a local binding in a macro expansion that will be visible to the body of the macro. This feature should be used sparingly, as it can be confusing to users of the macro, but sometimes it is indispensable.

The construct `?=` NAME in a template inserts into the expansion a reference to NAME, in the lexical context where the macro was called. It is as if NAME came from the macro call rather than from the template.

Hygiene Versus Module Encapsulation

A named value reference in a macro expansion that was produced by a *template-element* that is a NAME, UNARY-OPERATOR, BINARY-OPERATOR, or `[` $template_{opt}$ `]` and that does not refer to a local binding created by the macro expansion must have the same meaning as would a named value reference with the same name adjacent to the macro definition. This is true even if the

macro call is in a different module or a different library from the one in which the macro definition occurs, even if the binding is not exported.

This allows exported macros to make use of private bindings without requiring these bindings to be exported for general use. The module that calls the macro does not need to import the private bindings used by the expansion.

If one of the following template-element sequences appears in the right-hand side of a rewrite rule, it may introduce named value references to the indicated name in an expansion of the macro. If such a named value reference does not refer to a local binding created by the macro expansion then it must have the same meaning as would a named value reference with the same name adjacent to the macro definition.

- *variable-name*

  Reference to *variable-name*

- *getter-name* `(` *template*$_{opt}$ `)` *assignment-operator*
  assignment-operator-name `(` *template*$_{opt}$ *getter-name* `(` *template*$_{opt}$ `)` `)`

  Reference to *getter-name* `## "-setter"`

- `[` *template*$_{opt}$ `]` *assignment-operator*
  assignment-operator-name `(` *template*$_{opt}$ `[` *template*$_{opt}$ `]` `)`

  Reference to either `element-setter` and `aref-setter`

- `.` *getter-name assignment-operator*
  assignment-operator-name `(` *template*$_{opt}$ `.` *getter-name* `)`

  Reference to *getter-name* `## "-setter"`

- `define` *definition-head*$_{opt}$ *definer-name*

  Reference to *definer-name* `## "-definer"`

Items in the preceding template-element sequences have the following meanings:

- *assignment-operator-name* is a NAME, and the value of the binding with that name (in the module containing the macro definition in which the template occurs) is the assignment macro which is the value of the binding named `\:=` exported by the Dylan module of the Dylan library.

- *assignment-operator* is a binary-operator whose associated binding is an *assignment-operator-name*.)
- *definer-name* is a DEFINE-BODY-WORD or DEFINE-LIST-WORD.
- *variable-name* and *getter-name* are NAMEs.
- *template* is defined in Appendix A, "BNF," on page 415.
- *definition-head* is defined in Appendix A, "BNF," on page 413.
- The notation *foo* `## "string"` indicates a new token composed of the text of *foo* concatenated with the string.

Note that these template-element sequences can overlap in a template. For example `{ foo (bar) := }` is a potential reference to `foo`, to `foo-setter`, and to `bar`.

Implementations must use some automatic mechanism for noting the bindings associated with the named value references in macro expansions produced by the template-element sequences described above, and must make such bindings available to any library where the macro is accessible. In general, the set of bindings that must be made available to other libraries cannot be computed precisely because the right-hand sides of rewrite rules are not fully parsed until after a macro is called and expanded, making it impossible to determine whether an occurance of one of the described sequences of template elements will actually produce a named variable reference in the expansion. However, an upper bound on this set of bindings can be computed by assuming that all occurances of the described template-element sequences might introduce the indicated named value reference if there is a binding for that name accessible from the module in which the macro definition appears.

Rewrite Rule Examples

The following definitions of all of the built-in macros are provided as examples. This section is not intended to be a tutorial on how to write macros, just a collection of demonstrations of some of the tricks.

The built-in macros cannot really be implemented this way, for example, `if` and `case` cannot really both be implemented by expanding to the other. Certain built-in macros cannot be implemented with rewrite rules or necessarily rewrite into implementation-dependent code; in these cases the right-hand sides are shown as *id*.

Statement Macros

Begin

```
define macro begin
  { begin ?:body end } => { ?body }
end;
```

Block

```
define macro block
  { block () ?ebody end }
   => { ?ebody }
  { block (?:name) ?ebody end }
   => { with-exit(method(?name) ?ebody end) }

 // Left-recursive so leftmost clause is innermost
 ebody:
  { ... exception (?type:expression, ?eoptions) ?:body }
   => { with-handler(method() ... end,
                     method(ignore) ?body end,
                     ?type, ?eoptions) }
  { ... exception (?:name :: ?type:expression, ?eoptions) ?:body }
   => { with-handler(method() ... end,
                     method(?name) ?body end,
                     ?type, ?eoptions) }
  { ?abody cleanup ?cleanup:body}
   => { with-cleanup(method() ?abody end, method () ?cleanup end) }
  { ?abody }
   => { ?abody }

 abody:
  { ?main:body }
   => { ?main }
  { ?main:body afterwards ?after:body }
   => { with-afterwards(method() ?main end, method () ?after end) }
```

```
 eoptions:
  { #rest ?options:expression,
    #key ?test:expression = always(#t),
    ?init-arguments:expression = #() }
   => { ?options }
end;
```

Case

```
define macro case
  { case ?:case-body end }             => { ?case-body }
 case-body:
  { }                                  => { #f }
  { otherwise ?:body }                 => { ?body }
  { ?test:expression => ?:body; ... } => { if (?test) ?body
                                           else ... end if }
end;
```

For

```
// This macro has three auxiliary macros, whose definitions follow
define macro for
 { for (?header) ?fbody end }       => { for-aux ?fbody, ?header end }

 // pass main body and finally body as two expressions
 fbody:
  { ?main:body }                     => { ?main, #f }
  { ?main:body finally ?val:body }   => { ?main, ?val }

 // convert iteration clauses to property list via for-clause macro
 header:
  { ?v:variable in ?c:expression, ... }
   => { for-clause(?v in ?c) ... }
  { ?v:variable = ?e1:expression then ?e2:expression, ... }
   => { for-clause(?v = ?e1 then ?e2) ... }
  { ?v:variable from ?e1:expression ?to, ... }
```

```
   => { for-clause(?v from ?e1 ?to) ... }
  { }                                 => { }
  { #key ?while:expression }          => { for-clause(~?while stop) }
  { #key ?until:expression }          => { for-clause(?until stop) }

 // parse the various forms of numeric iteration clause
 to:
  { to ?limit:expression by ?step:expression }
                                      => { hard ?limit ?step }
  { to ?limit:expression }            => { easy ?limit 1   > }
  { above ?limit:expression ?by }     => { easy ?limit ?by <= }
  { below ?limit:expression ?by }     => { easy ?limit ?by >= }
  { ?by }                             => { loop ?by }

 by:
  { }                                 => { 1 }
  { by ?step:expression }             => { ?step }
end;

// Auxiliary macro to make the property list for an iteration clause.
// Each iteration clause is a separate call to this macro so the
// hygiene rules will keep the temporary variables for each clause
// distinct.
// The properties are:
//  init0: - constituents for start of body, outside the loop
//  var1:  - a variable to bind on each iteration
//  init1: - initial value for that variable
//  next1: - value for that variable on iterations after the first
//  stop1: - test expression, stop if true, after binding var1's
//  var2:  - a variable to bind on each iteration, after stop1 tests
//  next2: - value for that variable on every iteration
//  stop2: - test expression, stop if true, after binding var2's
define macro for-clause

  // while:/until: clause
  { for-clause(?e:expression stop) }
   => { , stop2: ?e }
```

```
// Explicit step clause
{ for-clause(?v:variable = ?e1:expression then ?e2:expression) }
 => { , var1: ?v, init1: ?e1, next1: ?e2 }

// Collection clause
{ for-clause(?v:variable in ?c:expression) }
 => { , init0: [ let collection = ?c;
                 let (initial-state, limit,
                      next-state, finished-state?,
                      current-key, current-element)
                     = forward-iteration-protocol(collection); ]
      , var1: state, init1: initial-state
      , next1: next-state(collection, state)
      , stop1: finished-state?(collection, state, limit)
     , var2: ?v, next2: current-element(collection, state) }

// Numeric clause (three cases depending on ?to right-hand side)
{ for-clause(?v:name :: ?t:expression from ?e1:expression
             loop ?by:expression) }
 => { , init0: [ let init = ?e1;
                 let by = ?by; ]
      , var1: ?v :: ?t, init1: init, next1: ?v + by }

{ for-clause(?v:name :: ?t:expression from ?e1:expression
             easy ?limit:expression ?by:expression ?test:token) }
 => { , init0: [ let init = ?e1;
                 let limit = ?limit;
                 let by = ?by; ]
      , var1: ?v :: ?t, init1: init, next1: ?v + by
      , stop1: ?v ?test limit }

{ for-clause(?v:name :: ?t:expression from ?e1:expression
             hard ?limit:expression ?by:expression) }
 => { , init0: [ let init = ?e1;
                 let limit = ?limit;
                 let by = ?by; ]
```

```
        , var1: ?v :: ?t, init1: init, next1: ?v + by
        , stop1: if (by >= 0) ?v > limit else ?v < limit end if }
end;

// Auxiliary macro to expand multiple for-clause macros and
// concatenate their expansions into a single property list.
define macro for-aux
  { for-aux ?main:expression, ?value:expression, ?clauses:* end }
   => { for-aux2 ?main, ?value ?clauses end }

 clauses:
  { } => { }
  { ?clause:macro ... } => { ?clause ... }
end;

// Auxiliary macro to assemble collected stuff into a loop.
// Tricky points:
// loop iterates by tail-calling itself.
// return puts the finally clause into the correct lexical scope.
// ??init0 needs an auxiliary rule set to strip off the shielding
// brackets that make it possible to stash local declarations in
// a property list.
// ??var2 and ??next2 need a default because let doesn't allow
// an empty variable list.
// ??stop1 and ??stop2 need a default because if () is invalid.
define macro for-aux2
  { for-aux2 ?main:expression, ?value:expression,
             #key ??init0:*, ??var1:variable,
                  ??init1:expression, ??next1:expression,
                  ??stop1:expression = #f,
                  ??var2:variable = x, ??next2:expression = 0,
                  ??stop2:expression = #f
    end }
   => { ??init0 ...
        local method loop(??var1, ...)
                let return = method() ?value end method;
                if (??stop1 | ...) return()
                else let (??var2, ...) = values(??next2, ...);
```

```
                    if(??stop2 | ...) return()
                    else ?main; loop(??next1, ...)
                    end if;
                end if;
            end method;
        loop(??init1, ...) }

 // strip off brackets used only for grouping
 init0:
  { [ ?stuff:* ] } => { ?stuff }
end;
```

If

```
define macro if
  { if (?test:expression) ?:body ?elses end }
                                    => { case ?test => ?body;
                                         otherwise ?elses end }
 elses:
  { }                               => { #f }
  { else ?:body }                   => { ?body }
  { elseif (?test:expression) ?:body ... }
                                    => { case ?test => ?body;
                                         otherwise ... end }
end;
```

Method

```
define macro method
  { method (?parameters:*) => (?results:*) ; ?:body end }     => id
  { method (?parameters:*) => (?results:*) ?:body end }       => id
  { method (?parameters:*) => ?result:variable ; ?:body end } => id
  { method (?parameters:*) ; ?:body end }                     => id
  { method (?parameters:*) ?:body end }                       => id
end;
```

Select

```
define macro select
  { select (?what) ?:case-body end } => { ?what; ?case-body }

 what:
  { ?object:expression by ?compare:expression }
                                    =>  { let object = ?object;
                                          let compare = ?compare }
  { ?object:expression }            =>  { let object = ?object;
                                          let compare = \== }

 case-body:
  { }
   => { error("select error, %= doesn't match any key", object) }
  { otherwise ?:body }              => { ?body }
  { ?keys => ?:body; ... }          => { if (?keys) ?body
                                         else ... end if }

 keys:
  { ?key:expression }               => { compare(object, ?key) }
  { (?keys2) }                      => { ?keys2 }
  { ?keys2 }                        => { ?keys2 }

 keys2:
  { ?key:expression }               => { compare(object, ?key) }
  { ?key:expression, ... }          => { compare(object, ?key) | ... }
end;
```

Unless

```
define macro unless
  { unless (?test:expression) ?:body end }
   => { if (~ ?test) ?body end }
end;
```

Until

```
define macro until
  { until (?test:expression) ?:body end }
   => { local method loop ()
                if (~ ?test)
                  ?body;
                  loop()
                end if;
              end method;
        loop() }
end;
```

While

```
define macro while
  { while (?test:expression) ?:body end }
   => { local method loop ()
                if (?test)
                  ?body;
                  loop()
                end if;
              end method;
        loop() }
end;
```

Definition Macros

Define Class

```
define macro class-definer
  { define ?mods:* class ?:name (?supers) ?slots end }  =>  id
```

```
 supers:
  { }                                                    =>  id
  { ?super:expression, ... }                             =>  id

 slots:
  { }                                                    =>  id
  { inherited slot ?:name, #rest ?options:*; ... }       =>  id
  { inherited slot ?:name = ?init:expression,
    #rest ?options:*; ... }                              =>  id
  { ?mods:* slot ?:name, #rest ?options:*; ... }         =>  id
  { ?mods:* slot ?:name = ?init:expression,
    #rest ?options:*; ... }                              =>  id
  { ?mods:* slot ?:name :: ?type:expression,
    #rest ?options:*; ... }                              =>  id
  { ?mods:* slot ?:name :: ?type:expression = ?init:expression,
    #rest ?options:*; ... }                              =>  id
  { required keyword ?key:expression,
    #rest ?options:*; ... }                              =>  id
  { required keyword ?key:expression ?equals:token ?init:expression,
    #rest ?options:*; ... }                              =>  id
  { keyword ?key:expression, #rest ?options:*; ... }     =>  id
  { keyword ?key:expression ?equals:token ?init:expression,
    #rest ?options:*; ... }                              =>  id
end;
```

Define Constant

```
define macro constant-definer
  { define ?modifiers:* constant
    ?:name :: ?type:expression = ?init:expression }      =>  id
  { define ?modifiers:* constant
    (?variables:*) ?equals:token ?init:expression }      =>  id
end;
```

Define Domain

```
define macro domain-definer
  { define sealed domain ?:name ( ?types ) }            =>  id

 types:
  { } => { }
  { ?type:expression, ... } => { ?type, ... }
end;
```

Define Generic

```
define macro generic-definer
  { define ?mods:* generic ?:name ?rest:* }             =>  id

 rest:
  { ( ?parameters:* ), #key }                           =>  id
  { ( ?parameters:* ) => ?:variable, #key }             =>  id
  { ( ?parameters:* ) => (?variables:*), #key }         =>  id
end;
```

Define Library

```
define macro library-definer
  { define library ?:name ?items end }                  =>  id

 items:
  { }                                                   =>  id
  { use ?:name, #rest ?options:*; ... }                 =>  id
  { export ?names; ... }                                =>  id

 names:
  { ?:name }                                            =>  id
  { ?:name, ... }                                       =>  id
end;
```

Define Method

```
define macro method-definer
  { define ?mods:* method ?:name ?rest end }            =>  id
 rest:
  { (?parameters:*) => (?results:*) ; ?:body }          =>  id
  { (?parameters:*) => (?results:*) ?:body }            =>  id
  { (?parameters:*) => ?result:variable ; ?:body }      =>  id
  { (?parameters:*) ; ?:body }                          =>  id
  { (?parameters:*) ?:body }                            =>  id
end;
```

Define Module

```
define macro module-definer
  { define module ?:name ?items end }                   =>  id

 items:
  { }                                                   =>  id
  { use ?:name, #rest ?options:*; ... }                 =>  id
  { export ?names; ... }                                =>  id
  { create ?names; ... }                                =>  id

 names:
  { ?:name }                                            =>  id
  { ?:name, ... }                                       =>  id
end;
```

Define Variable

```
define macro variable-definer
  { define ?modifiers:* variable
    ?:name :: ?type:expression = ?init:expression }     =>  id
  { define ?modifiers:* variable
    (?variables:*) ?equals:token ?init:expression }     =>  id
end;
```

Operator Function Macros

&

```
define macro \&
  { \&(?first:expression, ?second:expression) }
   => { if (?first) ?second else #f end }
end;
```

|

```
define macro \|
  { \|(?first:expression, ?second:expression) }
   => { let temp = ?first;
        if (temp) temp else ?second end }
end;
```

:=

```
define macro \:=
  { \:=(?place:macro, ?value:expression) }              => id
  { \:=(?place:expression, ?value:expression) }         => id
end;
```

Additional Examples

The following macros are not built-in, but are simply supplied as examples. Each is shown as a definition followed by a sample call.

Test and Test-setter

```
define macro test
  { test(?object:expression) } =>
                                { frame-slot-getter(?object, #"test") }
end macro;

define macro test-setter
  { test-setter(?value:expression, ?object:expression) }
   => { frame-slot-setter(?value, ?object, #"test") }
end macro;

test(foo.bar) := foo.baz;
```

Transform!

```
define macro transform!
 // base case
 { transform!(?xform:expression) } => { ?xform }
 // the main recursive rule
 { transform!(?xform:expression, ?x:expression, ?y:expression,
              ?more:*) }
  => { let xform = ?xform;
       let (nx, ny) = transform(xform, ?x, ?y);
       ?x := nx; ?y := ny;
       transform!(xform, ?more) }
end macro;

transform!(w.transformation, xvar, yvar, w.pos.x, w.pos.y);
```

Formatting-table

```
define macro formatting-table
  { formatting-table (?:expression,
                      #rest ?options:expression,
                      #key ?x-spacing:expression = 0,
```

```
                          ?y-spacing:expression = 0)
      ?:body end }
   => { do-formatting-table(?expression, method() ?body end,
                            ?options) }
end macro;

formatting-table (stream, x-spacing: 10, y-spacing: 12)
  foobar(stream)
end;
```

With-input-context

```
define macro with-input-context
  { with-input-context (?context-type:expression,
                        #key ?override:expression = #f)
     ?bbody end }
   => { do-with-input-context(?context-type, ?bbody,
                              override: ?override) }

 bbody:
  { ?:body ?clauses }  => { list(?clauses), method() ?body end }

 clauses:
  { }                  => { }
  { on (?:name :: ?spec:expression, ?type:variable) ?:body ... }
   => { pair(?spec, method (?name :: ?spec, ?type) ?body end),
        ... }
end macro;

with-input-context (context-type, override: #t)
      // the body that reads from the user
      read-command-or-form (stream);
    // the clauses that dispatch on the type
    on (object :: <command>, type) execute-command (object);
    on (object :: <form>, type) evaluate-form (object, type);
end;
```

Define Command

```
define macro command-definer
 { define command ?:name (?arguments:*) (#rest ?options:expression)
     ?:body end }
  => { define-command-1 ?name (?arguments) ?body end;
       define-command-2 ?name (?arguments) (?options) end }
end macro;

// define the method that implements a command
// throws away the "stuff" in each argument used by the command parser
define macro define-command-1
 { define-command-1 ?:name (?arguments) ?:body end }
  => { define method ?name (?arguments) ?body end }

 // map over ?arguments, reducing each to a parameter-list entry
 // but when we get to the first argument that has a default, put
 // in #key and switch to the key-arguments loop
 arguments:
  { } => { }
  { ?:variable = ?default:expression ?stuff:*, ?key-arguments }
   => { #key ?variable = ?default, ?key-arguments }
  { ?argument, ... } => { ?argument, ... }

 // map over keyword arguments the same way, each must
 // have a default
 key-arguments:
  { } => { }
  { ?key-argument, ... } => { ?key-argument, ... }

 // reduce one required argument spec to a parameter-list entry
 argument:
  { ?:variable ?stuff:* } => { ?variable }
```

```
 // reduce one keyword argument spec to a parameter-list entry
 key-argument:
  { ?:variable = ?default:expression ?stuff:* }
   => { ?variable = ?default }
end macro;

// generate the datum that describes a command and install it
define macro define-command-2
 { define-command-2 ?:name (?arguments) (#rest ?options:*) end }
  => { install-command(?name, list(?arguments), ?options) }

 // map over ?arguments, reducing each to a data structure
 arguments:
  { } => { }
  { ?argument, ... } => { ?argument, ... }

 // reduce one argument specification to a data structure
 argument:
  { ?:name :: ?type:expression = ?default:expression ?details }
   => { make(<argument-info>, name: ?"name", type: ?type,
             default: ?default, ?details) }
  { ?:name :: ?type:expression ?details }
   => { make(<argument-info>, name: ?"name", type: ?type, ?details) }

 // translate argument specification to <argument-info> init keywords
 details:
  { } => { }
  { ?key:name ?value:expression ... } => { ?#"key" ?value, ... }
end macro;

define command com-show-home-directory
       (directory :: <type> provide-default #t,
        before :: <time> = #() prompt "date",
        after  :: <time> = #() prompt "date")
       // Options
       (command-table: directories,
```

```
        name: "Show Home Directory")
    body()
end command com-show-home-directory;
```

Get-resource

```
// The idea is that in this application each library has its own
// variable named $library, which is accessible to modules in that
// library.  Get-resource gets a resource associated with the library
// containing the call to it. Get-resource-from-library is a function.
// The get-resource macro is a device to make programs more concise.
define macro get-resource
  { get-resource(?type:expression, ?id:expression) }
   => { get-resource-from-library(?=$library, ?type, ?id) }
end macro;

show-icon(get-resource(ResType("ICON"), 1044));
```

Completing-from-suggestions

```
// The completing-from-suggestions macro defines a lexically visible
// helper function called "suggest", which is only meaningful inside
// of calls to the completer. The "suggest" function is passed as an
// argument to the method passed to complete-input; alternatively it
// could have been defined in a local declaration wrapped around the
// method.
define macro completing-from-suggestions
  { completing-from-suggestions (?stream:expression,
                                 #rest ?options:expression)
      ?:body end }
   =>{ complete-input(?stream,
                      method (?=suggest) ?body end,
                      ?options) }
end macro;
```

```
completing-from-suggestions (stream, partial-completers: #(' ', '-'))
  for (command in commands)
    suggest (command, command-name (command))
  end for;
end completing-from-suggestions;
```

Define Jump-instruction

```
define macro jump-instruction-definer
  { define jump-instruction ?:name ?options:* end }
   => { register-instruction("j" ## ?#"name",
                             make(<instruction>,
                                  debug-name: "j" ## ?"name",
                                  ?options)) }
end macro;

define jump-instruction eq cr-bit: 2, commutative?: #t end;
```

Macros