Dylan Macros

Version 1.2 of 20 April 1993

David A. Moon
Moon@Cambridge.Apple.com
Apple Computer Eastern Research & Technology

This paper shows how to add macros to Dylan that are hygienic (no name clashes or unintentional violations of referential transparency), general (arbitrary computations can occur during macro expansion, macros can expand into any expression of the language), and efficient (the Dylan run time model is not affected by the existence of macros). This design is based on the Scheme macros of Dybvig and Hieb, however many details have been changed to eliminate dependencies on a particular surface syntax and to make the design simpler and more object-oriented.

This paper covers only the language design, not the implementation.

***** DISCLAIMER *****
This is an early draft, it is not an official part of the Dylan language specification. Currently this is only for the classic syntax. The algebraic syntax will be put in soon.

Major Concepts

There are three inter-related major concepts:

Program Stages—a program is divided into "stages" that execute at different places and times. The stages that concern us here are the compile stage, when macros are expanded, and the run stage, when ordinary execution occurs. Since macros can themselves use macros, there is actually an indefinitely tall tower of compile stages on top of the run stage.

The `<expression>` Class—macros manipulate program fragments, which are represented as instances of `<expression>`. These are abstract expressions, completely free of surface syntax. An instance of `<expression>` encodes not only the abstract syntax tree underlying an expression that was written in some surface syntax, but also information about the lexical context and possibly additional implementation-dependent information such as source code location. Subsidiary concepts are two subclasses of `<expression>`, patterns for taking expressions apart, and templates for putting expressions together. These features exist only in the compile stage.

Macros—a macro is an operator that transforms the expression that calls it into another expression, which is then executed in place of the original expression. This is called "macro expansion." Macro expansion takes place in the compile stage. The input and output of a macro transformation are each an instance of `<expression>`.

Expressions

The definition of “expression” on pages 27-30 of the Dylan book needs some refinement for macros. The following is a sketch of the refinements but not a complete replacement for the existing text.

There are three kinds of expressions: literal constants, variable names, and combinations. A combination has an operator and zero or more operands. The operator and operands are expressions. Combinations are divided into three sub-kinds—function calls, macro calls, and special forms—according to whether the value of their operator expression is a function, a macro, or a special form operator. A combination whose operator evaluates to an object unknown during the compile stage is treated as a function call and an error is signalled when the combination is executed if the object does not turn out to be an instance of`<function>`.

Macros

A macro is an “ectoplasmic” object: it is an intangible object but it does have an effect on the program’s behavior. It is debatable whether a macro object is “real” or “imaginary .”

Macros can only be used in very restricted ways: they can be created, given names, and called. In more detail:

• Macros are created by the `macro` special form:

Classic syntax: (macro*pattern-clauses* ...) [Compile Stage Special Form]
Algebraic syntax: *tbd*

The *pattern-clauses* (defined under `expression-case` below) are matched against the macro call expression object, the body expressions in the first matching clause are executed, and the value of the last body expression is a replacement expression object. Everything inside the *pattern-clauses* is in the compile stage.

• Macros are given names the same way as anything else, using the`define` and `bind` special forms. The initial-value expression in a `define` or `bind` can evaluate to a macro. This can only happen when the initial-value expression, after macro expansion, is a`macro` special form or a variable name that is bound to a macro.

It is forbidden to `set!` a variable name that is bound to a macro, because that could make it uncertain whether a reference to the variable name elsewhere in the program refers to that macro, or to another macro, or to a normal object, which would make compilation impossible.

A module variable that names a macro can be exported from a module, imported into another module, and renamed during import, all in the ordinary way .

• A macro call is an expression that is a combination whose operator expression would evaluate to a macro. This happens if and only if, after macro expansion, the operator expression is a `macro` special form or a variable name that is bound to a macro. A macro call is expanded at compile time by passing the macro call, represented as an instance of `<expression>`, to the macro. The last body expression in the first matching pattern clause of the macro returns an instance of `<expression>` which represents an expression that the compiler processes in place of the macro call.

A macro object cannot be an argument to a function or a result returned by a function. A `macro` special form or a variable name that is bound to a macro can be an operand in a macro call expression, provided the expansion obeys these restrictions.

It's easy to show, given the above restrictions, that all uses of macros can be found at compile time and expanded out before any execution occurs, assuming that it is known at compile time exactly which module variables name macros. Thus although macros are conceptually objects, they have no existence during execution and take up no space.

Expression Objects

`<expression>` [Compile Stage Abstract Class]

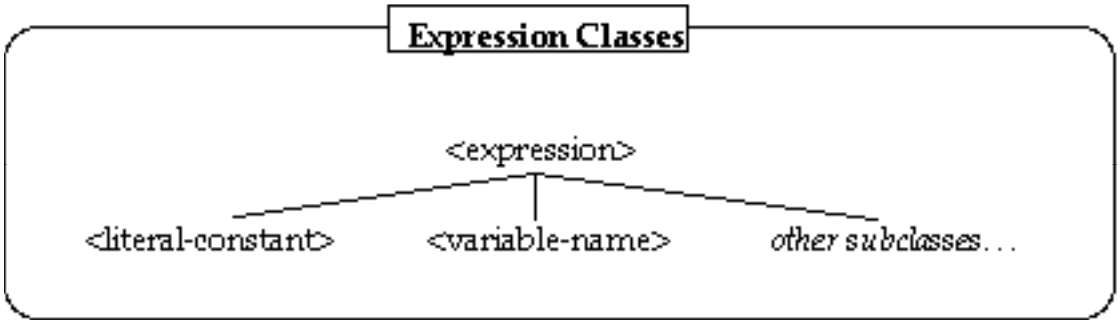
An expression object, an instance of the class `<expression>`, represents a Dylan expression in a particular lexical context. The operations provided for expression objects are independent of surface syntax.

Note that "lexical context" does not mean "lexical scope" or "lexical environment." Lexical variable bindings can only be determined after macro expansion, so expression objects in general do not encode lexical scopes. The lexical context of an expression indicates when and where that expression was introduced into the program. Binding a variable name that came from one lexical context does not affect references to a similar variable name that came from a different lexical context. A free reference to a variable name uses its lexical context to help find the proper lexical environment.

The purpose of these lexical contexts is to distinguish variable names inserted into the program by macros from variable names present in the original text, so that the meaning of each variable name can be determined in the appropriate lexical

environment after macro expansion is complete, and so that no unwanted name clashes between macros and their callers will occur. This is the so-called “hygienic” property.

There is one “original” lexical context used for all expressions that were not created by macro expansion. A new lexical context is created each time an `expression` special form is evaluated. Thus when a macro inserts into its expansion a reference to a variable that was not in the macro call, it will be in this new lexical context. This is discussed in more detail later.



An expression that is a literal constant is represented by an instance of `<literal-constant>`, a subclass of `<expression>`. It is *not* represented by the constant object! Thus the expression `10` is represented by an instance of `<literal-constant>`, not by an instance of `<integer>`. The meaning of a literal constant is not affected by its lexical context.

An expression that is a variable name is represented by an instance of the class `<variable-name>`, which is a subclass of `<expression>`. Note well that the expression object for a variable name is not the symbol that names the variable.

No other subclasses of `<expression>` are standardized, although some might exist in some implementations.

These classes and all methods on them only exist in the compile stage. Thus expression objects and the programs that manipulate them do not exist at run time.

The classes `<expression>` and `<sequence>` are disjoint. The proper functioning of templates depends on this.

The `id?` function is not very useful on expression objects, because taking the same expression apart multiple times doesn’t necessarily produce `id?` objects for the sub-expressions. In other words, `id?` tests object identity of expression objects, but does not depend on anything about the expressions represented by those objects.

The = function is not very useful on expression objects in general, however it has useful methods for <literal-constant> and <variable-name>. For general expressions, it is simply the same as id?.

The < function is not defined on expression objects.

Expression objects are opaque; there is no function analogous to Dybvig and Hieb's syntax-object->datum function that exposes an S-expression representation.

There are four categories of operations on expression objects: operations on literal constants, operations on variable names, operations for taking expressions apart, and operations for putting expressions together. I'll explain the first two categories first, since they are simpler, even though they are much less frequently used.

Operations on Literal Constants

<literal-constant> [Compile Stage Instantiable Class]

An expression that is a literal constant is represented by an instance of <literal-constant>, a subclass of <expression>.

(make <literal-constant> object: *object*) creates an expression object that represents a literal constant that evaluates to a value at the next program stage down that is similar to *object*. "Similar" is defined below . The keyword object: is required. *Object* must be an object that is valid for use in literal constants.

At present there is no function that retrieves *object* from a literal constant.

The concept of "literal constant similarity" provides a way to compare two objects that are in different program stages and thus cannot be brought together and compared with a function such as id? or =. Similarity is only defined for the objects that are valid for use in literal constants. No one has specifically defined yet what those are in Dylan, so I will take a guess:

Two objects are similar if and only if one of the following statements is true:

- They are both rational numbers and they have the same value.
- They are both floating point numbers of the same format and they have the same value.
- They are both instances of <character> and they represent the same character
- They are both #f.
- They are both #t.
- They are both the same parameter-list token (#key, #rest, or #next).
- They are both keywords and their names are the same when alphabetic case is disregarded.

- They are both symbols and their names are the same when alphabetic case is disregarded.
- They are both instances of `<string>`, their sizes are the same, and their corresponding elements are similar.
- They are both instances of `<simple-object-vector>`, their sizes are the same, and their corresponding elements are similar.
- They are both instances of `<pair>`, their heads are similar, and their tails are similar.
- They are both the empty list.
- They are both the ellipsis object (see `$pattern-ellipsis` below).

I assume issue “Remove Complex Numbers” will pass, so there is no rule for similarity of non-real complex numbers.

Note that because all compile stages share the same objects, an instance of `<literal-constant>` inserted into a compile stage expression will evaluate to the same object (tested with `id?`) that was used to make the instance of `<literal-constant>` and therefore this object and its elements must not be modified (if it’s a string, simple-object-vector, or pair.) In contrast, except in implementations that unify the compile and run stages, an instance of `<literal-constant>` inserted into a run stage expression will evaluate to an object that is distinct from any compile stage object, including the one that was used to make the instance of `<literal-constant>`.

`= literal-constant-1 literal-constant-2 => boolean` [Compile Stage G.F. Method]

This returns true if the objects inside the two literal constants are similar, as defined above.

`$pattern-ellipsis` [Compile Stage Constant]

The value of `$pattern-ellipsis` is an instance of `<literal-constant>` that represents the ... notation in patterns and templates. This object can be useful in macro-defining macros that need to construct patterns and templates. The result of evaluating that literal constant is unspecified. That literal constant can only appear in compile stage expressions.

Operations on Variable Names

`<variable-name>` [Compile Stage Instantiable Class]

An expression that is a variable name is represented by an instance of the class `<variable-name>`, which is a subclass of `<expression>`.

The following functions are useful in macros that need to do special things with

variable names.

`(make <variable-name> name:` *n* `context:` *c*`)` creates an object that represents a variable name.

The name *n* is a symbol that provides the variable's name; if unsupplied it defaults to `temp`. It is okay to create lots of instances of `<variable-name>` with the same symbol as the `name:` argument; we presume that a development environment used for debugging macros will provide a pretty-printer that clarifies which variable names are the same as which other variable names, using an implementation-dependent notation or visual appearance. Programmers writing macros should not need to do anything kludgey such as constructing variable names by concatenating strings and numbers.

If the context *c* is not supplied, a new, never before seen lexical context is created and used. As a result, the new instance of `<variable-name>` cannot be = to any existing instance of `<variable-name>`. In addition a free reference to this variable name will have no meaning. This is similar but not identical to the handling of non-question-mark variables in an `expression` special form.

If the context *c* is supplied, it must be an instance of `<variable-name>`. The new variable name will have the same lexical context as *c*. Thus the newly made variable name behaves as if it had been introduced into the program at the exact same point as *c* and had been copied around by macros in exactly the same way. The idea is that if the context variable name is visible in some places and hidden in others as a result of hygiene, the constructed variable name will be visible or hidden in just the same places. This is orthogonal to lexical scoping.

The newly created instance of `<variable-name>` will be = to any instance of `<variable-name>` with the same name *n* in the same lexical context.

There are two main uses for contexts when making variable names. One is for name concatenation, as in the `define-structure` and `set!` examples below. Here part of the documented interface of a macro is that it references a variable name constructed by some processing on a variable name it was given. If the given variable name was visible in the part of the program that called the macro, the constructed variable name should be visible there too. If the given variable name was inserted into the macro call by another macro, the constructed variable name should be the one that was visible to that other macro. The context mechanism takes care of this.

The other use is for macros whose documented interface includes binding a variable with a particular name, such as the `loop` example below, which binds a variable named `exit`. Here the context should be such that if the call to `loop` was visible in

the original program before macro expansion, then the binding of `exit` should be visible, otherwise `exit` should be invisible. If `loop` is exported from a module, `exit` should be defined with a dummy definition and exported too. Defining it as a macro is best since it can give a compile-time error if called. Then a user can rename them both when importing them, if desired. This type of macro is usually considered poor programming style, but if it is needed it can be implemented using the context mechanism. The preferred programming style is to list all bound variable names explicitly in the macro call.

`(as <symbol>` *`variable-name`*`) => `*`symbol`* [Compile Stage G.F. Method]

This obtains the name symbol of a variable name.

`(as <string>` *`variable-name`*`) => `*`string`* [Compile Stage G.F. Method]

This obtains the name of a variable name as a string. It is really just a handy abbreviation for `(as <string> (as <symbol>` *`variable-name`*`))`.

`=` *`variable-name-1 variable-name-2 => boolean`* [Compile Stage G.F. Method]

This is a predicate that tests whether two variable names are the same name, i.e. have the same name symbol and are in the same lexical context. Operationally if the values of ?x and ?y are instances of `<variable-name>`, executing the expression whose representation is returned by

```
(expression (bind ((?x 1) (?y 2)) ?x))
```

returns 1 if `(/= ?x ?y)`, but returns 2 if `(= ?x ?y)`, in which case the binding named `?y` shadows the binding named `?x`.

The method is actually on `binary=`, but you call it via `=`.

Taking Expressions Apart

`expression-case` [Compile Stage Special Form]

An `expression-case` special form can be used to take apart an expression object, for example to extract the operator and operands of a combination. A `macro` special form implicitly uses `expression-case` to take apart a macro call expression that called the macro.

This special form binds local variables to expression objects representing the operands, or to sequences of expression objects when the ellipsis feature is used. As a bonus `expression-case` allows matching multiple patterns in case a macro has optional operands or formats and also checks for invalid syntax in a macro call.

Classic syntax: `(expression-case` *`expression`*
`(`*`pattern body-expression...`*`)`
`...)`
Algebraic syntax: *tbd*

The value of *`expression`* is the expression object to be taken apart. `Expression-case` considers each clause in order, trying to match the expression object to a clause's pattern. If the pattern does not match, `expression-case` goes on to the next clause. If it runs out of clauses, `expression-case` signals an error. If a match occurs, `expression-case` binds the question-mark variable names in the pattern to their matching subexpressions of the value of *`expression`*, evaluates the *`body-expressions`* in order, and examines the first value returned by the last *`body-expression`*. If that value is an instance of `<expression>`, it is the result of the `expression-case`. If that value is `#f`, `expression-case` goes on to the next clause, just as if the pattern had not matched. If that value is anything else, `expression-case` signals a type error.

A *`pattern`* is syntactically an expression. Certain subexpressions of a pattern have special treatment:

- The ellipsis `...`, which is a special compile-stage literal constant.
- The token `#key` (might change to the keyword `key:`), also a literal constant.
- Any variable name that begins with a question mark, including one that is nothing but a question mark.

As an expression, a pattern is a variable name, a literal constant, or a combination whose operator and operands are patterns. A pattern matches an expression object as follows:

- If the pattern is a variable name that begins with a question mark, then it matches anything, and a local variable with that name is bound to the matching expression, around the *`body-expressions`*. If a question mark variable name appears more than once in a pattern, the local variable's value is the expression matching the last occurrence; there is no unification of multiple occurrences. By convention, the variable name ? is used to match an expression that we don't care about.

- If the pattern is a variable name that does not begin with a question mark, then it matches any variable name with the same meaning. Two variable names have the same meaning if they are two names for one variable established by a `define`, `bind`, `bind-methods`, etc. form or by a parameter-list, or if they are two names for one macro. Two different variable names can have the same meaning because of renaming during module import, because of hygienic macro expansion, or because a macro was given a new name with `define` or `bind`.
  The meaning of a variable name depends on a lexical environment consisting

of local variable bindings and a module.
The meaning of a variable name in the expression object being taken apart is taken relative to the lexical environment at the dynamically innermost program stage transition. In the usual case, this is the lexical environment surrounding the macro call now being expanded.
The meaning of a variable name in a pattern is taken relative to the lexical environment at theinnermost program stage transition lexically nested around the pattern. In the usual case, that is the lexical environment surrounding the `macro` special form that defined the macro called by the macro call now being expanded.
Note that these meanings come from the next lower program stage, which is the stage where the expression being taken apart is presumably intended to be evaluated.

- If the pattern is a literal constant, it matches an expression that is a similar literal constant in the next lower program stage. The "similarity" concept was defined earlier.

- If the pattern is a combination not containing `...` or `#key`, it matches a combination that has the same number of operands, when each sub-pattern matches the corresponding sub-expression.

- The ellipsis `...` can be used as the last operand of a pattern that is a combination. If the pattern is a combination with n+1 operands, n≥0, it matches an expression that is a combination with n+m operands, m≥0, when all of the following are true:
  - the operators match
  - the first n operands match
  - each of the last m operands of the expression matches the "repeated" sub-pattern that immediately precedes the ellipsis; if n>0, this is the next to last operand of the pattern; if n=0, this is the operator of the pattern.

  If this repeated sub-pattern contains a variable name that begins with a question mark, then the corresponding local variable will be bound to a sequence of m+1 matching subexpressions. Ellipses can be nested, producing nested sequences.

- The token `#key` can be used as an operand of a pattern that is a combination. The pattern operands following `#key` have the same form as keyword parameters in a parameter-list. If the keyword is not specified explicitly and the variable name begins with a question mark, the keyword name defaults to the variable name without the question mark.
  If the pattern is a combination with n items (operator plus operands) preceding `#key`, n≥0, it matches an expression that is a combination with n+m items, m≥0 and a multiple of 2, when the first n items match, and the n+2nd, n+4th, etc.

items are each a literal constant whose value is a keyword that matches a keyword parameter in the pattern. Each keyword parameter causes a local variable to be bound around the *body-expressions*. If the n+kth item matches the keyword, the variable is bound to the n+k+1st item, otherwise it is bound to the value of its default.
An ellipsis cannot appear as an operand of the same combination in which `#key` appears. However, `#key` can be used inside a sub-pattern that is followed by an ellipsis, in which case the local variables bound by the keyword parameters are bound to sequences, with each sequence element being a matching expression object or the default. Note that, as in all other keyword parameters, if no default is specified, the default is `#f`. Usually you will want to specify a default such as `(expression #f)`.

The use of patterns should become clearer after reading the examples below

Putting Expressions Together

`expression` [Compile Stage Special Form]

Almost every macro uses an `expression` special form to put together an expression from subexpressions. This special form inserts subexpressions (including variable names) from locally accessible variables into a template to create an expression object. The result has a new lexical context, distinct from all existing lexical contexts, that refers to the lexical environment at the lexically innermost program stage transition nested around this `expression` special form.

This special form can also be used outside of macros, but only in the compile stage. For example, it can be used in helper functions that are called from the bodies of macros.

Classic syntax: `(expression` *template*`)`
Algebraic syntax: *tbd*

Templates are similar to patterns. A template is any expression. Each variable name in the template is replaced by that variable's value, in one of two ways:

- If the variable name does not begin with a question mark, then an instance of `<variable-name>` representing a variable name with the same meaning (see below) is inserted.

  This is a freshly created variable name in the lexical context created by this `expression` special form. It is not = to any existing instance of `<variable-name>`, so if this variable name is bound by the constructed expression, the binding will not be visible to any expressions in its scope except those constructed by

this same evaluation of this `expression` special form.

If the same non-question-mark variable name appears more than once in a given `expression` special form, only one corresponding new variable name is created. A new lexical context and therefore a new set of variable names is freshly created each time the `expression` special form is evaluated.

If the constructed expression contains a free reference to this freshly created variable name, its meaning is taken relative to the lexical environment at the lexically innermost program stage transition nested around the template. In the usual case, that is the lexical environment surrounding the `macro` special form that defined the macro called by the macro call now being expanded. Note that this meaning comes from the next lower program stage, which is the stage where the expression being put together is presumably intended to be evaluated.

- If the variable's name begins with a question mark, then the value of the locally accessible variable with that name must be an expression object (unless ellipses are involved; see below) and it is inserted. Most often the variable will have been bound by a surrounding `expression-case`, but this is not required.

A literal constant in a template is replaced in the constructed expression by a similar literal constant in the next lower program stage.

The structure of combinations in a template is used without change in the constructed expression. The operator and operands in a template that is a combination are themselves templates.

The ellipsis feature makes templates a little more general. Any template that is a combination can have the special literal constant ... as an operand. Unlike in a pattern, an ellipsis in a template does not have to be the last operand. The sub-template (whether operator or operand) immediately preceding the ellipsis is repeated zero or more times. The repeated sub-template must contain at least one variable name beginning with a question mark whose value is a sequence; the size of this sequence determines the number of repetitions. The nth repetition inserts the nth element of the sequence into the constructed expression. If the repeated sub-template contains more than one question mark variable whose value is a sequence, all the sequences must be the same size or an error is signalled. If the repeated sub-template also contains a question mark variable whose value is an expression object rather than a sequence, that expression object is repeated in each repetition.

Furthermore, ellipses can be nested: a repeated sub-template can contain ellipses. The value of a question mark variable can be a nested sequence of expression objects; the number of levels of sequence nesting can be as large as the number of

levels of ellipsis nesting, but no larger. Each level of ellipsis, from the outside in, iterates through elements of a level of sequence, also from the outside in. All sequences at a given level must be the same size, which is the number of repetitions of that level. If the number of levels of sequence nesting in a particular question mark variable is less than the number of levels of ellipsis nesting at a place where that variable name occurs in a template, the inner ellipses repeat an element of the innermost sequence.

The use of templates should become clearer after reading the examples below

Program Stages

A stage of a program consists of a set of expressions that could be executed at a particular time or place, such as in the runtime environment during normal execution (the run stage) or in the compiler during compilation (a compile stage), a set of variables that can be accessed from those expressions, and a set of objects that can be manipulated by those expressions.

At the bottom is the run stage. On top of that is the first compile stage, where computations occur that expand macros called from run stage expressions. Since macros can themselves use macros, there is actually an indefinitely tall tower of compile stages on top of the run stage. Because all of the compile stages share a single set of variables and objects, we often speak of "the compile stage," which comprises the whole tower of compile stages. We can speak of a variable or an object being in "the" compile stage, but we must speak of an expression being executed in "a" compile stage. We often speak of "the next lower program stage" below the one where a pattern or template is executed. Since the pattern and template features only exist in compile stages, we don't have to be concerned with what is the next lower program stage below the run stage.

The stage structure does not fork. There is only one second compile stage, used for computations involved in expanding all macros called from expressions in the first compile stage. This is as opposed to a design where each macro would have its own compile stage.

In delivery-oriented Dylan implementations such as Apple's, the run stage has completely separate objects and variables from the compile stage. The run stage is probably not even on the same machine as the compile stage (the compile stage runs on a programmer's machine while the run stage runs on a user's machine). It is also valid to implement Dylan in a "resident" style that uses the same objects and variables for all stages.

All stages share one set of modules. Thus the mapping of module variable names to their owning module is the same in all stages. There are separate copies of a

module variable in the run and compile stages (except in resident implementations). It is possible for a module variable to exist only in the run stage or only in the compile stages, in which case it is an error to reference it from an expression in the wrong stage.

Unless otherwise specified, variables in the Dylan module are available in all stages. Certain Dylan variables specified in this paper are only available in the compile stage. Are there any that are only available in the run stage? Probably not in the core Dylan language, but certainly some target-specific libraries will not be available in the compile stage.

Unless otherwise specified, all user code is in the run stage. The `macro` and `compile-stage` special forms boost their bodies up to the next higher stage. A `macro` special form can appear anywhere, not just at top level, so it is possible for run stage local variables to be in scope in compile stage code inside a macro. It is an error to reference such variables.

`compile-stage` [Top-level Special Form]

Classic syntax: `(compile-stage` *body* `...)`
Algebraic syntax: *tbd*

A `compile-stage` special form can be used only as a "top-level expression," as defined in the "Non-top-level Definitions" issue writeup. The *body* expressions are also "top-level expressions." The *body* expressions are in the compile stage one level up from where the special form was evaluated. `Compile-stage` special forms can be nested.

Examples:

```
(compile-stage
  (define foo (method ...)))
```

defines a method that can be used by macros as a helper function called at macro expansion time.

```
(define run-time-and-compile-time (macro
  ((? ?body ...)
   (expression
     (begin
       (compile-stage ?body ...)
       ?body ...)))))
(run-time-and-compile-time
  (define $buffer-size 1000))
```

defines a constant that can be used both at run-time and at macro expansion time.

Rewrite Rules

It is possible to use this mechanism in a simple rewrite-rules style without thinking about expression objects, stages, or hygiene. That style would probably be the first thing described in user documentation, but for purposes of this proposal I thought it best to explain the detailed mechanisms first.

In rewrite-rules style, each *pattern-clause* in the body of a `macro` has only one *body-expression,* which is an `expression` special form. Thus a macro does nothing but match its call to a pattern and build a new expression by rearranging pieces of the call. There is no explicit computation in the compile stage. `Macro` and `expression` are the only variables among those defined in this proposal that are used.

Many of the examples below are in rewrite-rules style. The examples section also shows how to define Scheme's `syntax-rules` operator as a macro-defining macro, however its syntax ends up not much different from the syntax of the existing `macro` operator, so it's probably not worth the bother .

A key feature is the smooth transition from rewrite rules to stage-shifting macros. When rewrite rules run out of gas and actual compile-stage computation is required, the programmer does not fall off a cliff and have to learn something completely different. It's only necessary to broaden the range of *body-expressions* used in *pattern-clauses.*

Variable Name Accessibility Restriction

There is a restriction on what variable names can appear in the expression constructed as the expansion of a macro, whether they are put in as non-question-mark variable names or by question-mark variables whose values are or contain instances of `<variable-name>`. This restriction exists to prevent breaking the encapsulation of lexical contours, of modules, and of stages.

Specifically, each free variable in an expression must exist in the program stage where the expression is evaluated and must be "potentially accessible" in the lexical environment surrounding the expression. Intuitively , "potentially accessible" means the variable either is lexically visible, is lexically shadowed by another variable with the same name, is an accessible module variable, or is a module variable that could have been imported. More precisely, a local variable is "potentially accessible" if the expression is lexically contained within the variable's binding, and a module variable is "potentially accessible" if it belongs to the same module in which the expression is compiled, or if it is exported from any module.

For example, in this program:

```
(bind ((var1 10)
       (bad (macro (? (bind ((var2 11))
                        (expression var2)))))
       (good (macro (? (expression var1))))
       (okay (macro (? (bind ((var1 11))
                        (expression var1))))))
  (bind ((var1 12))
    (list (bad) (good) (okay))))
```

the macro named `good` is valid, because the `var1` whose value is 10 is potentially accessible where `good` is called, even though it's shadowed by the `var1` whose value is 12. However, the macro named `bad` is not valid, because the variable name `var2` in its template does not refer to the binding of `var2` inside the macro; that `var2` is in the same program stage as the template, not one stage lower Instead the variable name `var2` in the template refers to some `var2` that was visible where the `macro` expression appears, but there does not seem to be any such variable. If there had been a local or module variable `var2`, the macro named `bad` would have been valid. In contrast, the macro named `okay` is valid; the variable name var1 in its template refers to the `var1` whose value is 10 which is lexically visible where the `macro` expression appears and is still potentially accessible where the macro is called, not the `var1` whose value is 11 that is lexically visible where the template appears, and not the `var1` whose value is 12 that is lexically visible where the macro is called.

Another example is:

```
(bind ((foo 0)
       (var1 10)
       (foo 1)
       (m (macro ((? ?var)
                  (make <variable-name> name: (as <symbol> ?var)
                                        context: (expression var1)))))
       (foo 2))
  (m foo))
```

The call to `m` expands into a reference to the variable named `foo` visible where the `macro` expression appears. This is the `foo` whose value is 1, and is potentially accessible where `m` is called. Here's how it works: The `expression` special form creates a new lexical context that refers to the lexical environment surrounding the `macro` special form. This lexical environment contains two variables named `foo` and a variable named `var1`. `Make` makes a variable named `foo` in that lexical context, and the macro call to `m` expands into a free reference to that variable. This free reference is taken relative to the lexical environment where `foo` is bound to 1.

Any variable name could have been used in the `expression` form used as the context and the result would have been the same, unless this example had been constructed by macro expansion, in which case the variable name used in the `expression` form could have been copied from someplace else where there is a different `foo`. The binding of `var1` to 10 is only for explanatory purposes.

It isn't possible to use a construct directly analogous to the above to violate the restriction and gain access to a local variable, because the required context cannot be constructed. For example, this:

```
(bind ((foo 0)
       (lose 0)
       (var1 10)
       (m (macro ((? ?var)
                  (bind ((lose 10)
                         (foo 20))
                    (make <variable-name> name: (as <symbol> ?var)
                                          context: (expression lose)))))))
  (m foo))
```

does not violate the restriction because `(expression lose)` refers to the variable named `lose` whose value is 0, visible where the `macro` expression appears, not the `lose` bound inside the macro. Therefore the variable name `foo` which is the expansion of `(m foo)` refers to the `foo` whose value is 0, not the `foo` inside the macro whose value is 20.

However, it is possible to violate the restriction by playing games with expression objects, saving an expression object from one macro call and using it in another, as in the following example:

```
(compile-stage
  (define-variable shenanigans 0))
  (define capture (macro
    ((? ?var) (set! shenanigans ?var) (expression #f)))))
(bind ((foo 0)
       (lose 0)
       (var1 10)
       (m (macro ((? ?var)
                  (bind ((lose 10)
                         (foo 20))
                    (capture lose)
                    (make <variable-name> name: (as <symbol> ?var)
                                          context: shenanigans))))))
  (m foo))
```

Here the context, `shenanigans`, is an instance of `<variable-name>` that represents some occurrence of the variable `lose` bound to 10 inside the macro. Thus you might expect the expansion of `(m foo)` to be a reference to some occurrence of the variable `foo` bound to 20 inside the macro. In fact this program is invalid because that `foo` is not potentially accessible where the macro is called. The behavior of this program is not defined.

In general, programs such as the above that play fast-and-loose with expression objects, moving them from one stage to another or one macro call to another, will

produce undefined results.

The following example potentially violates the restriction for module variables. An implementation could check macro expansions to detect this violation.

```
(define var1 10)
(define foo (secret-internal-stuff))
(define m (macro ((? ?var)
                  (make <variable-name> name: (as <symbol> ?var)
                                        context: (expression var1)))))
;and then in another module, assuming m is exported and foo is not
(m foo)
```

The macro call `(m foo)` expands into a reference to the non-exported variable `foo` that belongs to the module where the macro `m` was defined.

Rationale

Most of the rationale for doing it this particular way will only become apparent when algebraic syntax is put in, so I am going to delay writing this section.

Possible Future Features

This section sketches features that are not being proposed now, but that someone might possibly want to add later .

A “public but non-exported” module variable feature that the programmer can use to declare that certain variables will be incorporated into the expansion of macros exported by a module, but should not be imported when another module “uses” this module. This has been discussed in the past but is not on the Dylan Design Queue. For now programmers can just export such variables.

Special form operators could be treated like macros. That would make it possible to do nifty things like:

```
(define Dylan-if if)
(bind ((if 35)
       (fi 53))
  (Dylan-if (foo?) if fi))
```

The punctuation words found in algebraic syntax, such as `else`, could be treated similarly . This doesn’t let us rename keywords though, so it’s not really enough to enable a user to translate the entire Dylan language into a foreign tongue.

Constant expressions could be supported. Define a constant expression as a literal constant, a read-only variable whose initial-value is a constant expression, a macro call that expands into a constant expression, or a function call whose operands are constant expressions and whose operator is one of a certain set.

`(constant? expression)` is a predicate that returns true for a constant expression. `(constant-value expression)` accepts a constant expression and returns the object to which it evaluates. This provides the promised way to extract the object from a `<literal-constant>`.

Compile stage interface to the compiler's database. This means a way for macros to associate their own information with module variable names, leveraging off the compiler's mechanism for transmitting information about the values of module variables between separate compilation units.

`<syntax-error>` condition, signalled when expression-case runs out of clauses and also signallable by user code in macros. For now we don't specify what subclass of `<error>` is signalled.

A `make-load-form` generic function as in Common Lisp, so more types of objects can be enclosed in literal constants.

Type-inference on expression objects, which macros could certainly make use of.

A way to manipulate lexical environments explicitly and expand macros relative to a specified lexical environment, like Common Lisp's `macroexpand-1`. This can be used to pass information from one macro call to another by encapsulating it in local macro definitions. Alternatively there could be a way for macros to associate their own information with local variable names.

Reflective stuff, functional substrate underlying patterns and templates.

Symbol-macros using a `(symbol-macro expansion)` special form.

Examples (after Dybvig TR #356)

At present, none of these examples have been tested. Page numbers refer to pages in Dybvig's report.

```
;p.6
(define and2 (macro
  ((? ?x ?y) (expression (if ?x ?y #f)))))

;p.7
(define when (macro
  ((? ?test ?forms ...)
   (expression (if ?test (begin ?forms ...) #f)))))

;p.7
(define unless (macro
  ((? ?test ?forms ...)
   (expression (when (not ?test) ?forms ...)))))
```

```
;p.7
(bind ((when (macro ((? ?test ?forms ...)
                     (expression (if ?test (begin ?forms ...) #f))))))
  (when blah blah blah))

;p.7
;this shows how to write a recursive local macro
(bind ((and (macro ((?) (expression #t))
                   ((? ?e) ?e)
                   ((?and ?e1 ?e2 ...)
                    (expression (if ?e1 (?and ?e2 ...) #f))))))
  (and blah blah blah))

;p.8
;a recursive global macro is easier since its name is in scope
;this also shows how to introduce a new non-conflicting binding
(define or (macro
  ((?) (expression #f))
  ((? ?e) ?e)
  ((? ?e1 ?e2 ...)
   (expression
     (bind ((temp ?e1))
       (if temp temp (or ?e2 ...)))))))

;p.9
;this shows how to introduce a new visible binding
(define let1 (macro
  ((? ((?var ?val) ...) ?body ...)
   (expression ((method (?var ...) ?body ...) ?val ...)))))

;p.9
;this adds Scheme's "named let" feature
(define let2 (macro
  ((? ((?var ?val) ...) ?body ...)
   (expression ((method (?var ...) ?body ...) ?val ...)))
  ((? ?name ((?var ?val) ...) ?body ...)
   (expression ((bind-methods ((?name (?var ...) ?body ...))
                  ?name)
                ?val ...)))))

;p.9
;In Dybvig's version, this shows how to use a fender
; (instead of clause ordering)
;We use ordinary Dylan conditionals instead of fenders
(define let3 (macro
  ((? ?name ((?var ?val) ...) ?body ...)
   (when (instance? ?name <variable-name>)
     (expression ((bind-methods ((?name (?var ...) ?body ...))
                    ?name)
                  ?val ...))))
  ((? ((?var ?val) ...) ?body ...)
   (expression ((method (?var ...) ?body ...) ?val ...)))))
```

```
;p.10
;this shows how to define a local function inside a macro, shared
;among all branches of the expression-case.  This requires using
;the implicit expression-case only to bind ?x to the macro call.
(define let4 (macro
  (?x
   (bind-methods ((all-vars? (ls)
                    (every? (rcurry instance? <variable-name>) ls)))
     (expression-case ?x
       ((? ((?var ?val) ...) ?body ...)
        (when (all-vars? ?var)
          (expression ((method (?var ...) ?body ...) ?val ...))))
       ((? ?name ((?var ?val) ...) ?body ...)
        (when (all-vars? (concatenate (list ?name) ?var))
          (expression ((bind-methods ((?name (?var ...) ?body ...))
                        ?name)
                      ?val ...)))))))))

;The same thing with the helper function globally instead of locally defined.
;This way let5 could have avoided having its own copy of all-vars?
(compile-stage
  (define all-vars? (method (ls)
    (every? (rcurry instance? <variable-name>) ls))))
(define let4a (macro
  ((? ((?var ?val) ...) ?body ...)
   (when (all-vars? ?var)
     (expression ((method (?var ...) ?body ...) ?val ...))))
  ((? ?name ((?var ?val) ...) ?body ...)
   (when (all-vars? (concatenate (list ?name) ?var))
     (expression ((bind-methods ((?name (?var ...) ?body ...))
                   ?name)
                 ?val ...))))))

;p.10
;this shows how to report errors explicitly rather than letting
;the implicit expression-case do it
(define let5 (macro
  (?x
   (bind-methods ((all-vars? (ls)
                    (every? (rcurry instance? <variable-name>) ls)))
     (expression-case ?x
       ((? ((?var ?val) ...) ?body ...)
        (if (all-vars? ?var)
          (expression ((method (?var ...) ?body ...) ?val ...))
          (error "Bad variable name in ~S" ?x)))
       ((? ?name ((?var ?val) ...) ?body ...)
        (when (instance? ?name <variable-name>)
          (if (all-vars? ?var)
            (expression ((bind-methods ((?name (?var ...) ?body ...))
                          ?name)
                        ?val ...))
            (error "Bad variable name in ~S" ?x)))))))))
```

```
;p.11
;this adds a check for duplicate variable names
;note that = on instances of <variable-name> is the "same name" test we want
(define let6 (macro
  (?x
   (bind-methods ((unique-vars? (ls)
                    (and (all-vars? ls)
                         (unique? (as <list> ls))))
                  (all-vars? (ls)
                    (every? (rcurry instance? <variable-name>) ls))
                  (unique? (ls)
                    (or (empty? ls)
                        (and (notmem? (car ls) (cdr ls))
                             (unique? (cdr ls)))))
                  (notmem? (x ls)
                    (or (empty? ls)
                        (and (/= x (car ls))
                             (notmem? x (cdr ls))))))
     (expression-case ?x
       ((? ((?var ?val) ...) ?body ...)
        (if (unique-vars? ?var)
          (expression ((method (?var ...) ?body ...) ?val ...))
          (error "Bad or duplicated variable name in ~S" ?x)))
       ((? ?name ((?var ?val) ...) ?body ...)
        (when (instance? ?name <variable-name>)
          (if (unique-vars? ?var)
            (expression ((bind-methods ((?name (?var ...) ?body ...))
                           ?name)
                         ?val ...))
            (error "Bad or duplicated variable name in ~S" ?x)))))))))

;p.12
;shows nested ellipses
(define let7 (macro
  ((? ((?var ?val1 ?val2 ...) ...) ?body ...)
   (expression ((method (?var ...) ?body ...)
                (begin ?val1 ?val2 ...) ...)))))

;p.12
;we don't need with-syntax so I'll skip the example of defining it
;the or example that uses with-syntax would instead be done this way
(define or (macro
  ((?) (expression #f))
  ((? ?e) ?e)
  ((? ?e1 ?e2 ?e3 ...)
   (bind ((?rest (expression (or ?e2 ?e3 ...))))
     (expression (bind ((temp ?e1))
                   (if temp temp ?rest)))))))
```

```
;p.13
;Dybvig uses dotted pairs for optionals, which is wrong since that doesn't
;limit it to just one argument (but he checks for that) and which we can't do
;because our combinations aren't necessarily lists.  I'll use ellipsis.
;Actually Dybig's code is rife with horrifying puns between combinations
;and sequences.  It's actually kind of nice to be forced by our more abstract
;notion of expression objects to get rid of them.
(define do (macro
  (?orig-x
   (expression-case ?orig-x
     ((? ((?var ?init ?step ...) ...) ?e0 ?e1 ...) ?body ...)
      (bind ((?step (map (method (v sl)  ;default unsupplied step expressions
                           (cond ((empty? sl) v)
                                 ((= (size sl) 1) (first sl))
                                 (else: (error "Bad syntax: ~S" ?orig-x))))
                         ;; ?var is a sequence of variables
                         ;; ?step is a sequence of sequences of optional steps
                         ?var ?step)))
        ;; Now ?step is just a sequence of step expressions
        ;; This 'if' is necessary in Scheme where a begin is not allowed to
        ;; be empty, but in Dylan the 'else' clause would be sufficient
        ;; named-bind is not an official part of Dylan but I used it anyway
        ;; to keep the code here analogous to Dybvig's code
        (if (empty? ?e1)
          (expression
            (named-bind doloop ((?var ?init) ...)
              (when (not ?e0)
                ?body ...
                (doloop ?step ...))))
          (expression
            (named-bind doloop ((?var ?init) ...)
              (if ?e0
                (begin ?e1 ...)
                (begin ?body ... (doloop ?step ...)))))))))))

;p.13
;shows that we don't need with-syntax and generate-temporaries
(define letrec (macro
  ((? ((?var ?val) ...) ?body ...)
   (bind ((?temp (map (method (v) (make <variable-name>)) ?val)))
     (expression (bind ((?var #f) ...)
                   (bind ((?temp ?val) ...)
                     (set! ?var ?temp) ...
                     ?body ...)))))))

;another way to write the same thing
;each execution of (expression tem) creates a fresh variable-name
;can't use (map (always (expression tem)) ?val) since that would create
;a sequence each of whose elements was the same variable-name
(define letrec (macro
  ((? ((?var ?val) ...) ?body ...)
   (bind ((?temp (map (method (v) (expression tem)) ?val)))
     (expression (bind ((?var #f) ...)
                   (bind ((?temp ?val) ...)
                     (set! ?var ?temp) ...
                     ?body ...)))))))
```

```
;p.13
(define cond1 (macro
  ((? (else ?e1 ?e2 ...))
   (expression (begin ?e1 ?e2 ...)))
  ((? (?e0))
   (expression (bind ((temp ?e0))       ;could just return ?e0
                 (if temp temp #f))))
  ((? (?e0) ?c1 ?c2 ...)
   (expression (bind ((temp ?e0))
                 (if temp temp (cond1 ?c1 ?c2 ...)))))
  ((? (?e0 => ?e1))
   (expression (bind ((temp ?e0))
                 (if temp (?e1 temp) #f))))
  ((? (?e0 => ?e1) ?c1 ?c2 ...)
   (expression (bind ((temp ?e0))
                 (if temp (?e1 temp) (cond1 ?c1 ?c2 ...)))))
  ((? (?e0 ?e1 ?e2 ...))
   (expression (if ?e0 (begin ?e1 ?e2 ...) #f)))
  ((? (?e0 ?e1 ?e2 ...) ?c1 ?c2 ...)
   (expression (if ?e0 (begin ?e1 ?e2 ...) (cond1 ?c1 ?c2 ...))))))

;p.14
;this is a dumb way to write cond but exercises some features
;Dylan has no equivalent of Dybvig's free-identifier=? so this has
;to be written a little differently.  I've kept the structure
;analogous to his example so you can see the correspondence.
(define cond2 (macro
  ((? (?x ?e1 ?e2 ...))
   (and (instance? ?x <variable-name>)
        (expression-case ?x (else #t) (? #f))
        (expression (begin ?e1 ?e2 ...))))
  ((? (?e0))
   (expression (bind ((temp ?e0))       ;could just return ?e0
                 (if temp temp #f))))
  ((? (?e0) ?c1 ?c2 ...)
   (expression (bind ((temp ?e0))
                 (if temp temp (cond1 ?c1 ?c2 ...)))))
  ((? (?e0 ?x ?e1))
   (and (instance? ?x <variable-name>)
        (expression-case ?x (=> #t) (? #f))
        (expression (bind ((temp ?e0))
                      (if temp (?e1 temp) #f)))))
  ((? (?e0 ?x ?e1) ?c1 ?c2 ...)
   (and (instance? ?x <variable-name>)
        (expression-case ?x (=> #t) (? #f))
        (expression (bind ((temp ?e0))
                      (if temp (?e1 temp) (cond1 ?c1 ?c2 ...))))))
  ((? (?e0 ?e1 ?e2 ...))
   (expression (if ?e0 (begin ?e1 ?e2 ...) #f)))
  ((? (?e0 ?e1 ?e2 ...) ?c1 ?c2 ...)
   (expression (if ?e0 (begin ?e1 ?e2 ...) (cond1 ?c1 ?c2 ...))))))
```

```
;p.15
;This shows an internal recursion.  It doesn't need with-syntax
(define cond3 (macro
  (?orig-x
   (bind-methods
       ((docond (x)
         (expression-case x
          ((? (else ?e1 ?e2 ...))
           (expression (begin ?e1 ?e2)))
          ((? (?e0))
           (expression (bind ((temp ?e0)) (if temp temp #f))))
          ((? (?e0) ?c1 ?c2 ...)
           (bind ((?rest (docond (expression (cond3 ?c1 ?c2 ...)))))
             (expression (bind ((temp ?e0)) (if temp temp ?rest)))))
          ((? (?e0 => ?e1))
           (expression (bind ((temp ?e0)) (if temp (?e1 temp) #f))))
          ((? (?e0 => ?e1) ?c1 ?c2 ...)
           (bind ((?rest (docond (expression (cond3 ?c1 ?c2 ...)))))
             (expression (bind ((temp ?e0)) (if temp (?e1 temp) ?rest)))))
          ((? (?e0 ?e1 ?e2 ...))
           (expression (if ?e0 (begin ?e1 ?e2 ...) #f)))
          ((? (?e0 ?e1 ?e2 ...) ?c1 ?c2 ...)
           (bind ((?rest (docond (expression (cond3 ?c1 ?c2 ...)))))
             (expression (if ?e0 (begin ?e1 ?e2 ...) ?rest))))
          (? (error "Bad cond syntax: ~S" ?orig-x)))))
     (docond ?orig-x)))))

;p.15
;Dybvig's cond4 has a pun between sequences and combinations again
;We can't use expression to construct a sequence of expression objects; we
;either need to make a sequence explicitly, or make a begin expression
;Dybvig uses dotted lists here too, which we cannot
;The expansion uses elseless if, which is invalid in Dylan, but the
;whole point of the example rests on that, so I left it
(define cond4 (macro
  (?x
   (bind ((cond-if-tail (method (clauses)
            (if (empty? clauses)
              clauses
              (list (expression (cond4 ?clauses ...)))))))
     (expression-case ?x
       ((? (else ?e1 ?e2 ...))
        (expression (begin ?e1 ?e2)))
       ((? (?e0) ?c ...)
        (bind ((?tail (cond-if-tail ?c)))
          (expression (bind ((temp ?e0)) (if temp temp ?tail ...)))))
       ((? (?e0 => ?e1) ?c ...)
        (bind ((?tail (cond-if-tail ?c)))
          (expression (bind ((temp ?e0)) (if temp (?e1 temp) ?tail ...)))))
       ((? (?e0 ?e1 ?e2 ...) ?c ...)
        (bind ((?tail (cond-if-tail ?c)))
          (expression (if ?e0 (begin ?e1 ?e2 ...) ?tail ...)))))))))
```

```
;p.16
;a different way of writing the above, again we have to be clear what's
;a sequence versus what's an expression object
;Dybvig's "(null? (syntax tail))" is either a bug, or shouldn't work!
(define cond5 (macro
  (?orig-x
   (expression-case ?orig-x
     ((? ?c0 ?c1 ...)
      (bind ((?tail (if (empty? ?c1)
                        ?c1
                        (list (expression (cond5 ?c1 ...))))))
        (expression-case ?c0
          ((else ?e1 ?e2 ...)
           (when (empty? ?tail)
             (expression (begin ?e1 ?e2 ...))))
          ((?e0)
           (expression (bind ((temp ?e0)) (if temp temp ?tail ...))))
          ((?e0 => ?e1)
           (expression (bind ((temp ?e0)) (if temp (?e1 temp) ?tail ...))))
          ((?e0 ?e1 ?e2 ...)
           (expression (if ?e0 (begin ?e1 ?e2 ...) ?tail ...)))
          (? (error "Bad cond syntax: ~S" ?orig-x)))))))))

;p.17
(define cond6 (macro
  (?orig-x
   (bind-methods
       ((docond (x)
         (expression-case x
           ((? ?c0 ?c1 ...)
            (bind ((?tail (if (empty? ?c1)
                              ?c1
                              (list (docond (expression (cond6 ?c1 ...)))))))
              (expression-case ?c0
                ((else ?e1 ?e2 ...)
                 (when (empty? ?tail)
                   (expression (begin ?e1 ?e2 ...))))
                ((?e0)
                 (expression (bind ((temp ?e0)) (if temp temp ?tail ...))))
                ((?e0 => ?e1)
                 (expression (bind ((temp ?e0))
                                   (if temp (?e1 temp) ?tail ...))))
                ((?e0 ?e1 ?e2 ...)
                 (expression (if ?e0 (begin ?e1 ?e2 ...) ?tail ...)))
                (? (error "Bad cond syntax: ~S" ?orig-x))))))))
     (docond ?orig-x)))))

;a different way to write the same thing, where the argument to docond
;is a list of clauses instead of a cond expression
(define cond6a (macro
  (?orig-x
   (bind-methods
       ((docond (clauses)
          (bind ((?tail (if (empty? (cdr clauses))
                            '()
                            (list (docond (cdr clauses))))))
```

```
            (expression-case (car clauses)
              ((else ?e1 ?e2 ...)
               (when (empty? ?tail)
                 (expression (begin ?e1 ?e2 ...))))
              ((?e0)
               (expression (bind ((temp ?e0)) (if temp temp ?tail ...))))
              ((?e0 => ?e1)
               (expression (bind ((temp ?e0))
                             (if temp (?e1 temp) ?tail ...))))
              ((?e0 ?e1 ?e2 ...)
               (expression (if ?e0 (begin ?e1 ?e2 ...) ?tail ...)))
              (? (error "Bad cond syntax: ~S" ?orig-x))))))
      (expression-case ?orig-x
        ((? ?c ...)
         (unless (empty? ?c)
           (docond (as <list> ?c)))))))))

;p.17
;This is complicated and inefficient but shows how to use local macros
;in the expansion of a macro to perform additional complex processing.
;Using mutually recursive local macros without letrec-syntax requires
;a cute "lambda-lifting" trick.
;This also shows how to insert ellipses and metavariables into patterns
;and templates costructed by a macro-defining macro.
;I think Dybvig has a "requoting bug" at his call to syntax-error; I've
;fixed it here by making orig-x a literal constant whose value is an
;expression object.
(define cond7 (macro
  ((? ?xc1 ?xc2 ...)
   (bind ((?dots $pattern-ellipsis)
          (? (make <variable-name> name: '?))
          (?e0 (make <variable-name> name: '?e0))
          (?e1 (make <variable-name> name: '?e1))
          (?e2 (make <variable-name> name: '?e2))
          (?c (make <variable-name> name: '?c))
          (?c1 (make <variable-name> name: '?c1))
          (?c2 (make <variable-name> name: '?c2))
          (?cond1 (make <variable-name> name: '?cond1))
          (?cond2 (make <variable-name> name: '?cond2))
          (?orig-x (make <literal-constant>
                     object: (expression (cond7 ?xc1 ?xc2 ...)))))
     (expression
       (bind ((cond1 (macro
                ((?cond1 ?cond2 (else ?e1 ?e2 ?dots))
                 (expression (begin ?e1 ?e2 ?dots)))
                ((?cond1 ?cond2 (?e0 => ?e1) ?c ?dots)
                 (expression (bind ((temp ?e0))
                               (?cond2 ?cond1 temp (?e1 temp) ?c ?dots))))
                ((?cond1 ?cond2 (?e0 ?e1 ?e2 ?dots) ?c ?dots)
                 (expression (?cond2 ?cond1 ?e0
                                     (begin ?e1 ?e2 ?dots)
                                     ?c ?dots)))
                ((?cond1 ?cond2 (?e0) ?c ?dots)
                 (expression (bind ((temp ?e0))
                               (?cond2 ?cond1 temp temp ?c ?dots))))
                (? (error "Bad cond syntax: ~S" ?orig-x))))
```

```
              (cond2 (macro
                ((?cond2 ?cond1 ?e0 ?e1)
                 (expression (if ?e0 ?e1)))
                ((?cond2 ?cond1 ?e0 ?e1 ?c1 ?c2 ?dots)
                 (expression (if ?e0
                                 ?e1
                                 (?cond1 ?cond2 ?c1 ?c2 ?dots)))))))
         (cond1 cond2 ?xc1 ?xc2 ...)))))))

;p.18
(bind ((if (macro ((? ?e1 ?e2 ?e3)
                   (expression (if ?e1 ?e2 ?e3))))))
  (if 1 2 3))

;p.18
;This is an example of something that doesn't work
;It causes an infinite loop of macro expansion because the
;?if in the expansion is bound by the pattern to the local
;'if' macro, not the global one, so the expansion is identical
;to the macro call.
(bind ((if (macro ((?if ?e1 ?e2 ?e3)
                   (expression (?if ?e1 ?e2 ?e3))))))
  (if 1 2 3))

;p.18
;This also is an example of something that doesn't work
;It fails because the 'if' in the pattern is outside the scope
;of the 'bind if', so it matches only a variable with the same
;meaning as the global 'if', which is not what it's matched against.
(bind ((if (macro ((if ?e1 ?e2 ?e3)
                   (expression (if ?e1 ?e2 ?e3))))))
  (if 1 2 3))

;p.19
;The expansion uses both the global 'if' and the local 'if' (value of p).
;This assumes the Scheme 'if', which allows omitting the else clause.
;The locally defined 'if' is more like 'cond' than any normal 'if'
(bind ((if (macro
             ((? ?e) ?e)
             ((? ?e1 ?e2) (expression (if ?e1 ?e2)))
             ((?p ?e1 ?e2 ?e3 ?e4 ...)
              (expression (if ?e1 ?e2 (?p ?e3 ?e4 ...)))))))
  (list (if #f 1 #f 2 #f 3 #f 4 5)
        (if #f 1 #f 2 #t 3 #f 4 5)
        (if #t 1 #f 2 #f 3 #f 4 5)))
```

```
;This example demonstrates that the meaning of a non-question-mark
;variable in a pattern or template is its meaning where the current
;program stage was entered, not its meaning where the pattern or
;template occurs.
(bind ((x 1)
       (m (macro ((? ?y)
                  (bind ((x 2))
                    (expression (+ ?y x)))))))
  (m 3))
;this evaluates to 4, the value of (+ 3 1), not 5, the value
;of (+ 3 2), because the x in the expansion is the outer x whose
;value is 1.  The inner x does not even exist in the right program
;stage for it to be able to appear in the expansion.

;pp.19-20
;This version of syntax-rules uses Dylan question-mark-style patterns
;and templates.  This makes it a very easy macro-defining-macro because
;it does not need to look inside the patterns nor inside the templates.
;It has the same bug as the first of Dybvig's two versions.
(define syntax-rules (macro
  ((? (?pattern => ?template) ...)
   (expression
     (macro (?pattern (expression ?template)) ...)))))

;An example of using the above version of syntax-rules
(define my-if (syntax-rules
  ((my-if ?test ?then)       => (cond (?test ?then) (else: #f)))
  ((my-if ?test ?then ?else) => (cond (?test ?then) (else: ?else)))))

;This version of syntax-rules ignores the operator position of each
;pattern, like it's supposed to, just like Dybvig's second version, which
;for some reason does this in a much more complicated way.
(define syntax-rules (macro
  ((? ((? ?subpattern ...) => ?template) ...)
   (bind ((? (make <variable-name> name: '?)))
     (expression
       (macro ((? ?subpattern ...) (expression ?template)) ...))))))

;This version of syntax-rules uses Scheme-style patterns and templates,
;with literals.  We have to translate them to Dylan style.  It also
;ignores the operator position of each pattern, like it's supposed to.
;This does not support improper lists, unlike the Scheme version.
(define syntax-rules (macro
  ((? (?literal ...) ((? ?subpattern ...) => ?template) ...)
   (bind ((? (make <variable-name> name: '?))
          (?dots $pattern-ellipsis)
          (pattern-variables '()))
     (bind-methods ((question ((v <variable-name>))
                      (make <variable-name>
                        name: (as <symbol> (concatenate "?" (as <string> v)))
                        context: v))
                    (translate-pattern (p)
                      (cond ((= p dots) p)
                            ((instance? p <literal-constant>) p)
                            ((and (instance? p <variable-name>)
                                  (member? p ?literal test: =))
```

```
                                p)
                               ((instance? p <variable-name>)
                                (set! pattern-variables
                                      (add-new! pattern-variables p))
                                (question p))
                               (else:
                                (expression-case p
                                  ((?pp ...)
                                   (bind ((?tr (map translate-pattern ?pp)))
                                     (expression (?tr ...)))))))))
                      (translate-template (t)
                        (cond ((= t dots) t)
                              ((instance? t <literal-constant>) t)
                              ((and (instance? t <variable-name>)
                                    (member? t pattern-variables test: =))
                               (question t))
                              ((instance? t <variable-name>) t)
                              (else:
                               (expression-case t
                                 ((?te ...)
                                  (bind ((?tr (map translate-template ?te)))
                                    (expression (?tr ...)))))))))
          (bind ((translations (map (method (pattern template)
                                      (set! pattern-variables '())
                                      (list (translate-pattern pattern)
                                            (translate-template template)))
                                    ?subpattern ?template))
                 (?subpattern (map first translations))
                 (?template (map second translations)))
            (expression
              (macro ((? ?subpattern ...) (expression ?template)) ...)))))))

;An example of using the above version of syntax-rules
(define my-if (syntax-rules (cond)
  ((my-if test then)       => (cond (test then) (else: #f)))
  ((my-if test then else)  => (cond (test then) (else: else)))))

;p.20
;This version assumes Dylan-style patterns and templates, hence no literals
;It could be expanded like the last definition of syntax-rules above
(define extend-syntax (macro
  ((? (?k0) ?c ...)
   (bind ((expand-clause (method (c)
            (expression-case c
              ((?pattern ?fender ?template)
               (expression (?pattern (when ?fender (expression ?template)))))
              ((?pattern ?template)
               (expression (?pattern (expression ?template))))))))
          (?c (map expand-clause ?c)))
     (expression (define ?k0 (macro ?c ...)))))))
```

```
;p.21
;Still uses Dylan-style patterns and templates
(define extend-syntax (macro
  ((? (?k0) ?c ...)
   (bind-methods
       ((expand-template (t)
          (expression-case t
            ((with ((?pattern ?expression) ...) ?template)
             (bind ((?template (expand-template ?template)))
               (expression
                 (expression-case (expression ?expression ...)
                   ((?pattern ...) ?template)))))
            (?template
             (expression (expression ?template)))))
        (expand-clause (c)
          (expression-case c
            ((?pattern ?fender ?template)
             (bind ((?template (expand-template ?template)))
               (expression (?pattern (when ?fender
                                       (expression ?template))))))
            ((?pattern ?template)
             (bind ((?template (expand-template ?template)))
               (expression (?pattern (expression ?template))))))))
     (bind ((?c (map expand-clause ?c)))
       (expression (define ?k0 (macro ?c ...)))))))

;p.21
;the key thing here is the context: argument to make <variable-name>
(define loop (macro
  ((?k ?e1 ?e2 ...)
   (bind ((?exit (make <variable-name> name: 'exit context: ?k)))
     (expression
       (bind-exit (?exit)
         (named-bind f ()
           ?e1 ?e2 ...
           (f))))))))

;p.22
(define loop (macro
  ((?k ?e1 ?e2 ...)
   (bind ((?exit (make <variable-name> name: 'exit context: ?k))
          (? (make <variable-name> name: '?))
          (?e (make <variable-name> name: '?e)))
     (expression
       (bind-exit (t)
         (bind ((?exit (macro ((? ?e ...)
                               (expression (t ?e ...))))))
           (named-bind f ()
             ?e1 ?e2 ...
             (f)))))))))
```

```
;p.23
;Note that we can't use quote (or ') to insert literal constants, we
;have to make them with make <literal-constant>.  That's because the
;expression object for a literal constant is not the actual constant.
(define define-structure (macro
  ((? (?name ?id1 ...))
   (expression (define-structure (?name ?id1 ...) ())))
  ((? (?name ?id1 ...) ((?id2 ?init) ...))
   (bind ((slots (concatenate ?id1 ?id2))
          (string (method (x) (as <string> x)))
          (construct-name (method (template-var #rest args)
            (make <variable-name>
              context: template-var
              name: (as <symbol> (apply concatenate (map string args))))))
          (constructor (construct-name ?name "make-" ?name))
          (predicate (construct-name ?name ?name "?"))
          (access (map (method (slot)
                         (construct-name slot ?name "-" slot))
                       slots))
          (assign (map (method (slot)
                         (construct-name slot "set-" ?name "-" slot "!"))
                       slots))
          (structure-name (make <literal-constant>
                                object: (as <symbol> ?name)))
          (structure-size (make <literal-constant>
                                object: (+ (size slots) 1)))
          (index (map (curry make <literal-constant> object:)
                      (range from: 1 size: (size slots)))))
     (expression
       (begin
         (define ?constructor (method (?id1 ...)
           (bind ((?id2 ?init) ...)
             (vector ?structure-name ?id1 ... ?id2 ...))))
         (define ?predicate (method (x)
           (and (instance? x <simple-object-vector>)
                (= (size x) ?structure-size)
                (id? (element x 0) ?structure-name))))
         (define ?access (method ((x <simple-object-vector>))
           (element x ?index)))
         ...
         (define ?assign (method ((x <simple-object-vector>) update)
           (set-element! x ?index update)))
         ...))))))
```

```
;The same but defines the constructor, predicate, and accessors as
;macros instead of functions, as an example of a macro-defining macro
(define define-structure (macro
  ((? (?name ?id1 ...))
   (expression (define-structure (?name ?id1 ...) ())))
  ((? (?name ?id1 ...) ((?id2 ?init) ...))
   (bind ((slots (concatenate ?id1 ?id2))
          (string (method (x) (as <string> x)))
          (construct-name (method (template-var #rest args)
            (make <variable-name>
              context: template-var
              name: (as <symbol> (apply concatenate (map string args))))))
          (question (method ((v <variable-name>))
                      (make <variable-name>
                        name: (as <symbol> (concatenate "?" (as <string> v)))
                        context: v)))
          (no-question
            (method (exp)
              (when (and (instance? exp <variable-name>)
                         (= (element (as <string> exp) 0 default: #\a) #\?))
                (error "~S isn't allowed to begin with a question mark"
                       exp))))
          (constructor (construct-name ?name "make-" ?name))
          (predicate (construct-name ?name ?name "?"))
          (access (map (method (slot)
                         (construct-name slot ?name "-" slot))
                       slots))
          (assign (map (method (slot)
                         (construct-name slot "set-" ?name "-" slot "!"))
                       slots))
          (structure-name (make <literal-constant>
                                object: (as <symbol> ?name)))
          (structure-size (make <literal-constant>
                                object: (+ (size slots) 1)))
          (index (map (curry make <literal-constant> object:)
                      (range from: 1 size: (size slots))))
          (? (make <variable-name> name: '?))
          (??id1 (map question ?id1))
          (?x (make <variable-name> name: '?x))
          (?update (make <variable-name> name: '?update)))
     (map no-question ?id1)
     (map no-question ?id2)
     (map no-question ?init)
     (expression
       (begin
         (define ?constructor (macro
           ((? ??id1 ...)
            (expression
              (bind ((?id1 ??id1) ...
                     (?id2 ?init) ...)
                (vector ?structure-name ?id1 ... ?id2 ...))))))
         (define ?predicate (macro
           ((? ?x)
            (expression
              (bind ((temp ?x))
                (and (instance? temp <simple-object-vector>)
```

```
                    (= (size temp) ?structure-size)
                    (id? (element temp 0) ?structure-name)))))))
          (define ?access (macro
            ((? ?x)
             (expression (element ?x ?index)))))
          ...
          (define ?assign (macro
            ((? ?x ?update)
             (expression (set-element! ?x ?index ?update)))))
          ...)))))))

;p.24
;I'll skip quasiquote since it's big and introduces no new concepts
;It would need to use make <literal-constant> rather than quote

;p.26
(define be-like-begin (macro
  ((? ?name)
   (bind ((?dots $pattern-ellipsis)
          (? (make <variable-name> name: '?))
          (?e0 (make <variable-name> name: '?e0))
          (?e1 (make <variable-name> name: '?e1)))
     (expression
       (define ?name (macro
         ((? ?e0 ?e1 ?dots)
          (expression (begin ?e0 ?e1 ?dots)))))))))))



;Here's an example that's instructive but not in Dybvig:
(define set!
  (macro
    ((? (?fn ?arg ...) ?new-value)
     (when (instance? ?fn <variable-name>)
       (bind ((?setter (make <variable-name>
                         context: ?fn
                         name: (as <symbol> (concatenate "set-"
                                                         (as <string> ?fn)
                                                         "!")))))
         (expression (?setter ?arg ... ?new-value)))))
    ((? ?var ?new-value)
     (when (instance? ?var <variable-name>)
       (expression (primitive-set! ?var ?new-value))))))

;not in Dybvig, nor in standard Dylan, but used in a lot of the preceding
;examples and also in a lot of Apple Dylan code that I have seen, and
;maybe even in some that I have written
(define named-bind (macro
  ((? ?name ((?var ?val) ...) ?body ...)
   (expression ((bind-methods ((?name (?var ...) ?body ...))
                  ?name)
                ?val ...)))))
```

```
;not in Dybvig
;Showing how to use #key
(define key-if (macro
  ((? ?test #key (?then (expression #t))
                 (?else (expression #f)))
   (expression (if ?test ?then ?else)))))

;Examples of using that macro
(key-if (> x y) then: x else: y)
(key-if a else: b)



;;;; Examples translated from Dylan-macros.lisp
;;;; I kept as closely as possible to the structure of the originals

(define cond (macro
  ((?) (expression #f))
  ((? (?else ?e ...))
   (when (else-name? ?else)
     (expression (begin ?e ...))))
  ((? (?e) ?c ...)
   (expression (or ?e (cond ?c ...))))
  ((? (?e => ?f) ?c ...)
   (expression (bind ((g ?e)) (if g (?f g) (cond ?c ...)))))
  ((? (?t ?=_or_bind ?var ?e ...) ?c ...)
   (and (expression-case ?=_or_bind
          (= #t)
          (bind #t)
          (? #f))
        (expression-case ?var
         ((?var ?specializer) (instance? ?var <variable-name>))
         (?var (instance? ?var <variable-name>))
         (? #f))
        (expression (cond (?t => (method (?var) ?e ...)) ?c ...))))
  ((? (?t ?e ...) ?c ...)
   (expression (if ?t (begin ?e ...) (cond ?c ...))))))

;see *else-names*
(define else-name? (method (x)
  (expression-case x
    (else #t)
    (otherwise #t)
    (else: #t)
    (otherwise: #t)
    (#t #t)
    (? #f))))

(define and (macro
  ((?) (expression #t))
  ((? ?f1) ?f1)
  ((? ?f1 ?ff ...)
   (expression (if ?f1 (and ?ff ...) #f)))))
```

```
(define or (macro
  ((?) (expression #f))
  ((? ?f1) ?f1)
  ((? ?f1 ?ff ...)
   (expression (bind ((v ?f1)) (if v v (or ?ff ...)))))))

(define when (macro
  ((? ?test ?form ...)
   (expression (if ?test (begin ?form ...) #f)))))

(define unless (macro
  ((? ?test ?form ...)
   (expression (if ?test #f (begin ?form ...))))))

;Note the screwy feature (not denied by the documentation) that
;the else clause does not have to be at the end
;This has the same bug as the original that it makes a literal constant
;out of an expression and passes it to case-error for use in an error
;message.  This is not really allowed.  We could add a mechanism for
;coercing an expression into a runtime object that can be used in an
;error message, but we don't really want that thing at runtime anyway.
;The qitem argument to case-error could be removed; for now I left it.
(define case (macro
  (?original-form
   (expression-case ?original-form
    ((? ?item (?keys ?expr ...) ...)
     (bind ((else-exprss (choose-by else-name? ?keys ?expr))
            (?main-exprss (choose-by (complement else-name?) ?keys ?expr))
            (main-keyss (choose (complement else-name?) ?keys))
            (combination->sequence (method (x)
                                     (expression-case x
                                       ((?key ...) ?key))))
            (main-keyss (map combination->sequence main-keyss))
            (all-keys (reduce concatenate '() main-keyss)))
       (when (> (size else-exprss) 1)
         (signal "Multiple ELSE clauses in ~S" ?original-form))
       (for ((k all-keys (cdr k)))
            ((empty? k))
         (when (member? (car k) (cdr k))
           (signal "CASE key ~S occurs multiple times in ~S."
                   (car k) ?original-form)))
       (bind ((?v (make <variable-name>))
              (?tests
               (map (method (keys)
                      (bind ((?k (map (curry make <literal-constant> object:)
                                      keys)))
                        (expression (or (id? ?v ?k) ...))))
                    main-keyss))
              (?else-exprs
               (if (empty? else-exprss)
                 (bind ((?qitem (make <literal-constant> object: ?item))
                        (?qkeys (make <literal-constant> object: all-keys)))
                   (list (expression
                           (case-error ?qitem ?v ?qkeys))))
                 (first else-exprss))))
         (expression
```

```
            (bind ((?v ?item))
              (cond (?tests ?main-exprss ...) ...
                    (#t ?else-exprs ...))))))))))))

(define select (macro
  (?original-form
   (expression-case ?original-form
    ((? ?item ?test (?keys ?expr ...) ...)
     (bind ((else-exprss (choose-by else-name? ?keys ?expr))
            (?main-exprss (choose-by (complement else-name?) ?keys ?expr))
            (main-keyss (choose (complement else-name?) ?keys))
            (combination->sequence (method (x)
                                     (expression-case x
                                       ((?key ...) ?key))))
            (main-keyss (map combination->sequence main-keyss))
            (all-keys (reduce concatenate '() main-keyss)))
       (when (> (size else-exprss) 1)
         (signal "Multiple ELSE clauses in ~S" ?original-form))
       (bind ((?obj (make <variable-name>))
              (?tst (make <variable-name>))
              (?tests
               (map (method (keys)
                      (expression (or (?tst ?obj ?keys) ...)))
                    main-keyss))
              (?else-exprs
               (if (empty? else-exprss)
                 (bind ((?qitem (make <literal-constant> object: ?item))
                        (?qkeys (make <literal-constant> object: all-keys)))
                   (list (expression
                           (select-error ?qitem ?obj ?tst ?qkeys))))
                 (first else-exprss))))
         (expression
          (bind ((?obj ?item)
                 (?tst ?test))
            (cond (?tests ?main-exprss ...) ...
                  (#t ?else-exprs ...)))))))))))

(define bind-exit (macro
  ((? (?variable) ?body ...)
   (expression (%catch-internal (method (?variable) ?body ...))))))

(define unwind-protect (macro
  ((? ?protected-form ?cleanup-form ...)
   (expression (%unwind-protect-internal (method () ?protected-form)
                                         (method () ?cleanup-form ...))))))
```

```
;; Includes feature that var means (var #f).
(define named-bind (macro
  ((? ?name (?bindings ...) ?body ...)
   (bind ((?vars (map (method (b) (expression-case b
                                    ((?var ?init) ?var)
                                    (?var         ?var)))
                      ?bindings))
          (?inits (map (method (b) (expression-case b
                                     ((?var ?init) ?init)
                                     (?var         (expression #f))))
                       ?bindings)))
     (expression (bind-methods ((?name ?vars ?body ...))
                   (?name ?inits ...)))))))

(define values-call (macro
  ((? ?function ?values)
   (bind ((function ?function)
          (#rest values-list ?values))
     (apply function values-list)))))

(define for (macro
  ((? ((?var ?init ?step) ...)
      (?exit-test ?exit-form ...)
    ?body ...)
   (expression
    (bind-methods ((loop ?var
                     (if ?exit-test
                       (begin ?exit-form ...)
                       (begin ?body ...
                              (loop ?step ...)))))
      (loop ?init ...))))))

(define iterate (macro
  ((? ((?var ?collection) ...)
      ?stuff
    ?body ...)
   ;; Parse out optional thingies
   (bind ((?exit-test ?incomplete-result ?complete-result
           (expression-case ?stuff
             (()
              (values #f #f #f))
             ((?exit-test)
              (values ?exit-test #f #f))
             ((?exit-test ?incomplete-result)
              (values ?exit-test ?incomplete-result #f))
             ((?exit-test ?incomplete-result ?complete-result)
              (values ?exit-test ?incomplete-result ?complete-result))))
          ;; Make temporary variables
          (?collection-var (map (method (v) (make <variable-name>)) ?var))
          (?state-var (map (method (v) (make <variable-name>)) ?var)))
     (expression
       (bind ((?collection-var ?collection) ...)
         (bind-methods
           ((loop (?state-var ...)
              (if (and ?state-var ...)
                (bind ((?var (current-element ?collection-var ?state-var))
```

```
                    ...)
                (if ?exit-test
                  ?incomplete-result
                  (begin ?body ...
                         (loop (next-state ?collection-var ?state-var)
                               ...))))
            ?complete-result)))
        (loop (initial-state ?collection-var) ...)))))))))

;This calls a Common Lisp library function at macro expansion time
(define 0x (macro
  ((? ?hex)
   (make <literal-constant>
         object: (#?parse-integer (as <string> ?hex) #?:radix 16)))))

;I dealt with the optional result a different way in dotimes
(define dolist (macro
 ((? (?var ?list) ?body ...)
  (expression
    (bind-methods ((loop (list)
                     (if (pair? list)
                       (bind ((?var (car list)))
                         ?body ...
                         (loop (cdr list)))
                       #f)))
      (loop ?list))))
 ((? (?var ?list ?result) ?body ...)
  (expression
    (bind-methods ((loop (list)
                     (if (pair? list)
                       (bind ((?var (car list)))
                         ?body ...
                         (loop (cdr list)))
                       ?result)))
      (loop ?list))))))

;I dealt with the optional result a different way in dolist
(define dotimes (macro
 ((? (?var ?limit ?optional-result ...) ?body ...)
  (when (<= (size ?optional-result) 1)
    (bind ((?result (if (empty? ?optional-result)
                      (expression #f)
                      (first ?optional-result))))
      (expression
        (bind ((n ?limit))
          (if (integer? n)
            (bind-methods ((f (?var)
                             (if (< ?var n)
                               (begin ?body ...
                                      (f (+ ?var 1)))
                               ?result)))
              (f 0))
            (error "Not an integer: ~S" n)))))))))
```

```
(define for-each (macro
  ((? ((?var ?collection) ...)
      ((?exit-test-and-results ...))
    ?body ...)
   (bind ((?exit-test ?results
           (expression-case ?exit-test-and-results
             (() (values #f '()))
             ((?exit-test ?results ...)
              (values ?exit-test ?results))))
          (?collection-var (map (method (v) (make <variable-name>)) ?var))
          (?state-var (map (method (v) (make <variable-name>)) ?var)))
     (expression
       (bind ((?collection-var ?collection) ...)
         (bind-methods
             ((loop (?state-var)
                (if (and ?state-var ...)
                  (bind ((?var (current-element ?collection-var ?state-var)))
                    (if ?exit-test
                      (begin #f ?results ...)
                      (begin ?body ...
                             (loop (next-state ?collection-var ?state-var)
                                   ...))))
                  #f)))
           (loop (initial-state ?collection-var) ...)))))))

;the extraneous introduction of thunks was probably a vain attempt
;to be hygienic.  I imitated it, just to show it could be done.
(define while (macro
  ((? ?test ?body ...)
   (expression
    (bind ((test (method () ?test))
           (body (method () ?body ...)))
      (bind-methods ((loop ()
                       (when (test) (body) (loop))))
        (loop)))))))

;this version of while is simpler and just as hygienic
(define while (macro
  ((? ?test ?body ...)
   (expression
    (bind-methods ((loop ()
                     (when ?test
                       ?body ...
                       (loop))))
      (loop))))))

(define ignore (macro
  ((? ?body ...)
   (expression
    (when #f ?body ...)))))
```

```
(define handler-bind (macro
  ((? (?type ?function #key (?test (expression (always #t)))
                            (?description (expression #f)))
      ?body ...)
   (expression
    (handler-bind-internal ?type ?test ?function ?description
      (method () ?body ...))))))

;I replicated the bug in the existing macro that the clauses are
;handled in reverse order, so that the innermost handler is the one
;from the last clause.  According to the Dylan book the innermost
;handler should be the one from the first clause.
(define handler-case (macro
  ((? ?protected-form
      ((?type #key (?condition (make <variable-name> name: 'condition))
                   (?test (expression (always #t)))
                   (?description (expression #f)))
       ?body ...) ...)
   (bind ((?inner-exit (map (method (c) (make <variable-name>)) ?condition))
          (?outer-exit (make <variable-name>)))
     (bind-methods ((wrap-handler-bind (i)
                      (if (< i (size ?condition))
                        (bind ((?type (element ?type i))
                               (?test (element ?test i))
                               (?desc (element ?description i))
                               (?inner-exit (element ?inner-exit i))
                               (?function (expression
                                            (method (condition next-handler)
                                              (ignore next-handler)
                                              (?inner-exit condition))))
                               (?inner (wrap-handler-bind (+ i 1))))
                          (expression
                            (handler-bind-internal ?type ?test ?function
                                                   ?desc
                              (method () ?inner))))
                        (expression
                          (values-call ?outer-exit ?protected-form))))
                    (wrap-body (i)
                      (if (< i (size ?condition))
                        (bind ((?inner-exit (element ?inner-exit i))
                               (?cond (element ?condition i))
                               (?body (element ?body i))
                               (?inner (wrap-body (+ i 1))))
                          (expression
                            (bind ((?cond (bind-exit (?inner-exit)
                                                     ?inner)))
                              (ignore ?cond)
                              (values-call ?outer-exit
                                           (begin ?body ...)))))
                        (wrap-handler-bind 0))))
       (bind ((?doit (wrap-body 0)))
         (expression
          (bind-exit (?outer-exit) ?doit)))))))))
```

```
(define with-safe-printing (macro
  ((? ?body ...)
   (expression
    (handler-case (begin ?body ...)
      ((<error>) (format #t "{whoops! printing error occurred}")))))))

;Well, approximately, it's hard to deal with all the #?'s and
;this probably should have been a define-Dylan-defining-macro
;instead of a defmacro or def-Dylan-macro-function.
(define define-tag/field-frobber (macro
  ((? ?dylan-name ?lisp-name)
   (bind ((lispfn (make <literal-constant>
                        object: (as <symbol> ?lisp-name)))
          (?form (make <variable-name> name: '?form)))
     (expression
      (define ?dylan-name (macro
        (?form (tag/field-frobber-internal ?form ?lispfn)))))))))
```

What I learned from that exercise:
• things like case-error and select-error that take arguments that are run-time representations of expressions are bad.
• lack of “or” and “optional” in patterns can be awkward, but it seems always possible to get around it by adding additional calls to expression-case in the body-expression of a pattern clause.
• Dylan macros whose expansion functions really need to call Common Lisp are always going to be a problem to write in Dylan!

Comparison to Dybvig and Hieb

This system is basically similar to Dybvig and Hieb’s, with some names changed to be more consistent with Dylan terminology . For example, I have renamed “syntax” to “expression” because where Scheme uses “syntax” to mean macros, Dylan uses “syntax” to mean surface syntax, for example infix versus parenthesized.

In addition, this system is simpler than Dybvig and Hieb’s. I have employed objects to decrease the number of named functions, substituting `make`, `as`, and `instance?`. The idea of macros as ectoplasmic objects eliminates the need for the special name binding forms `define-syntax` and `let-syntax` (use `define` and `bind` respectively) and thus avoids any need to add a macro feature to modules.

`Letrec-syntax` is eliminated because I don’t believe it is necessary (recursive macros receive a name for themselves as the operator of the expression that called them; mutually recursive macros can be done as in the `cond7` example).

This system flags pattern variables with question marks instead of listing “literals” in `expression-case`. Literals would require all uses of `expression` to be lexically nested inside an `expression-case`, would be error-prone because the status of a

variable in a pattern or template is not attached directly to an occurrence of the variable, and cannot work comfortably in algebraic syntax where some variable references in an abstract expression are represented in the surface syntax by punctuation, rather than by name. As a bonus, the elimination of “literals” eliminates Dybvig and Hieb’s kludgey pattern variables (we use ordinary variables) and thus eliminates `with-syntax`.

I have eliminated fenders. They were only necessary because Dybvig and Hieb did not use an abstract data type for expressions and thus could not distinguish the false value from the literal constant expression object whose value is false.

The special case`(syntax ...)` [or `(syntax (... ...))`, I’m not sure which one in Dybvig’s paper is the typo for the other one], has been replaced by the constant variable `$pattern-ellipsis`.

Certain things that can be done in Scheme cannot be done in Dylan as a result of the switch from S-expressions to abstract syntax in the macro system. For example, a macro call cannot involve improper lists (dotted pairs) except of course inside of literal constants. Furthermore, symbols in a macro call that appear to be variables but are really literal constants with the ‘ left off for convenience should be avoided. Dybvig’s kludgey processing of `quote`, which walks over the constant object stripping “marks” from symbols, is not needed in this system.

Table of name mappings:

Dybvig and Hieb	Dylan
`define-syntax`	`define`
`let-syntax`	`bind` (scoping differs with multiple clauses)
`letrec-syntax`	not provided
`syntax-case`	`expression-case`
`syntax`	`expression`
`...`	`...`
`with-syntax`	use `bind`
`identifier?`	`instance?` of `<variable-name>`
`free-identifier=?`	not provided
`bound-identifier=?`	`=`
`syntax-object->datum?`	`as <symbol>` for variable names not allowed for other expressions
`implicit-identifier`	`make` of `<variable-name>`
`generate-temporaries`	`map make` of `<variable-name>`
`syntax-error`	`error`

Comparison to Bawden

The <expression> class resembles Bawden's item class, however they are not really the same. Also <expression> is exposed to the macro writer Most of <expression>'s subclasses are not exposed.

This system does not allow for symbol macros, as a result of not having `bind-syntax`.

`make-setter-name` is replaced by `make` of `<variable-name>` as shown in the `set!` example.

Table of name mappings:

Bawden	Dylan
`define-syntax`	`define`
`bind-syntax-after`	`bind`
`bind-syntax-before`	not provided
`macro`	`macro` (but the body is different)
`syntax-rules`	too close to `macro` to bother with!
`syntax-case`	`expression-case`
`syntax`	`expression`
`...`	`...`
`classify`	not exposed
`make-closure`	not exposed
`make-setter-name`	not needed
`item`	`<expression>`
`variable-item`	`<variable-name>`
`constant-item`	`<literal-constant>`
`syntax-object-name?`	`instance?` of `<variable-name>`
`name-equal?`	`=`
`syntax-object->datum?`	`as <symbol>` for variable names not allowed for other expressions
`syntax-object->list`	not provided
`syntax-object-car`	not provided
`syntax-object-equal?`	not provided