```
                                        (define (bar) (+ x 3))
                                        (* (bar) 2))
$1 = (define foo
       (lambda (#{x 94}#) (* (+ #{x 94}# 3) 2)))
```

The partial evaluator does not inline top-level bindings, though, so this is a situation where you may find it interesting to use `define-inlinable`.

Procedures defined with `define-inlinable` are *always* inlined, at all direct call sites. This eliminates function call overhead at the expense of an increase in code size. Additionally, the caller will not transparently use the new definition if the inline procedure is redefined. It is not possible to trace an inlined procedures or install a breakpoint in it (see Section 6.26.4 [Traps], page 484). For these reasons, you should not make a procedure inlinable unless it demonstrably improves performance in a crucial way.

In general, only small procedures should be considered for inlining, as making large procedures inlinable will probably result in an increase in code size. Additionally, the elimination of the call overhead rarely matters for large procedures.

`define-inlinable` (*name parameter ...*) *body1 body2 ...* [Scheme Syntax]

Define *name* as a procedure with parameters *parameter*s and bodies *body1*, *body2*, ...

6.10 Macros

At its best, programming in Lisp is an iterative process of building up a language appropriate to the problem at hand, and then solving the problem in that language. Defining new procedures is part of that, but Lisp also allows the user to extend its syntax, with its famous *macros*.

Macros are syntactic extensions which cause the expression that they appear in to be transformed in some way *before* being evaluated. In expressions that are intended for macro transformation, the identifier that names the relevant macro must appear as the first element, like this:

```
(macro-name macro-args ...)
```

Macro expansion is a separate phase of evaluation, run before code is interpreted or compiled. A macro is a program that runs on programs, translating an embedded language into core Scheme[5].

6.10.1 Defining Macros

A macro is a binding between a keyword and a syntax transformer. Since it's difficult to discuss `define-syntax` without discussing the format of transformers, consider the following example macro definition:

```
(define-syntax when
  (syntax-rules ()
    ((when condition exp ...)
     (if condition
```

[5] These days such embedded languages are often referred to as *embedded domain-specific languages*, or EDSLs.

```
          (begin exp ...)))))

(when #t
  (display "hey ho\n")
  (display "let's go\n"))
⊣ hey ho
⊣ let's go
```

In this example, the `when` binding is bound with `define-syntax`. Syntax transformers are discussed in more depth in Section 6.10.2 [Syntax Rules], page 263, and Section 6.10.3 [Syntax Case], page 268.

`define-syntax` *keyword transformer* [Syntax]

Bind *keyword* to the syntax transformer obtained by evaluating *transformer*.

After a macro has been defined, further instances of *keyword* in Scheme source code will invoke the syntax transformer defined by *transformer*.

One can also establish local syntactic bindings with `let-syntax`.

`let-syntax` ((*keyword transformer*) ...) *exp1 exp2 ...* [Syntax]

Bind each *keyword* to its corresponding *transformer* while expanding *exp1 exp2 ...*.

A `let-syntax` binding only exists at expansion-time.

```
(let-syntax ((unless
              (syntax-rules ()
                ((unless condition exp ...)
                 (if (not condition)
                     (begin exp ...))))))
  (unless #t
    (primitive-exit 1))
  "rock rock rock")
⇒ "rock rock rock"
```

A `define-syntax` form is valid anywhere a definition may appear: at the top-level, or locally. Just as a local `define` expands out to an instance of `letrec`, a local `define-syntax` expands out to `letrec-syntax`.

`letrec-syntax` ((*keyword transformer*) ...) *exp1 exp2 ...* [Syntax]

Bind each *keyword* to its corresponding *transformer* while expanding *exp1 exp2 ...*.

In the spirit of `letrec` versus `let`, an expansion produced by *transformer* may reference a *keyword* bound by the same *letrec-syntax*.

```
(letrec-syntax ((my-or
                 (syntax-rules ()
                   ((my-or)
                    #t)
                   ((my-or exp)
                    exp)
                   ((my-or exp rest ...)
                    (let ((t exp))
```

```
                          (if t
                              t
                              (my-or rest ...)))))))
  (my-or #f "rockaway beach"))
⇒ "rockaway beach"
```

6.10.2 Syntax-rules Macros

`syntax-rules` macros are simple, pattern-driven syntax transformers, with a beauty worthy of Scheme.

`syntax-rules` *literals* (*pattern template*) ... [Syntax]

Create a syntax transformer that will rewrite an expression using the rules embodied in the *pattern* and *template* clauses.

A `syntax-rules` macro consists of three parts: the literals (if any), the patterns, and as many templates as there are patterns.

When the syntax expander sees the invocation of a `syntax-rules` macro, it matches the expression against the patterns, in order, and rewrites the expression using the template from the first matching pattern. If no pattern matches, a syntax error is signalled.

6.10.2.1 Patterns

We have already seen some examples of patterns in the previous section: `(unless condition exp ...)`, `(my-or exp)`, and so on. A pattern is structured like the expression that it is to match. It can have nested structure as well, like `(let ((var val) ...) exp exp* ...)`. Broadly speaking, patterns are made of lists, improper lists, vectors, identifiers, and datums. Users can match a sequence of patterns using the ellipsis (`...`).

Identifiers in a pattern are called *literals* if they are present in the `syntax-rules` literals list, and *pattern variables* otherwise. When building up the macro output, the expander replaces instances of a pattern variable in the template with the matched subexpression.

```
(define-syntax kwote
  (syntax-rules ()
    ((kwote exp)
     (quote exp))))
(kwote (foo . bar))
⇒ (foo . bar)
```

An improper list of patterns matches as rest arguments do:

```
(define-syntax let1
  (syntax-rules ()
    ((_ (var val) . exps)
     (let ((var val)) . exps))))
```

However this definition of `let1` probably isn't what you want, as the tail pattern *exps* will match non-lists, like `(let1 (foo 'bar) . baz)`. So often instead of using improper lists as patterns, ellipsized patterns are better. Instances of a pattern variable in the template must be followed by an ellipsis.

```
(define-syntax let1
  (syntax-rules ()
```

```
    ((_ (var val) exp ...)
     (let ((var val)) exp ...))))
```

This `let1` probably still doesn't do what we want, because the body matches sequences of zero expressions, like `(let1 (foo 'bar))`. In this case we need to assert we have at least one body expression. A common idiom for this is to name the ellipsized pattern variable with an asterisk:

```
(define-syntax let1
  (syntax-rules ()
    ((_ (var val) exp exp* ...)
     (let ((var val)) exp exp* ...))))
```

A vector of patterns matches a vector whose contents match the patterns, including ellipsizing and tail patterns.

```
(define-syntax letv
  (syntax-rules ()
    ((_ #((var val) ...) exp exp* ...)
     (let ((var val) ...) exp exp* ...))))
(letv #((foo 'bar)) foo)
⇒ bar
```

Literals are used to match specific datums in an expression, like the use of `=>` and `else` in `cond` expressions.

```
(define-syntax cond1
  (syntax-rules (=> else)
    ((cond1 test => fun)
     (let ((exp test))
       (if exp (fun exp) #f)))
    ((cond1 test exp exp* ...)
     (if test (begin exp exp* ...)))
    ((cond1 else exp exp* ...)
     (begin exp exp* ...))))

(define (square x) (* x x))
(cond1 10 => square)
⇒ 100
(let ((=> #t))
  (cond1 10 => square))
⇒ #<procedure square (x)>
```

A literal matches an input expression if the input expression is an identifier with the same name as the literal, and both are unbound[6].

Although literals can be unbound, usually they are bound to allow them to be imported, exported, and renamed. See Section 6.20 [Modules], page 410, for more information on imports and exports. In Guile there are a few standard auxiliary syntax definitions, as specified by R6RS and R7RS:

[6] Language lawyers probably see the need here for use of `literal-identifier=?` rather than `free-identifier=?`, and would probably be correct. Patches accepted.

`else` [Scheme Syntax]
`=>` [Scheme Syntax]
`_` [Scheme Syntax]
`...` [Scheme Syntax]

Auxiliary syntax definitions.

These are defined as if with a macro that never matches, e.g.:

```
(define-syntax else (syntax-rules ()))
```

If a pattern is not a list, vector, or an identifier, it matches as a literal, with `equal?`.

```
(define-syntax define-matcher-macro
  (syntax-rules ()
    ((_ name lit)
     (define-syntax name
       (syntax-rules ()
        ((_ lit) #t)
        ((_ else) #f))))))

(define-matcher-macro is-literal-foo? "foo")

(is-literal-foo? "foo")
⇒ #t
(is-literal-foo? "bar")
⇒ #f
(let ((foo "foo"))
  (is-literal-foo? foo))
⇒ #f
```

The last example indicates that matching happens at expansion-time, not at run-time.

Syntax-rules macros are always used as (*macro* . *args*), and the *macro* will always be a symbol. Correspondingly, a `syntax-rules` pattern must be a list (proper or improper), and the first pattern in that list must be an identifier. Incidentally it can be any identifier – it doesn't have to actually be the name of the macro. Thus the following three are equivalent:

```
(define-syntax when
  (syntax-rules ()
    ((when c e ...)
     (if c (begin e ...)))))

(define-syntax when
  (syntax-rules ()
    ((_ c e ...)
     (if c (begin e ...)))))

(define-syntax when
  (syntax-rules ()
    ((something-else-entirely c e ...)
     (if c (begin e ...)))))
```

For clarity, use one of the first two variants. Also note that since the pattern variable will always match the macro itself (e.g., `cond1`), it is actually left unbound in the template.

6.10.2.2 Hygiene

`syntax-rules` macros have a magical property: they preserve referential transparency. When you read a macro definition, any free bindings in that macro are resolved relative to the macro definition; and when you read a macro instantiation, all free bindings in that expression are resolved relative to the expression.

This property is sometimes known as *hygiene*, and it does aid in code cleanliness. In your macro definitions, you can feel free to introduce temporary variables, without worrying about inadvertently introducing bindings into the macro expansion.

Consider the definition of `my-or` from the previous section:

```
(define-syntax my-or
  (syntax-rules ()
    ((my-or)
     #t)
    ((my-or exp)
     exp)
    ((my-or exp rest ...)
     (let ((t exp))
       (if t
           t
           (my-or rest ...))))))
```

A naive expansion of `(let ((t #t)) (my-or #f t))` would yield:

```
(let ((t #t))
  (let ((t #f))
    (if t t t)))
⇒ #f
```

Which clearly is not what we want. Somehow the `t` in the definition is distinct from the `t` at the site of use; and it is indeed this distinction that is maintained by the syntax expander, when expanding hygienic macros.

This discussion is mostly relevant in the context of traditional Lisp macros (see Section 6.10.5 [Defmacros], page 275), which do not preserve referential transparency. Hygiene adds to the expressive power of Scheme.

6.10.2.3 Shorthands

One often ends up writing simple one-clause `syntax-rules` macros. There is a convenient shorthand for this idiom, in the form of `define-syntax-rule`.

`define-syntax-rule` (*keyword . pattern*) [*docstring*] *template* [Syntax]
Define *keyword* as a new `syntax-rules` macro with one clause.

Cast into this form, our `when` example is significantly shorter:

```
(define-syntax-rule (when c e ...)
  (if c (begin e ...)))
```

6.10.2.4 Reporting Syntax Errors in Macros

`syntax-error` *message* [*arg ...*] [Syntax]

Report an error at macro-expansion time. *message* must be a string literal, and the optional *arg* operands can be arbitrary expressions providing additional information.

`syntax-error` is intended to be used within `syntax-rules` templates. For example:

```
(define-syntax simple-let
  (syntax-rules ()
    ((_ (head ... ((x . y) val) . tail)
        body1 body2 ...)
     (syntax-error
      "expected an identifier but got"
      (x . y)))
    ((_ ((name val) ...) body1 body2 ...)
     ((lambda (name ...) body1 body2 ...)
      val ...))))
```

6.10.2.5 Specifying a Custom Ellipsis Identifier

When writing macros that generate macro definitions, it is convenient to use a different ellipsis identifier at each level. Guile allows the desired ellipsis identifier to be specified as the first operand to `syntax-rules`, as specified by SRFI-46 and R7RS. For example:

```
(define-syntax define-quotation-macros
  (syntax-rules ()
    ((_ (macro-name head-symbol) ...)
     (begin (define-syntax macro-name
              (syntax-rules ::: ()
                ((_ x :::)
                 (quote (head-symbol x :::)))))
            ...))))
(define-quotation-macros (quote-a a) (quote-b b) (quote-c c))
(quote-a 1 2 3) ⇒ (a 1 2 3)
```

6.10.2.6 Further Information

For a formal definition of `syntax-rules` and its pattern language, see See Section "Macros" in *Revised(5) Report on the Algorithmic Language Scheme*.

`syntax-rules` macros are simple and clean, but do they have limitations. They do not lend themselves to expressive error messages: patterns either match or they don't. Their ability to generate code is limited to template-driven expansion; often one needs to define a number of helper macros to get real work done. Sometimes one wants to introduce a binding into the lexical context of the generated code; this is impossible with `syntax-rules`. Relatedly, they cannot programmatically generate identifiers.

The solution to all of these problems is to use `syntax-case` if you need its features. But if for some reason you're stuck with `syntax-rules`, you might enjoy Joe Marshall's `syntax-rules` Primer for the Merely Eccentric (`http://sites.google.com/site/evalapply/eccentric.txt`).

6.10.3 Support for the `syntax-case` System

`syntax-case` macros are procedural syntax transformers, with a power worthy of Scheme.

`syntax-case` *syntax literals* (*pattern* [*guard*] *exp*) ... [Syntax]
Match the syntax object *syntax* against the given patterns, in order. If a *pattern* matches, return the result of evaluating the associated *exp*.

Compare the following definitions of `when`:

```
(define-syntax when
  (syntax-rules ()
    ((_ test e e* ...)
     (if test (begin e e* ...)))))

(define-syntax when
  (lambda (x)
    (syntax-case x ()
      ((_ test e e* ...)
       #'(if test (begin e e* ...))))))
```

Clearly, the `syntax-case` definition is similar to its `syntax-rules` counterpart, and equally clearly there are some differences. The `syntax-case` definition is wrapped in a `lambda`, a function of one argument; that argument is passed to the `syntax-case` invocation; and the "return value" of the macro has a `#'` prefix.

All of these differences stem from the fact that `syntax-case` does not define a syntax transformer itself – instead, `syntax-case` expressions provide a way to destructure a *syntax object*, and to rebuild syntax objects as output.

So the `lambda` wrapper is simply a leaky implementation detail, that syntax transformers are just functions that transform syntax to syntax. This should not be surprising, given that we have already described macros as "programs that write programs". `syntax-case` is simply a way to take apart and put together program text, and to be a valid syntax transformer it needs to be wrapped in a procedure.

Unlike traditional Lisp macros (see Section 6.10.5 [Defmacros], page 275), `syntax-case` macros transform syntax objects, not raw Scheme forms. Recall the naive expansion of `my-or` given in the previous section:

```
(let ((t #t))
  (my-or #f t))
;; naive expansion:
(let ((t #t))
  (let ((t #f))
    (if t t t)))
```

Raw Scheme forms simply don't have enough information to distinguish the first two `t` instances in `(if t t t)` from the third `t`. So instead of representing identifiers as symbols, the syntax expander represents identifiers as annotated syntax objects, attaching such information to those syntax objects as is needed to maintain referential transparency.

`syntax` *form* [Syntax]
Create a syntax object wrapping *form* within the current lexical context.

Syntax objects are typically created internally to the process of expansion, but it is possible to create them outside of syntax expansion:

```
(syntax (foo bar baz))
⇒ #<some representation of that syntax>
```

However it is more common, and useful, to create syntax objects when building output from a `syntax-case` expression.

```
(define-syntax add1
  (lambda (x)
    (syntax-case x ()
      ((_ exp)
       (syntax (+ exp 1))))))
```

It is not strictly necessary for a `syntax-case` expression to return a syntax object, because `syntax-case` expressions can be used in helper functions, or otherwise used outside of syntax expansion itself. However a syntax transformer procedure must return a syntax object, so most uses of `syntax-case` do end up returning syntax objects.

Here in this case, the form that built the return value was `(syntax (+ exp 1))`. The interesting thing about this is that within a `syntax` expression, any appearance of a pattern variable is substituted into the resulting syntax object, carrying with it all relevant metadata from the source expression, such as lexical identity and source location.

Indeed, a pattern variable may only be referenced from inside a `syntax` form. The syntax expander would raise an error when defining `add1` if it found *exp* referenced outside a `syntax` form.

Since `syntax` appears frequently in macro-heavy code, it has a special reader macro: `#'`. `#'foo` is transformed by the reader into `(syntax foo)`, just as `'foo` is transformed into `(quote foo)`.

The pattern language used by `syntax-case` is conveniently the same language used by `syntax-rules`. Given this, Guile actually defines `syntax-rules` in terms of `syntax-case`:

```
(define-syntax syntax-rules
  (lambda (x)
    (syntax-case x ()
      ((_ (k ...) ((keyword . pattern) template) ...)
       #'(lambda (x)
           (syntax-case x (k ...)
             ((dummy . pattern) #'template)
             ...))))))
```

And that's that.

6.10.3.1 Why `syntax-case`?

The examples we have shown thus far could just as well have been expressed with `syntax-rules`, and have just shown that `syntax-case` is more verbose, which is true. But there is a difference: `syntax-case` creates *procedural* macros, giving the full power of Scheme to the macro expander. This has many practical applications.

A common desire is to be able to match a form only if it is an identifier. This is impossible with `syntax-rules`, given the datum matching forms. But with `syntax-case` it is easy:

`identifier?` *syntax-object* [Scheme Procedure]

Returns `#t` if *syntax-object* is an identifier, or `#f` otherwise.

```
;; relying on previous add1 definition
(define-syntax add1!
  (lambda (x)
    (syntax-case x ()
      ((_ var) (identifier? #'var)
       #'(set! var (add1 var))))))

(define foo 0)
(add1! foo)
foo ⇒ 1
(add1! "not-an-identifier") ⇒ error
```

With `syntax-rules`, the error for `(add1! "not-an-identifier")` would be something like "invalid `set!`". With `syntax-case`, it will say something like "invalid `add1!`", because we attach the *guard clause* to the pattern: `(identifier? #'var)`. This becomes more important with more complicated macros. It is necessary to use `identifier?`, because to the expander, an identifier is more than a bare symbol.

Note that even in the guard clause, we reference the *var* pattern variable within a `syntax` form, via `#'var`.

Another common desire is to introduce bindings into the lexical context of the output expression. One example would be in the so-called "anaphoric macros", like `aif`. Anaphoric macros bind some expression to a well-known identifier, often `it`, within their bodies. For example, in `(aif (foo) (bar it))`, `it` would be bound to the result of `(foo)`.

To begin with, we should mention a solution that doesn't work:

```
;; doesn't work
(define-syntax aif
  (lambda (x)
    (syntax-case x ()
      ((_ test then else)
       #'(let ((it test))
           (if it then else))))))
```

The reason that this doesn't work is that, by default, the expander will preserve referential transparency; the *then* and *else* expressions won't have access to the binding of `it`.

But they can, if we explicitly introduce a binding via `datum->syntax`.

`datum->syntax` *template-id datum* [Scheme Procedure]

Create a syntax object that wraps *datum*, within the lexical context corresponding to the identifier *template-id*.

For completeness, we should mention that it is possible to strip the metadata from a syntax object, returning a raw Scheme datum:

`syntax->datum` *syntax-object* [Scheme Procedure]

Strip the metadata from *syntax-object*, returning its contents as a raw Scheme datum.

In this case we want to introduce `it` in the context of the whole expression, so we can create a syntax object as `(datum->syntax x 'it)`, where `x` is the whole expression, as passed to the transformer procedure.

Here's another solution that doesn't work:

```
;; doesn't work either
(define-syntax aif
  (lambda (x)
    (syntax-case x ()
      ((_ test then else)
       (let ((it (datum->syntax x 'it)))
         #'(let ((it test))
             (if it then else)))))))
```

The reason that this one doesn't work is that there are really two environments at work here – the environment of pattern variables, as bound by `syntax-case`, and the environment of lexical variables, as bound by normal Scheme. The outer let form establishes a binding in the environment of lexical variables, but the inner let form is inside a syntax form, where only pattern variables will be substituted. Here we need to introduce a piece of the lexical environment into the pattern variable environment, and we can do so using `syntax-case` itself:

```
;; works, but is obtuse
(define-syntax aif
  (lambda (x)
    (syntax-case x ()
      ((_ test then else)
       ;; invoking syntax-case on the generated
       ;; syntax object to expose it to 'syntax'
       (syntax-case (datum->syntax x 'it) ()
         (it
           #'(let ((it test))
               (if it then else))))))))

(aif (getuid) (display it) (display "none")) (newline)
⊣ 500
```

However there are easier ways to write this. `with-syntax` is often convenient:

`with-syntax` ((*pat val*) ...) *exp* ... [Syntax]

Bind patterns *pat* from their corresponding values *val*, within the lexical context of *exp* ....

```
;; better
(define-syntax aif
  (lambda (x)
    (syntax-case x ()
      ((_ test then else)
       (with-syntax ((it (datum->syntax x 'it)))
         #'(let ((it test))
             (if it then else)))))))
```

As you might imagine, `with-syntax` is defined in terms of `syntax-case`. But even that might be off-putting to you if you are an old Lisp macro hacker, used to building macro output with `quasiquote`. The issue is that `with-syntax` creates a separation between the point of definition of a value and its point of substitution.

So for cases in which a `quasiquote` style makes more sense, `syntax-case` also defines `quasisyntax`, and the related `unsyntax` and `unsyntax-splicing`, abbreviated by the reader as `#‘`, `#,`, and `#,@`, respectively.

For example, to define a macro that inserts a compile-time timestamp into a source file, one may write:

```
(define-syntax display-compile-timestamp
  (lambda (x)
    (syntax-case x ()
      ((_)
       #‘(begin
          (display "The compile timestamp was: ")
          (display #,(current-time))
          (newline))))))
```

Readers interested in further information on `syntax-case` macros should see R. Kent Dybvig's excellent *The Scheme Programming Language*, either edition 3 or 4, in the chapter on syntax. Dybvig was the primary author of the `syntax-case` system. The book itself is available online at `http://scheme.com/tspl4/`.

6.10.3.2 Custom Ellipsis Identifiers for syntax-case Macros

When writing procedural macros that generate macro definitions, it is convenient to use a different ellipsis identifier at each level. Guile supports this for procedural macros using the `with-ellipsis` special form:

`with-ellipsis` *ellipsis body ...* [Syntax]

ellipsis must be an identifier. Evaluate *body* in a special lexical environment such that all macro patterns and templates within *body* will use *ellipsis* as the ellipsis identifier instead of the usual three dots (`...`).

For example:

```
(define-syntax define-quotation-macros
  (lambda (x)
    (syntax-case x ()
      ((_ (macro-name head-symbol) ...)
       #’(begin (define-syntax macro-name
                  (lambda (x)
                    (with-ellipsis :::
                      (syntax-case x ()
                        ((_ x :::)
                         #’(quote (head-symbol x :::)))))))))
                ...)))))
(define-quotation-macros (quote-a a) (quote-b b) (quote-c c))
(quote-a 1 2 3) ⇒ (a 1 2 3)
```

Note that `with-ellipsis` does not affect the ellipsis identifier of the generated code, unless `with-ellipsis` is included around the generated code.

6.10.4 Syntax Transformer Helpers

As noted in the previous section, Guile's syntax expander operates on syntax objects. Procedural macros consume and produce syntax objects. This section describes some of the auxiliary helpers that procedural macros can use to compare, generate, and query objects of this data type.

`bound-identifier=?` *a b* [Scheme Procedure]
Return `#t` if the syntax objects *a* and *b* refer to the same lexically-bound identifier, or `#f` otherwise.

`free-identifier=?` *a b* [Scheme Procedure]
Return `#t` if the syntax objects *a* and *b* refer to the same free identifier, or `#f` otherwise.

`generate-temporaries` *ls* [Scheme Procedure]
Return a list of temporary identifiers as long as *ls* is long.

`syntax-source` *x* [Scheme Procedure]
Return the source properties that correspond to the syntax object *x*. See Section 6.26.2 [Source Properties], page 477, for more information.

Guile also offers some more experimental interfaces in a separate module. As was the case with the Large Hadron Collider, it is unclear to our senior macrologists whether adding these interfaces will result in awesomeness or in the destruction of Guile via the creation of a singularity. We will preserve their functionality through the 2.0 series, but we reserve the right to modify them in a future stable series, to a more than usual degree.

```
(use-modules (system syntax))
```

`syntax-module` *id* [Scheme Procedure]
Return the name of the module whose source contains the identifier *id*.

`syntax-local-binding` *id* [Scheme Procedure]
[*#:resolve-syntax-parameters?=#t*]
Resolve the identifer *id*, a syntax object, within the current lexical environment, and return two values, the binding type and a binding value. The binding type is a symbol, which may be one of the following:

`lexical` A lexically-bound variable. The value is a unique token (in the sense of `eq?`) identifying this binding.

`macro` A syntax transformer, either local or global. The value is the transformer procedure.

`syntax-parameter`
A syntax parameter (see Section 6.10.7 [Syntax Parameters], page 277). By default, `syntax-local-binding` will resolve syntax parameters, so that this value will not be returned. Pass `#:resolve-syntax-parameters? #f` to indicate that you are interested in syntax parameters. The value is the default transformer procedure, as in `macro`.

`pattern-variable`
: A pattern variable, bound via `syntax-case`. The value is an opaque object, internal to the expander.

`ellipsis`
: An internal binding, bound via `with-ellipsis`. The value is the (anti-marked) local ellipsis identifier.

`displaced-lexical`
: A lexical variable that has gone out of scope. This can happen if a badly-written procedural macro saves a syntax object, then attempts to introduce it in a context in which it is unbound. The value is `#f`.

`global`
: A global binding. The value is a pair, whose head is the symbol, and whose tail is the name of the module in which to resolve the symbol.

`other`
: Some other binding, like `lambda` or other core bindings. The value is `#f`.

This is a very low-level procedure, with limited uses. One case in which it is useful is to build abstractions that associate auxiliary information with macros:

```
(define aux-property (make-object-property))
(define-syntax-rule (with-aux aux value)
  (let ((trans value))
    (set! (aux-property trans) aux)
    trans))
(define-syntax retrieve-aux
  (lambda (x)
    (syntax-case x ()
      ((x id)
       (call-with-values (lambda () (syntax-local-binding #'id))
         (lambda (type val)
           (with-syntax ((aux (datum->syntax #'here
                                             (and (eq? type 'macro)
                                                  (aux-property val)))))
             #''aux)))))))
(define-syntax foo
  (with-aux 'bar
    (syntax-rules () ((_) 'foo))))
(foo)
⇒ foo
(retrieve-aux foo)
⇒ bar
```

`syntax-local-binding` must be called within the dynamic extent of a syntax transformer; to call it otherwise will signal an error.

`syntax-locally-bound-identifiers` *id* [Scheme Procedure]

Return a list of identifiers that were visible lexically when the identifier *id* was created, in order from outermost to innermost.

This procedure is intended to be used in specialized procedural macros, to provide a macro with the set of bound identifiers that the macro can reference.

As a technical implementation detail, the identifiers returned by `syntax-locally-bound-identifiers` will be anti-marked, like the syntax object that is given as input to a macro. This is to signal to the macro expander that these bindings were present in the original source, and do not need to be hygienically renamed, as would be the case with other introduced identifiers. See the discussion of hygiene in section 12.1 of the R6RS, for more information on marks.

```
(define (local-lexicals id)
  (filter (lambda (x)
            (eq? (syntax-local-binding x) 'lexical))
          (syntax-locally-bound-identifiers id)))
(define-syntax lexicals
  (lambda (x)
    (syntax-case x ()
      ((lexicals) #'(lexicals lexicals))
      ((lexicals scope)
       (with-syntax (((id ...) (local-lexicals #'scope)))
         #'(list (cons 'id id) ...))))))

(let* ((x 10) (x 20)) (lexicals))
⇒ ((x . 10) (x . 20))
```

6.10.5 Lisp-style Macro Definitions

The traditional way to define macros in Lisp is very similar to procedure definitions. The key differences are that the macro definition body should return a list that describes the transformed expression, and that the definition is marked as a macro definition (rather than a procedure definition) by the use of a different definition keyword: in Lisp, `defmacro` rather than `defun`, and in Scheme, `define-macro` rather than `define`.

Guile supports this style of macro definition using both `defmacro` and `define-macro`. The only difference between them is how the macro name and arguments are grouped together in the definition:

```
(defmacro name (args ...) body ...)
```

is the same as

```
(define-macro (name args ...) body ...)
```

The difference is analogous to the corresponding difference between Lisp's `defun` and Scheme's `define`.

Having read the previous section on `syntax-case`, it's probably clear that Guile actually implements defmacros in terms of `syntax-case`, applying the transformer on the expression between invocations of `syntax->datum` and `datum->syntax`. This realization leads us to the problem with defmacros, that they do not preserve referential transparency. One can be careful to not introduce bindings into expanded code, via liberal use of `gensym`, but there is no getting around the lack of referential transparency for free bindings in the macro itself.

Even a macro as simple as our `when` from before is difficult to get right:

```
(define-macro (when cond exp . rest)
  `(if ,cond
       (begin ,exp . ,rest)))
```

```
(when #f (display "Launching missiles!\n"))
⇒ #f

(let ((if list))
  (when #f (display "Launching missiles!\n")))
⊣ Launching missiles!
⇒ (#f #<unspecified>)
```

Guile's perspective is that defmacros have had a good run, but that modern macros should be written with `syntax-rules` or `syntax-case`. There are still many uses of defmacros within Guile itself, but we will be phasing them out over time. Of course we won't take away `defmacro` or `define-macro` themselves, as there is lots of code out there that uses them.

6.10.6 Identifier Macros

When the syntax expander sees a form in which the first element is a macro, the whole form gets passed to the macro's syntax transformer. One may visualize this as:

```
(define-syntax foo foo-transformer)
(foo arg...)
;; expands via
(foo-transformer #'(foo arg...))
```

If, on the other hand, a macro is referenced in some other part of a form, the syntax transformer is invoked with only the macro reference, not the whole form.

```
(define-syntax foo foo-transformer)
foo
;; expands via
(foo-transformer #'foo)
```

This allows bare identifier references to be replaced programmatically via a macro. `syntax-rules` provides some syntax to effect this transformation more easily.

`identifier-syntax` *exp* [Syntax]

Returns a macro transformer that will replace occurrences of the macro with *exp*.

For example, if you are importing external code written in terms of `fx+`, the fixnum addition operator, but Guile doesn't have `fx+`, you may use the following to replace `fx+` with `+`:

```
(define-syntax fx+ (identifier-syntax +))
```

There is also special support for recognizing identifiers on the left-hand side of a `set!` expression, as in the following:

```
(define-syntax foo foo-transformer)
(set! foo val)
;; expands via
(foo-transformer #'(set! foo val))
;; if foo-transformer is a "variable transformer"
```

As the example notes, the transformer procedure must be explicitly marked as being a "variable transformer", as most macros aren't written to discriminate on the form in the operator position.

`make-variable-transformer` *transformer* [Scheme Procedure]

Mark the *transformer* procedure as being a "variable transformer". In practice this means that, when bound to a syntactic keyword, it may detect references to that keyword on the left-hand-side of a `set!`.

```
(define bar 10)
(define-syntax bar-alias
  (make-variable-transformer
   (lambda (x)
     (syntax-case x (set!)
       ((set! var val) #'(set! bar val))
       ((var arg ...) #'(bar arg ...))
       (var (identifier? #'var) #'bar)))))

bar-alias ⇒ 10
(set! bar-alias 20)
bar ⇒ 20
(set! bar 30)
bar-alias ⇒ 30
```

There is an extension to identifier-syntax which allows it to handle the `set!` case as well:

`identifier-syntax` (*var exp1*) ((*set! var val*) *exp2*) [Syntax]

Create a variable transformer. The first clause is used for references to the variable in operator or operand position, and the second for appearances of the variable on the left-hand-side of an assignment.

For example, the previous `bar-alias` example could be expressed more succinctly like this:

```
(define-syntax bar-alias
  (identifier-syntax
    (var bar)
    ((set! var val) (set! bar val))))
```

As before, the templates in `identifier-syntax` forms do not need wrapping in `#'` syntax forms.

6.10.7 Syntax Parameters

Syntax parameters[7] are a mechanism for rebinding a macro definition within the dynamic extent of a macro expansion. This provides a convenient solution to one of the most common types of unhygienic macro: those that introduce a unhygienic binding each time the macro is used. Examples include a `lambda` form with a `return` keyword, or class macros that introduce a special `self` binding.

[7] Described in the paper *Keeping it Clean with Syntax Parameters* by Barzilay, Culpepper and Flatt.

With syntax parameters, instead of introducing the binding unhygienically each time, we instead create one binding for the keyword, which we can then adjust later when we want the keyword to have a different meaning. As no new bindings are introduced, hygiene is preserved. This is similar to the dynamic binding mechanisms we have at run-time (see Section 7.5.27 [SRFI-39], page 631), except that the dynamic binding only occurs during macro expansion. The code after macro expansion remains lexically scoped.

`define-syntax-parameter` *keyword transformer* [Syntax]

Binds *keyword* to the value obtained by evaluating *transformer*. The *transformer* provides the default expansion for the syntax parameter, and in the absence of `syntax-parameterize`, is functionally equivalent to `define-syntax`. Usually, you will just want to have the *transformer* throw a syntax error indicating that the *keyword* is supposed to be used in conjunction with another macro, for example:

```
(define-syntax-parameter return
  (lambda (stx)
    (syntax-violation 'return "return used outside of a lambda^" stx)))
```

`syntax-parameterize` ((*keyword transformer*) ...) *exp* ... [Syntax]

Adjusts *keyword* ... to use the values obtained by evaluating their *transformer* ..., in the expansion of the *exp* ... forms. Each *keyword* must be bound to a syntax-parameter. `syntax-parameterize` differs from `let-syntax`, in that the binding is not shadowed, but adjusted, and so uses of the keyword in the expansion of *exp* ... use the new transformers. This is somewhat similar to how `parameterize` adjusts the values of regular parameters, rather than creating new bindings.

```
(define-syntax lambda^
  (syntax-rules ()
    [(lambda^ argument-list body body* ...)
     (lambda argument-list
       (call-with-current-continuation
        (lambda (escape)
          ;; In the body we adjust the 'return' keyword so that calls
          ;; to 'return' are replaced with calls to the escape
          ;; continuation.
          (syntax-parameterize ([return (syntax-rules ()
                                          [(return vals (... ...))
                                           (escape vals (... ...))])])
            body body* ...))))]))

;; Now we can write functions that return early.  Here, 'product' will
;; return immediately if it sees any 0 element.
(define product
  (lambda^ (list)
           (fold (lambda (n o)
                   (if (zero? n)
                       (return 0)
                       (* n o)))
                 1
```

```
                    list)))
```

6.10.8 Eval-when

As `syntax-case` macros have the whole power of Scheme available to them, they present a problem regarding time: when a macro runs, what parts of the program are available for the macro to use?

The default answer to this question is that when you import a module (via `define-module` or `use-modules`), that module will be loaded up at expansion-time, as well as at run-time. Additionally, top-level syntactic definitions within one compilation unit made by `define-syntax` are also evaluated at expansion time, in the order that they appear in the compilation unit (file).

But if a syntactic definition needs to call out to a normal procedure at expansion-time, it might well need need special declarations to indicate that the procedure should be made available at expansion-time.

For example, the following code will work at a REPL, but not in a file:

```
;; incorrect
(use-modules (srfi srfi-19))
(define (date) (date->string (current-date)))
(define-syntax %date (identifier-syntax (date)))
(define *compilation-date* %date)
```

It works at a REPL because the expressions are evaluated one-by-one, in order, but if placed in a file, the expressions are expanded one-by-one, but not evaluated until the compiled file is loaded.

The fix is to use `eval-when`.

```
;; correct: using eval-when
(use-modules (srfi srfi-19))
(eval-when (expand load eval)
  (define (date) (date->string (current-date))))
(define-syntax %date (identifier-syntax (date)))
(define *compilation-date* %date)
```

`eval-when` *conditions exp...* [Syntax]

Evaluate *exp...* under the given *conditions*. Valid conditions include:

`expand` Evaluate during macro expansion, whether compiling or not.

`load` Evaluate during the evaluation phase of compiled code, e.g. when loading a compiled module or running compiled code at the REPL.

`eval` Evaluate during the evaluation phase of non-compiled code.

`compile` Evaluate during macro expansion, but only when compiling.

In other words, when using the primitive evaluator, `eval-when` expressions with `expand` are run during macro expansion, and those with `eval` are run during the evaluation phase.

When using the compiler, `eval-when` expressions with either `expand` or `compile` are run during macro expansion, and those with `load` are run during the evaluation phase.

When in doubt, use the three conditions `(expand load eval)`, as in the example above. Other uses of `eval-when` may void your warranty or poison your cat.

6.10.9 Macro Expansion

Usually, macros are expanded on behalf of the user as needed. Macro expansion is an integral part of `eval` and `compile`. Users can also expand macros at the REPL prompt via the `expand` REPL command; See Section 4.4.4.4 [Compile Commands], page 51.

Macros can also be expanded programmatically, via `macroexpand`, but the details get a bit hairy for two reasons.

The first complication is that the result of macro-expansion isn't Scheme: it's Tree-IL, Guile's high-level intermediate language. See Section 9.4.3 [Tree-IL], page 859. As "hygienic macros" can produce identifiers that are distinct but have the same name, the output format needs to be able to represent distinctions between variable identities and names. Again, See Section 9.4.3 [Tree-IL], page 859, for all the details. The easiest thing is to just run `tree-il->scheme` on the result of macro-expansion:

```
(macroexpand '(+ 1 2))
⇒
#<tree-il (call (toplevel +) (const 1) (const 2))>

(use-modules (language tree-il))
(tree-il->scheme (macroexpand '(+ 1 2)))
⇒
(+ 1 2)
```

The second complication involves `eval-when`. As an example, what would it mean to macro-expand the definition of a macro?

```
(macroexpand '(define-syntax qux (identifier-syntax 'bar)))
⇒
?
```

The answer is that it depends who is macro-expanding, and why. Do you define the macro in the current environment? Residualize a macro definition? Both? Neither? The default is to expand in "eval" mode, which means an `eval-when` clauses will only proceed when `eval` (or `expand`) is in its condition set. Top-level macros will be `eval`'d in the top-level environment.

In this way `(macroexpand` *foo*`)` is equivalent to `(macroexpand` *foo* `'e '(eval))`. The second argument is the mode (`'e` for "eval") and the third is the eval-syntax-expanders-when parameter (only `eval` in this default setting).

But if you are compiling the macro definition, probably you want to reify the macro definition itself. In that case you pass `'c` as the second argument to `macroexpand`. But probably you want the macro definition to be present at compile time as well, so you pass `'(compile load eval)` as the *esew* parameter. In fact `(compile` *foo* `#:to 'tree-il)` is entirely equivalent to `(macroexpand` *foo* `'c '(compile load eval))`; See Section 9.4.2 [The Scheme Compiler], page 858.

It's a terrible interface; we know. The macroexpander is somewhat tricksy regarding modes, so unless you are building a macro-expanding tool, we suggest to avoid invoking it directly.

6.10.10 Hygiene and the Top-Level

Consider the following macro.

```
(define-syntax-rule (defconst name val)
  (begin
    (define t val)
    (define-syntax-rule (name) t)))
```

If we use it to make a couple of bindings:

```
(defconst foo 42)
(defconst bar 37)
```

The expansion would look something like this:

```
(begin
  (define t 42)
  (define-syntax-rule (foo) t))
(begin
  (define t 37)
  (define-syntax-rule (bar) t))
```

As the two `t` bindings were introduced by the macro, they should be introduced hygienically – and indeed they are, inside a lexical contour (a `let` or some other lexical scope). The `t` reference in `foo` is distinct to the reference in `bar`.

At the top-level things are more complicated. Before Guile 2.2, a use of `defconst` at the top-level would not introduce a fresh binding for `t`. This was consistent with a weaselly interpretation of the Scheme standard, in which all possible bindings may be assumed to exist, at the top-level, and in which we merely take advantage of toplevel `define` of an existing binding being equivalent to `set!`. But it's not a good reason.

The solution is to create fresh names for all bindings introduced by macros – not just bindings in lexical contours, but also bindings introduced at the top-level.

However, the obvious strategy of just giving random names to introduced toplevel identifiers poses a problem for separate compilation. Consider without loss of generality a `defconst` of `foo` in module `a` that introduces the fresh top-level name `t-1`. If we then compile a module `b` that uses `foo`, there is now a reference to `t-1` in module `b`. If module `a` is then expanded again, for whatever reason, for example in a simple recompilation, the introduced `t` gets a fresh name; say, `t-2`. Now module `b` has broken because module `a` no longer has a binding for `t-1`.

If introduced top-level identifiers "escape" a module, in whatever way, they then form part of the binary interface (ABI) of a module. It is unacceptable from an engineering point of view to allow the ABI to change randomly. (It also poses practical problems in meeting the recompilation conditions of the Lesser GPL license, for such modules.) For this reason many people prefer to never use identifier-introducing macros at the top-level, instead making those macros receive the names for their introduced identifiers as part of their arguments, or to construct them programmatically and use `datum->syntax`. But this approach requires omniscience as to the implementation of all macros one might use, and also limits the expressive power of Scheme macros.

There is no perfect solution to this issue. Guile does a terrible thing here. When it goes to introduce a top-level identifier, Guile gives the identifier a pseudo-fresh name: a name

that depends on the hash of the source expression in which the name occurs. The result in this case is that the introduced definitions expand as:

```
(begin
  (define t-1dc5e42de7c1050c 42)
  (define-syntax-rule (foo) t-1dc5e42de7c1050c))
(begin
  (define t-10cb8ce9fdddd6e9 37)
  (define-syntax-rule (bar) t-10cb8ce9fdddd6e9))
```

However, note that as the hash depends solely on the expression introducing the definition, we also have:

```
(defconst baz 42)
⇒ (begin
    (define t-1dc5e42de7c1050c 42)
    (define-syntax-rule (baz) t-1dc5e42de7c1050c))
```

Note that the introduced binding has the same name! This is because the source expression, `(define t 42)`, was the same. Probably you will never see an error in this area, but it is important to understand the components of the interface of a module, and that interface may include macro-introduced identifiers.

6.10.11 Internal Macros

`make-syntax-transformer` *name type binding* [Scheme Procedure]

Construct a syntax transformer object. This is part of Guile's low-level support for syntax-case.

`macro?` *obj* [Scheme Procedure]
`scm_macro_p` (*obj*) [C Function]

Return `#t` if *obj* is a syntax transformer, or `#f` otherwise.

Note that it's a bit difficult to actually get a macro as a first-class object; simply naming it (like `case`) will produce a syntax error. But it is possible to get these objects using `module-ref`:

```
(macro? (module-ref (current-module) 'case))
⇒ #t
```

`macro-type` *m* [Scheme Procedure]
`scm_macro_type` (*m*) [C Function]

Return the *type* that was given when *m* was constructed, via `make-syntax-transformer`.

`macro-name` *m* [Scheme Procedure]
`scm_macro_name` (*m*) [C Function]

Return the name of the macro *m*.

`macro-binding` *m* [Scheme Procedure]
`scm_macro_binding` (*m*) [C Function]

Return the binding of the macro *m*.

`macro-transformer` *m* [Scheme Procedure]
`scm_macro_transformer` (*m*) [C Function]

Return the transformer of the macro *m*. This will return a procedure, for which one may ask the docstring. That's the whole reason this section is documented. Actually a part of the result of `macro-binding`.

6.11 General Utility Functions

This chapter contains information about procedures which are not cleanly tied to a specific data type. Because of their wide range of applications, they are collected in a *utility* chapter.

6.11.1 Equality

There are three kinds of core equality predicates in Scheme, described below. The same kinds of comparisons arise in other functions, like `memq` and friends (see Section 6.6.9.7 [List Searching], page 185).

For all three tests, objects of different types are never equal. So for instance a list and a vector are not `equal?`, even if their contents are the same. Exact and inexact numbers are considered different types too, and are hence not equal even if their values are the same.

`eq?` tests just for the same object (essentially a pointer comparison). This is fast, and can be used when searching for a particular object, or when working with symbols or keywords (which are always unique objects).

`eqv?` extends `eq?` to look at the value of numbers and characters. It can for instance be used somewhat like `=` (see Section 6.6.2.8 [Comparison], page 117) but without an error if one operand isn't a number.

`equal?` goes further, it looks (recursively) into the contents of lists, vectors, etc. This is good for instance on lists that have been read or calculated in various places and are the same, just not made up of the same pairs. Such lists look the same (when printed), and `equal?` will consider them the same.

`eq?` *x y* [Scheme Procedure]
`scm_eq_p` (*x*, *y*) [C Function]

Return `#t` if *x* and *y* are the same object, except for numbers and characters. For example,

```
(define x (vector 1 2 3))
(define y (vector 1 2 3))

(eq? x x)  ⇒ #t
(eq? x y)  ⇒ #f
```

Numbers and characters are not equal to any other object, but the problem is they're not necessarily `eq?` to themselves either. This is even so when the number comes directly from a variable,

```
(let ((n (+ 2 3)))
  (eq? n n))       ⇒ *unspecified*
```

Generally `eqv?` below should be used when comparing numbers or characters. `=` (see Section 6.6.2.8 [Comparison], page 117) or `char=?` (see Section 6.6.3 [Characters], page 129) can be used too.