Using mkmac

by

Eugene Kohlbecker

Computer Science Department
Indiana University
Bloomington, IN 47405

TECHNICAL REPORT NO. 157

Using mkmac

May, 1984

by

Eugene Kohlbecker

This material is based on work supported by the National Science Foundation under grant MCS 83-04567.

23 May 1984

Using mkmac

Eugene Kohlbecker

Computer Science Department
Lindley Hall 101
Indiana University
Bloomington, Indiana 47405

Abstract

This paper is a reference guide to using the Scheme 84 macro writing tool mkmac. Specific rules about the syntax and semantics of mkmac-expressions are presented along with many illustrative examples.

Writing macros in Scheme or Lisp is a two-step process. The first step is to design the macro's *expansion*, a Scheme or Lisp encoding of what the macro is to do at run-time. This is the creative step, the one in which the macro-writer is called upon to be innovative, to design something new. In composing macros for an assembly language this is usually the only thing that needs to be done, but in writing a macro for a lisp-like language, a second step must be performed. The user writes a program that transforms expressions matching a given pattern into expressions conforming to the structure of the previously designed expansion. Such a program is called a *syntactic transform*.

In most cases the second step can be done by machine; in Scheme 84 [Friedman, *et al.* 1984] there is a system program **mkmac** which creates syntactic transforms. **mkmac** itself is a Scheme macro that expects two arguments, a pattern specification and an expansion specification. Each specification is written in a language peculiar to **mkmac**, and each language is an extension of Scheme 84. The goal of this paper is to acquaint the reader with those extensions and demonstrate their use.

For defining simple macros, unextended Scheme 84 serves as the language for both the pattern and the expansion specifications. For example, a **while** macro that expands to a standard while-loop is specified as

```
(mkmac (while predicate body)
       (let ([p (lambda () predicate)]
             [b (lambda () body)])
            ((rec loop (lambda () (if (p)
                                      (begin (b) (loop))))))))).
```

This material is based on work supported by the National Science Foundation under grant MCS 83-04567.

mkmac produces a function, the syntactic transform, of one argument which will, when given a **while**-expression, change it into the corresponding **let**-expression. Since the transformation and substitution of the resulting code take place at compilation time, calling the **while** macro has the same run-time effect as using the **let**-expression.

mkmac constructs the desired function and enters it into the Scheme 84 macro table just as if the transform had been defined by hand. If one wishes to examine the code that is actually generated, the macro **show-mkmac-code** can be used. The code displayed is always that of a function of the single argument ***pattern***. Furthermore, if instead of having the transform entered into the macro table, the user wants to use it in some other context, the macro **syntrans** returns the appropriate closure.

An example of the use of **show-mkmac-code** is

```
>>> (show-mkmac-code
       (while predicate body)
       (let ([p (lambda () predicate)]
             [b (lambda () body)])
            ((rec loop (lambda () (if (p)
                                      (begin (b) (loop))))))))
(lambda (*pattern*)
  (list 'let
        (list (list 'p (list 'lambda nil (cadr *pattern*)))
              (list 'b (list 'lambda nil (caddr *pattern*))))
        '((rec loop (lambda nil
                      (if (p) (begin (b) (loop))))))))
while
```

show-mkmac-code only displays the code; it returns the name of the macro but does not define it.

Those familiar with the backquote facility might define the **while** macro as

```
(macro while
   (lambda (*pattern*)
      `(let ([p (lambda () ,(cadr *pattern*))]
             [b (lambda () ,(caddr *pattern*))])
            ((rec loop (lambda () (if (p)
                                      (begin (b) (loop))))))))).
```

A basic premise behind **mkmac** is that commas and backquotes are unnecessary; macros can be defined in a more cleanly fashion, without the confusion of quasi-quoted text. If a user specifies a pattern and expansion, then, during the construction of the transform, **mkmac** is capable of determining which components of the expansion were named in the pattern. It replaces those components with appropriate list-structure references in the form of **car-cdr** chains.

Users often find macros with single-expression bodies like **while** restrictive; they would like to have a loop body that is longer than one expression. Loops in the form

```
(while p b1 b2 ...)
```

are typical. **mkmac** provides a means of expressing such macros—the ellipsis notation. Here is the **mkmac** version of the extended-body **while**-loop.

```
(mkmac (while predicate body1 ...)
       (let ([p (lambda () predicate)]
             [b (lambda () body1 ...)])
            ((rec loop (lambda ()
                         (if (p)
                             (begin (b) (loop))))))))

>>> (let ([n 5])
        (while (> n 0)
               (print n)
               (set! n (sub1 n))))
54321nil
```

This version allows one predicate expression and any finite number of expressions in the body of the loop.

1. The Pattern Specification Language

Patterns are specified in Scheme 84 extended with the single symbol "...", the ellipsis.

Because one of the principal uses of macros is defining an operation on an arbitrary number of terms, there must be a way of indicating that a list may be of any finite length. The ellipsis serves this purpose. The pattern for the **and** macro is

```
(and e1 ...).
```

Any number of expressions may follow the keyword **and** in some actual call to that macro. A non-operational syntax that denotes the same construction is used by Sussman and Steele in their Scheme papers [Steele and Sussman]. Their technique to represent the pattern for **and** is to write

```
(and . body).
```

The dotted-pair notation indicates that any number of expressions can appear in the cdr of the macro call.

The ellipsis may be used any number of times within a single pattern. However, each time it appears it must be the last item in the immediately enclosing list. Thus

```
(foo (a ...) (b ...))
```

is legal in the pattern specification language, but

```
(foo a ... b ...)
```

is not.

The term that appears to the left of the ellipsis is the *prototype*. Hence the significance of a pattern list containing an ellipsis is that the list described is one

of finite length, comprised of initial terms that match the specified items and an arbitrary number of terms that match the form of the prototype. For example, the pattern specification

```
(a b (c d) ...)
```

denotes a list of at least two elements. The first is named `a`, the second `b`, and if any other elements appear, they themselves will be lists of two elements, the first component called `c` and the second `d`.

Lists containing an ellipsis may be nested within other lists, as in the standard pattern specification of the macro `let`

```
(let ([i e] ...) b ...).
```

A `let`-expression is composed of two or more parts; its second element is itself an arbitrarily long list made of lists of two terms. Furthermore, because `mkmac` provides a naming of the parts of a pattern, whatever `i`, `e`, and `b` ultimately represent can be of any form. They do not have to be atomic terms.

This leads to the second major principle in the design of `mkmac`. Any part of the pattern may be given a name, but if one desires to refer to a sub-part, then its name must be explicitly specified. If the pattern specification for `let` were

```
(let (d ...) b ...),
```

it would not be possible to refer to the components that make up `d`, whatever they are in actual use. It is not possible to name both a part and its pieces.

Some critics of `mkmac` have indicated that it is also desirable to specify the tail of an arbitrarily long list as well as its head. A future version of `mkmac` may allow this. But for now, it is not possible to specify patterns such as

```
(a b ... c d)
```

where the list is of length greater than or equal to three, has arbitrarily many terms like `b`, and ends with terms named `c` and `d`.

In summary, the pattern specification language is Scheme 84 with the ellipsis. There are two important rules to remember in writing patterns:

- The ellipsis must appear as the last item in any list in which it is used.
- Names must be given to all parts of the pattern to which it is necessary to refer.

2. The Expansion Specification Language

As for the pattern specifications, Scheme 84 serves as the basis of the expansion specification language. However, it is extended not only with an ellipsis but also with a keyword `*mkmac-symbol*` that enables the generation of expansion-time symbols, a lexical definition keyword `with`, and three conditional expansion constructions, `on-num-terms`, `by-cases`, and `on-own-cases`.

1. The ellipsis in expansion specifications. The use of the ellipsis "..." varies slightly from the way it is used in the pattern language. Here the restriction on the ellipsis appearing last in any list is removed; terms in an expansion list may be placed after the ellipsis. However, its interpretation is similar. The appearance

of the ellipsis-list `(a ...)` in an expansion specification means that `mkmac` will generate code to construct a list of terms in the same form as the prototype `a`. The list, which will be generated at macro-expansion time, will have the same length as the shortest ellipsis-list in the pattern containing prototype-terms that match the expansion prototype. Here is an example; `trip-up` takes three lists and produces a single list with respective elements associated.

```
(mkmac (trip-up (a ...) (b ...) (c ...))
       (list '(a b c) ...))
```

Examples of its use are

```
>>> (trip-up (a b c) (1 2 3) (x y z))
((a 1 x) (b 2 y) (c 3 z))

>>> (trip-up (a b c) (1 2 3) (x y))
((a 1 x) (b 2 y))
```

Another pair of macros forms the Cartesian product of two lists:

```
(mkmac (horse a (b ...))
       (list '(a b) ...))

(mkmac (cart (a ...) list2)
       (append (horse a list2) ...))

>>> (cart (x y z) (1 2 3))
((x 1) (x 2) (x 3) (y 1) (y 2) (y 3) (z 1) (z 2) (z 3))
```

Both macros are needed because all terms that are members of any pattern prototype vary together in the construction of an expansion ellipsis-list.

It was indicated above that in an expansion specification, the ellipsis does not have to be the last item in a list. This is useful in constructing a macro such as `double` that produces a list containing two copies of an original list:

```
(mkmac (double (a ...))
       (list 'a ... 'a ...))

>>> (double (1 2 3 4))
(1 2 3 4 1 2 3 4)
```

Even though the ellipsis has great intuitive meaning, its significance within `mkmac` expressions depends on both the pattern and expansion specifications. If there is no pattern specification prototype or combination of prototypes that corresponds to a given expansion prototype, then `mkmac` aborts in an error state.

As a final example before considering the use of the special expansion keywords, here is the complete definition of a version of `let` extended with implicit `begin`-blocks in the definition list.

```
(mkmac (xlet ([i e ...] ...) b ...)
       ((lambda (i ...) b ...) (begin e ...) ...))
```

2. Generated symbols. Sometimes one wants to have a unique set of identifiers generated at macro-expansion time, identifiers unknown at macro-definition time. A good example of this is the macro **parset!**, a parallel assignment operation. The pattern is easy to specify as

```
(parset! (i e) ...).
```

This macro represents the operation of evaluating all expressions `e` within the current environment before changing the bindings of any of the identifiers `i`. The new bindings are effected only after all the expressions are evaluated. To define this macro in Scheme 84 we must use some auxiliary identifiers that are distinct from the `i`'s and not among the free identifiers appearing in the the `e`'s. For example, one might define the expansion as

```
(let ([foo1 e] ...)
     (set! i foo1) ...))
```

intending the set of identifiers {`foo1`, ...} to be used as auxiliary names in some run-time expansion of a call to **parset!**.

However, if this specification is given to **mkmac**, it will produce code that generates pairs "`[foo1 e]`" where the `e` varies but the `foo1` appears as "`foo1`" in every term. The `foo1` is treated as a constant in the prototype; it is the only auxiliary. One really wants to have a number of special symbols that vary from term to term in an ellipsis-list generated macro. The **mkmac** facility for doing this is the reserved identifier `*mkmac-symbol*`; it stands for an arbitrary, expansion-time generated identifier. The correct way of writing the expansion specification for **parset!** is

```
(let ([*mkmac-symbol* e] ...)
     (set! i *mkmac-symbol*) ...)).
```

Each occurrence of `*mkmac-symbol*` *within an ellipsis list prototype* instructs **mkmac** to generate code that will produce identifiers that vary from term to term in the generated ellipsis list. **mkmac** allows one such set of identifiers per macro; the second or any subsequent appearance of `*mkmac-symbol*` generates the same set, in the same order.

The mechanism that produces this behavior involves the binding of four identifiers that become local to the resulting macro definition. These are `↑mkgen↑`, `↑mysyms↑`, `↑mkflag↑`, and `↑mkcount↑`. The first represents a function of zero arguments that generates the auxiliary identifiers; the second, a list of generated symbols; the third, a flag indicating that symbols have been generated; and the fourth, a counter that keeps track of how many identifiers have been used for a specific ellipsis list. The counter is reset after each ellipsis-list expansion. Only the most adventurous macro-definers will use these identifiers in their expansion specifications involving `*mkmac-symbol*`.

The generated identifiers are produced with **gensym**, insuring that unexpected lexical clashes will not occur. Thus, users of macros defined with `*mkmac-symbol*` need not be concerned that the free variables in their macro calls might be inadvertently caught by one of the generated symbols.

In summary, the reserved identifier ***mkmac-symbol*** may appear once in each ellipsis list prototype. Its presence indicates that a set of identifiers is to be generated at expansion-time; that each member of the set is to be used in one term of the ellipsis-list generation; and that should ***mkmac-symbol*** be used in another ellipsis list prototype, the same set of identifiers will be used again. Even though ***mkmac-symbol*** can be used outside of a prototype, we do not recommend that users adopt this habit. If it is known that one needs only a single special identifier, a user can name it himself instead of paying the expensive cost of obtaining a **mkmac**-generated name.

To finish this section, we present an example of the use of **parset!**.

```
>>> (let ([a 1] [b 2])
         (writeln a b)
         (parset! (a b) (b a))
         (writeln a b))
12
21
nil
```

3. Lexical definitions—Using **with** *and* **withrec**. Occasionally it is desirable to define special help functions and data that are local to a macro, not accessible from outside of the code represented by the expansion. **mkmac** provides this capability with two forms for the declaration of constants, **with** and **withrec**. These constants are true compile-time constants; their values are determined at macro-expansion time. Consider the example **wl**, a macro version of **writeln** with a counter:

```
(mkmac (wl e1 ...)
       (withrec ([printall (lambda (l)
                             (cond [(null? l) (newline)]
                                   [t (print (car l))
                                      (print " ")
                                      (set! count (add1 count))
                                      (printall (cdr l))]))]
                 [count 0])
           (begin (printall (list e1 ...))
                  (print count)
                  (print " items")
                  (newline))))
```

This does the expected in printing the evaluated arguments to **wl**, but there is a surprise involving **count**:

```
>>> (wl (add1 3) (sub1 4))
4 3
0 items
nil
```

```
>>> (wl 10 20 30)
10 20 30
0 items
nil
```

A counter that does keep track of the the number of items printed can be created by making it local to `printall` and providing some means to access it.

The syntax for both `with` and `withrec` expansions is similar to that of `let` and `letrec` restricted to having only one expression in the body of the expansion. The keyword `with` is followed by a list of definitions. In a `with` expansion, the definitions are scoped in the same fashion as in a `let` definition list. In a `withrec` expansion, the scoping is similar to that of the standard Scheme 84 `letrec` expression. In short, `with` provides for non-recursive compile-time constants, and `withrec` provides recursive ones. Examination of generated code with `show-mkmac-code` will provide the user with full scoping details. Here is the code for `wl`.

```
(lambda (*pattern*)
  (letrec ((printall
            (lambda (l)
              (cond ((null? l) (newline))
                    (t (print (car l))
                       (print " ")
                       (set! count (add1 count))
                       (printall (cdr l))))))
           (count 0))
          (list 'begin
                (list printall (cons 'list (cdr *pattern*)))
                (list 'print count)
                '(print " items")
                '(newline))))
```

The definitions do not become part of the expansion. Instead, they are part of the environment in existence at the time the macro expansion is produced. They are within the scope of the expansion-time binding of `*pattern*`.

Each definition created by these forms is given as an identifier-value pair. Neither the identifiers nor their corresponding values are processed by `mkmac` in any way. The code written by the user is lifted as is, and placed within the transform. A user cannot refer to any part of the pattern specification within a `with` or `withrec` definition list. Of course, the body of the expansion is processed to obtain a transform described in terms of the pattern specification. As a general rule, `with` and `withrec` should only appear at the outermost layers of an expansion specification.

4. Conditional expansion. Frequently macros are defined so that expansions vary, depending on certain conditions. The conditional expansions in `mkmac` code allow the user to specify these variations on the basis of the structure of the actual macro call.

The pattern specification for the standard Scheme macro and is

```
(and e1 e2 ...).
```

But there are at least three possible expansions based on the number of terms that follow the keyword **and**. If there are no terms, that is, if the macro call appears as **(and)**, then the expansion is defined to be **t**. If there is one term **e1**, the expansion is **e1**. If there are more terms, the expansion is a Scheme conditional expression. All of this is expressed in the **mkmac** expansion language as

```
(on-num-terms
    [0 t]
    [1 e1]
    [t (if e1 (and e2 ...))])
```

The expansion can contain a recursive call to the macro being defined as long as that call has its arguments reduced in some way. Here is the macro **or**.

```
(mkmac (or e1 e2 ...)
       (on-num-terms
           [0 nil]
           [1 e1]
           [t (let ([*00000 e1])
                (if *00000 *00000 (or e2 ...)))]))
```

The conditional forms **on-num-terms**, **by-cases**, and **on-own-cases** all have roughly the same behavior. Each is composed of the keyword followed by any number of two-element *conditional clauses*. The left-hand side of each clause describes some condition relating to the pattern. During macro expansion, the conditions are tested in order, and the right-hand expansion corresponding to the first true condition is produced. If no condition is true, the expansion is **nil**. Right-hand sides are always descriptors of the code to be generated and included in the transform.

Conditional expansions may appear by themselves or as the body of a **with** or **withrec** expansion specification. They may be nested; it is legal to use a conditional expansion anywhere any other expansion specification may be placed. In particular, one may use a conditional expansion in the right-hand side of another conditional expansion clause. However, because the left-hand sides of conditional expansions are not themselves complete expansion specifications; the conditional expansion keywords have no special meaning in a left-hand side. See the description of the standard Lisp **cond** macro in the Additional Examples section of this paper.

Furthermore, paralleling its traditional Lisp **cond**-clause use, the symbol **t** is used as the default case tag. But in **mkmac** expansions, **t** must be the default symbol, nothing else will work.

Examples of the use of **on-num-terms** have already been presented. The number that appears as the left member of a conditional expansion clause describes the number of terms that appear after the keyword.

The **by-cases** conditional allows the user to specify left-hand sides of conditional clauses that refer to the entire pattern. These left-hand sides are incorporated into the syntactic transform as is, unaltered by **mkmac**. Because the left sides are

not processed, it is impossible to refer to the named parts of the pattern within them. Users can refer to the entire pattern by using the identifier ***pattern*** and to parts of it by appropriate **car-cdr** chains. As an example, the standard macro **let*** is defined

```
(mkmac (let* ([i1 e1] [i2 e2] ...) b ...)
       (by-cases
           [(null? (cadr *pattern*)) (begin b ...)]
           [t (let ([i1 e1])
                   (let* ([i2 e2] ...) b ...))])).
```

Finally, **on-own-cases** allows the left side of conditional clauses to be processed by **mkmac** like right-hand sides are processed. This permits using the named pieces of the expression given in the pattern specification. This form differs from **by-cases** because the identifier ***pattern*** is not recognized as referring to the pattern specification.

Assuming that macros **until** and **while** exist, a general purpose repetition macro can be written to illustrate **on-own-cases**:

```
(mkmac (repeat tag test body)
       (on-own-cases
            [(eq? tag 'while) (while test body)]
            [(eq? tag 'until) (until body test)]))

>>> (let ([n 10])
        (repeat while
           (> n 0)
           (begin (print '*)
                  (set! n (subi n)))))
**********nil
```

Another example is that of **setf!**, a construction used in MacLisp to achieve a type of call-by-name functionality [Pitman 1980]. The pattern is

```
(setf! (fname form) value),
```

and whatever expansion is generated depends upon the particular **fname**.

```
(on-own-cases
     [(eq? fname 'car) (set-car! form value)]
     [(eq? fname 'cadr) (set-car! (cdr form) value)]
     [(eq? fname 'cdr) (set-cdr! form value)])
```

It is possible to use the ellipsis on the left-side of an **on-own-cases** conditional clause. However, **mkmac** does not perform any restructuring of ellipsis-lists there. That is, the ellipsis list from the pattern

```
(foo (a ...)  b)
```

can be referred to as **(a ...)**, but it is not possible to build a list such as

```
((a b) ...)
```

```
expansion-specification ::= one-line-expansion
    | (with (lexical-definitions) expansion-specification)
    | (withrec (lexical-definitions) expansion-specification)
    | (conditional-keyword {conditional-clause})

lexical-definitions ::= {(identifier value-expression)}

conditional-clause ::= (left-hand-side expansion-specification)

value-expression ::= a standard Scheme 84 expression
```

Figure 1. Expansion specifications.

as part of a conditional test. The `on-own-cases` facility is only designed to allow references to existing parts of the pattern structure.

In summary, the expansion specification language is Scheme 84 extended with the ellipsis, an expansion-time symbol generator, `with` and `withrec` lexical definition expansions, and three varieties of conditional expansions. Figure 1 gives a Backus-Naur form grammar of expansion specifications describing the interplay between the various forms of expansion specification. A "one-line-expansion" is any properly formed expansion that does not use any of the special **mkmac** keywords. It may include uses of the ellipsis and the identifier `*mkmac-symbol*`. Braces surrounding a non-terminal symbol indicate zero or more occurrences of that symbol. In the expansion language

- The ellipsis may appear any number of times in a given list provided each occurrence follows a prototype expression. It may appear in any portion of the expansion specification that is processed by **mkmac**.
- Expansion-time symbols may be produced with the special identifier `*mkmac-symbol*`. It is proper to use this keyword only within the prototype of an ellipsis list. Each occurrence of it refers to the same set of symbols generated at macro-expansion time.
- `with` and `withrec` expansions may be used to bind macro-expansion time identifiers. They may be used anywhere an expansion specification can appear.
- Conditional expansions may appear as the body of a `with` or `withrec` expansion or as the entire expansion. They may also be used inside each other as part of the right-hand side of a conditional clause.

3. NESTING mkmac

Experienced users of **mkmac** usually decide that they want to use it to define macro-generating macros. They want to use an entire **mkmac** expression as the expansion specification of another macro. This is possible, but the user should be aware that once he writes a call to **mkmac** in an expansion specification, the ordinary nesting of `with`, `withrec`, and conditional expansions no longer takes place. Within **mkmac** calls in an expansion specification, the keywords have no special meaning.

Here is an example of a macro-defining macro `alpha`, one that enables the user to declare a list of identifiers that are to be treated as unique names within the

expansion. The purpose of such identifiers is to avoid the capturing of free variables in user calls to the macro defined with alpha.

```
(mkmac (alpha p e (f ...))
       (mkmac p
              (with ([f (gensym '*)] ...)
                    e)))

(alpha (or e1 e2 ...)
       (on-num-terms
          [0 nil]
          [1 e1]
          [t (let ([v e1])
                  (if v v (or e2 ...)))])
       (v))
```

If the interpretation with as a special keyword were not suppressed during the definition of alpha, then two things would go wrong. First, the reference made by the identifier f and the ellipsis to the pattern specification would not be understood, and second, even if they were picked up on, the f would become a compile-time identifier, used during the expansion of a macro defined with alpha. Fortunately, mkmac is aware of all this; once a nested call to mkmac is detected, it no longer interprets its own keywords as special instructions.

4. Additional Examples

This section presents a number of standard Scheme 84 macros using mkmac. Unless otherwise indicated, the semantics of the macros defined here is equivalent to the semantics of the macros in the system. Definitions of standard macros not described here may be found in the Scheme 84 Reference Manual [Friedman, *et al.*]. In most cases when a macro definition is given in both places, the two definitions are different while still describing equivalent run-time behaviors.

```
(mkmac (step ((id init update) ...) (test exp ...) body ...)
       (with ([v (gensym '*)])
             (let ([id init] ...)
                ((rec v
                    (lambda ()
                       (if test
                          (begin exp ...)
                          (begin
                             body ...
                             (set! id (update id)) ...
                             (v)))))))))
```

```
(mkmac (case exp [t1 e1 ...] [t2 e2 ...] ...)
       (with ([tag (gensym '*)])
             (by-cases
                [(null? (cddr *pattern*)) (error)] ; some suitable error
                [(and (null? (cdddr *pattern*))    ;   reporting function
                      (eq? (car (caddr *pattern*)) 'else))
                 (begin exp e1 ...)]
                [t (let ([tag exp])
                     (if (eq? tag 't1)
                         (begin e1 ...)
                         (case tag [t2 e2 ...] ...)))])))
```

```
(mkmac (cond [i1 e1 ...] [i2 e2 ...] ...)
       (with ([v (gensym '*)])
             (by-cases
                [(null? (cdr *pattern*)) nil]
                [(null? (cddr *pattern*))
                 (by-cases
                    [(eq? (caadr *pattern*) t)
                     (by-cases
                        [(null? (cdadr *pattern*)) t]
                        [t (begin e1 ...)])]
                    [(null? (caadr *pattern*)) nil]
                    [(null? (cdadr *pattern*))
                     ((lambda (v) (if v v)) i1)]
                    [t (if i1 (begin e1 ...))])]
                [(null? (caadr *pattern*)) (cond [i2 e2 ...] ...)]
                [(null? (cdadr *pattern*))
                 ((lambda (v)
                     (if v v
                         (cond [i2 e2 ...] ...))) i1)]
                [t (if i1
                       (begin e1 ...)
                       (cond [i2 e2 ...] ...))])))
```

```
(mkmac (begin0 e1 e2 ...)
       (with ([v (gensym '*)])
             (let ([v e1])
                  (begin e2 ... v))))
```

```
(mkmac (apply-if pred fn b ...)
       (with ([v (gensym '*)])
             (let ([v pred])
                  (if v
                      (fn v)
                      (begin b ...)))))
```

REFERENCES

[Friedman, *et al.* 1984] Daniel P. Friedman, Christopher T. Haynes, Eugene Kohlbecker, and Mitchell Wand. "Scheme 84 Reference Manual", version 0. Indiana University Computer Science Department Technical Report No. 153. Bloomington, February 1984.

[Pitman 1980] Kent M. Pitman. "Special Forms in Lisp", in the *Conference Record of the 1980 LISP Conference*, Stanford; pages 179–187.

[Steele and Sussman] Guy Lewis Steele, Jr. and Gerald Jay Sussman. "LAMBDA the Ultimate Imperative", Massachusetts Institute of Technology, A.I. Memo No. 353. Cambridge, March 1976.