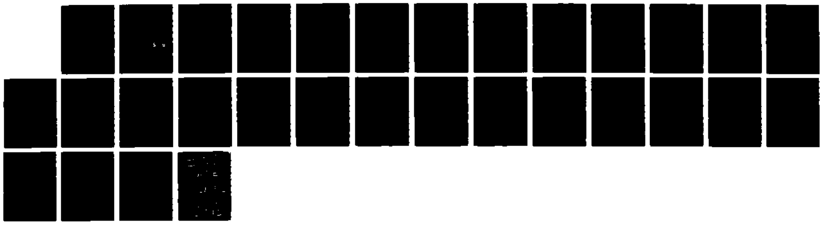
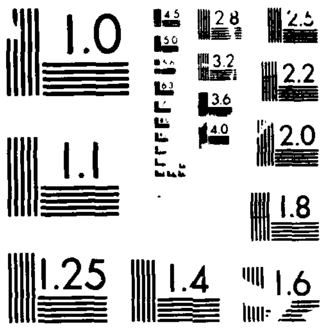
AD-A195 921 SYNTACTIC CLOSURES(U) MASSACHUSETTS INST OF TECH CAMBRIDGE ARTIFICIAL INTELLIGENCE LAB A BAWDEN ET AL. JUN 88 AI-M-1049 N00014-85-K-0124 1/1

UNCLASSIFIED F/G 5/2 NL





MICROCOPY RESOLUTION TEST CHART

DS 1963-A

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

DTIC FILE COPY

4

AD-A195 921

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER AI Memo 1049	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Syntactic Closures		5. TYPE OF REPORT & PERIOD COVERED memorandum
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Alan Bawden		8. CONTRACT OR GRANT NUMBER(s) N00014-85-K-0124 N00014-86-K-0180
9. PERFORMING ORGANIZATION NAME AND ADDRESS Artificial Inteligence Laboratory 545 Technology Square Cambridge, MA 02139		10. PROGRAM ELEMENT PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Advanced Research Projects Agency 1400 Wilson Blvd. Arlington, VA 22209		12. REPORT DATE June 1988
		13. NUMBER OF PAGES 27
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) Office of Naval Research Information Systems Arlington, VA 22217		15. SECURITY CLASS. (of this report)
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Distribution is unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

DTIC
ELECTE
JUL 2 7 1988
S D
D

18. SUPPLEMENTARY NOTES

None

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

LISP	extensible syntax
Scheme	referential transparency
macros	
lexical scoping	

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

In this paper ~~we~~ describe syntactic closures. Syntactic closures address the scoping problems that arise when writing macros. ~~We discuss~~ some issues raised by introducing syntactic closures into the macro expansion inferface~~, and we~~ ~~compare~~ syntactic closures with other approaches. Included is a complete implementation.

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

DTIC
COPY
INSPECTED
6

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

A.I. Memo 1049 June 1988

Accession For
NTIS CRA&I ✓
DTIC TAB
Unannounced
Justification
By
Distribution/
Availability Codes
Dist | Avail and/or Special
A-1

Syntactic Closures

Alan Bawden
Jonathan Rees

Abstract

In this paper we describe *syntactic closures*. Syntactic closures address the scoping problems that arise when writing macros. We discuss some issues raised by introducing syntactic closures into the macro expansion interface, and we compare syntactic closures with other approaches. Included is a complete implementation.

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the laboratory's artificial intelligence research is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contracts N00014-85-K-0124 and N00014-86-K-0180.

This is a revision of a paper that will appear in the *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming.*

©1988 Association for Computing Machinery.

1 The trouble with macros

Macros are an essential programming tool. Many programming languages support them, including virtually all dialects of Lisp. The use of macros can make programs easier to understand and maintain by allowing the programmer to extend the language with new constructs that suit his application.[3]

Macros are traditionally implemented using simple textual manipulations. Some examples of familiar macros in a hypothetical dialect of Scheme might be:

```
(define-macro (push obj-exp list-var)
  `(set! ,list-var (cons ,obj-exp ,list-var)))

(define-macro (or exp-1 exp-2)
  `(let ((temp ,exp-1))
     (if temp temp ,exp-2)))

(define-macro (catch body-exp)
  `(call-with-current-continuation
     (lambda (throw) ,body-exp)))
```

In each of these examples, the formal parameters to the macro are bound to source text, represented as S-expressions, and the body of the macro computes a replacement text, also represented as an S-expression.

This style of macro facility is simple and general, but it is prone to various kinds of scoping errors:

1. Variable references introduced by the macro can be inadvertently captured by lexical bindings in the client code. For example,

   ```
   (let ((cons 5))
     (push 'foo stack))
   ```

   would expand into

   ```
   (let ((cons 5))
     (set! stack (cons 'foo stack)))
   ```

   which is probably not what was intended. The client of push shouldn't have to know that push is implemented using cons.

2. A client's lexical reference can conflict with a binding introduced by the macro. For example,

```
(or (memq x y) temp)
```

would expand into

```
(let ((temp (memq x y)))
  (if temp temp temp))
```

which is even less likely to be what was intended. The client shouldn't have to be aware of this detail of `or`'s implementation.

3. Assuming that our hypothetical Scheme dialect also supports some kind of `with-macro` construct for defining local macros, syntactic keywords introduced by the macro can be inadvertently captured. For example,

   ```
   (with-macro (set! flag)
               `(set-flag! ',flag)
     (push 'foo stack))
   ```

   would expand into

   ```
   (with-macro (set! flag)
               `(set-flag! ',flag)
     (set! stack (cons 'foo stack))).
   ```

   This is analogous to the first case above. In the first case we were binding identifiers, while in this case we are binding syntactic keywords.

4. An example similar to case 2, but involving keywords, is possible, but never seems to happen in practice, so a good example is hard to provide. Our best attempt is as follows:

   ```
   (define-macro (contorted test x y)
     `(with-macro (use and/or)
                  `(,and/or ,',x ,',y)
        (if ,test
            (use and)
            (use or))))
   ```

   The client of `contorted` shouldn't have to be aware that the subexpressions named by `test`, `x`, and `y` will be within the scope of `contorted`'s auxiliary keyword `use`.

All of these problems are consequences of the fact that macros are oblivious to the lexical scoping of the program text that they are constructing. Any macro facility that proposes to address this shortcoming also has to take into account that sometimes the macro writer needs explicit control over scoping. For example when

```
(catch (+ 5 (throw 'x)))
```

expands into

```
(call-with-current-continuation
  (lambda (throw)
    (+ 5 (throw 'x))))
```

the client's use of `throw` *should* refer to the binding of `throw` introduced by the macro. This contrasts with case 2 above, where such capture was undesirable. There is an analogous case for binding of syntactic keywords.

In this paper we will present a solution to these scoping problems. We will also examine some of the related design issues, and discuss some remaining difficulties. In an appendix, we present a complete implementation of our solution.

2 Terminology

We need to precisely distinguish between various different kinds of names and environments. The following terminology will be used throughout this paper:

- A *name* is any token used to name something. Traditionally Lisp uses symbols, such as `quote` and `car`, for this purpose.
- A *keyword* (or a *syntactic keyword*) is a name used to introduce some special syntactic construct. `Lambda` and `set!` are familiar examples.
- An *identifier* is a name used to denote a variable. Familiar examples are `cdaadr` and `foo`.
- A *variable* is a particular binding of an identifier. For example, in

```
(lambda (x)
  (f x
     (lambda (x)
       (g x))))
```

there are two variables named by the identifier `x`.

- A *syntactic environment* maps identifiers to variables, and contains an interpretation for a number of syntactic keywords. Syntactic environments contain all of the contextual information necessary for interpreting the meaning of a particular expression.

- A *value environment* maps variables to their values (or more precisely, to locations that hold those values). Value environments contain the additional information necessary to execute an expression.

3 Our solution

In the same way that closures of `lambda`-expressions solve scoping problems at run time, we propose to introduce *syntactic closures* as a way to solve scoping problems at macro expansion time.

3.1 Syntactic closures

Like the closure returned by a `lambda`-expression, a syntactic closure consists of an environment of some kind, a list of names, and an expression. With both kinds of closures, all names occurring in the expression are taken relative to the environment, except those in the given list. The names in the list are to have their meanings determined later. In both cases a closure is a way of parameterizing an expression.

The difference is that the `lambda`-expression closure is invoked with positional arguments, while the syntactic closure is invoked in a "call-by-context" fashion. Call-by-context is natural in a situation where expressions are *constructed* out of other expressions; such context-dependence is the normal way expressions are combined.

Syntactic closures are created by the procedure `make-syntactic-closure`. It takes three arguments: a syntactic environment, a list of names, and an expression. It returns a closure of the expression in the environment, leaving the names free. Expressions are represented in the usual way, as

S-expressions. A syntactic closure can be thought of as a new kind of S-expression that does not have a printed representation. A syntactic closure can appear as a subexpression of another expression. (It can even appear as the left hand side of an assignment statement.)

Ordinarily, an expression inherits the meaning of the keywords and identifiers it contains from the context in which it appears, but a syntactic closure carries its own context with it. This enables tools that manipulate expressions to avoid identifier and keyword scoping problems while retaining the ability to construct expressions in the familiar way.

3.2 Writing expanders

The programmer defines macros by writing procedures called *expanders*. An expander is applied to a syntactic environment and an expression, and returns a syntactic closure. The expression is the piece of program text that is to be expanded, and the syntactic environment is derived from the context in which the expression occurred. The resulting syntactic closure is to be used in place of the original expression.

For example, here is the expander for the **push** macro:

```
(define (push-expander syntactic-env exp)
  (let ((obj-exp (make-syntactic-closure
                  syntactic-env '()
                  (cadr exp)))
        (list-var (make-syntactic-closure
                   syntactic-env '()
                   (caddr exp))))
    (make-syntactic-closure
      scheme-syntactic-environment '()
      `(set! ,list-var (cons ,obj-exp ,list-var)))))
```

In a production implementation it would be unnecessary to write this expander by hand. Since most expanders follow a common pattern, a convenient user interface can hide the calls to `make-syntactic-closure` except in cases where the programmer needs more precise control of syntactic environments. A front end can be designed that allows programmers to define the **push** macro in a more familiar and readable way. We are not advocating a user interface, but rather a set of tools suitable for constructing such user interfaces. (We will have more to say about the design of such user interfaces in the next section.)

The pattern followed by most expanders is as follows:

1. The subexpressions of the input expression are closed in the syntactic environment in which they occurred, that is, in the syntactic environment which was the argument to the expander. Note that these subexpressions might already be syntactic closures before they are passed to **`make-syntactic-closure`**. This isn't a problem, as any expression can be syntactically closed, even a syntactic closure that is already context insensitive.

2. An expression is created (typically using backquote) that is the expansion of the original expression. This expansion will include as subexpressions the syntactic closures created in step 1.

3. The expansion is then closed in a syntactic environment known to the expander. In the example this is the standard Scheme syntactic environment.

This avoids all capture problems by carefully closing each expression in the syntactic environment appropriate to the names it contains. Thus, the subexpressions of the input are closed in the environment of the input, and any new names introduced by the expander are resolved in an environment known to the expander.

In the `push` example, the programmer need not be concerned that the definition of the keyword **`set!`** might be locally redefined in the location where the `push`-expression occurred, because he uses a known syntactic environment to close the **`set!`** expression he constructs. He needn't worry about any local rebindings of `cons` either, because the mapping of the identifier named "`cons`" found in `scheme-syntactic-environment` will be the global variable named "`cons`", rather than any local variables that happen to have the same name.

To illustrate how another kind of capture is avoided, here is a definition of a simple version of `or`:

```
(define (or-expander syntactic-env exp)
  (let ((exp-1 (make-syntactic-closure syntactic-env '()
                 (cadr exp)))
        (exp-2 (make-syntactic-closure syntactic-env '()
                 (caddr exp))))
    (make-syntactic-closure
      scheme-syntactic-environment '()
      `((lambda (temp)
          (if temp temp ,exp-2))
        ,exp-1))))
```

As before, the programmer doesn't need to worry about local redefinitions of the keywords `lambda` and `if`, but notice that he doesn't have to worry that his use of a variable named "`temp`" will accidentally capture any variables of the same name in the second operand. This is because the second operand is closed in the syntactic environment that was current where the `or`-expression occurred, and thus any identifiers it may have contained named "`temp`" have already been resolved to the correct variable named "`temp`".

The second argument to `make-syntactic-closure` is used in those situations where the programmer wants a capture to occur. It is a list of names which are to be left syntactically free in the resulting expression. To illustrate, here is an expander for `catch`:

```
(define (catch-expander syntactic-env exp)
  (let ((body-exp (make-syntactic-closure
                    syntactic-env '(throw)
                    (cadr exp))))
    (make-syntactic-closure
      scheme-syntactic-environment '()
      `(call-with-current-continuation
         (lambda (throw) ,body-exp)))))
```

Here the expression in the body of a `catch` is closed using the syntactic environment current where the `catch`-expression occurred, with the name "`throw`" excepted. Thus the meaning of all the names in the expression will be correctly determined, with the name "`throw`" left free to be captured by the `lambda`-expression in which it is embedded.

4 Pragmatics

The process of macro expansion is overseen by a preprocessor whose nature is not specified here: it could be a simple rewrite phase that precedes (or is interleaved with) interpretation, or it could be integral to the front end of a compiler. The preprocessor starts with an input expression—perhaps read from a file or terminal—and some known syntactic environment. The environment gives meaning to top level identifiers (such as `car`) and keywords (such as `if`). Identifier bindings are added to the syntactic environment as the preprocessor descends through `lambda`-expressions. Keyword bindings are added by constructs like `with-macro`.

The traditional interface between expanders and the preprocessor is defined only in terms of program text. We have augmented this interface by introducing syntactic environments and closures. This raises a number of design questions that have to be answered somehow in any practical implementation. For example: Can programmers create new syntactic environments, or add new keywords to existing syntactic environments? And are there any operations on syntactic closures, such as extracting the original expression, or detecting that it represents a call to the `car` procedure?

The previous section describes the general low-level mechanism by which expanders communicate with the preprocessor. This section suggests a rudimentary framework that might help support any practical implementation of syntactic closures. We give an implementation of this framework in the appendix.

4.1 `Extend-syntactic-environment`

It is useful to have a procedure that extends a syntactic environment by associating an expander with a given keyword. For example

```
(extend-syntactic-environment
  scheme-syntactic-environment
  'push
  push-expander)
```

returns a new syntactic environment in which expressions of the form `(push ...)` are expanded by the expander `push-expander`. Any other expression is interpreted according to `scheme-syntactic-environment`.

4.2 Advertised syntactic environments

We have already seen several examples in which a known syntactic environment was needed. In particular, when an expander introduces a name into an expansion, it needs to be certain that the name has the intended meaning. For example, the expander for **push** requires that **cons** and **set!** have the meanings documented in the Scheme manual. The expander for a **without-interrupts** macro might wish to employ names that are defined in an internal system syntactic environment.

Both Scheme and Common Lisp draw a distinction between primitive keywords such as `lambda` and **quote** and derived keywords such as and and **case** that can be expressed in terms of the primitive ones. Thus the `scheme-syntactic-environment` itself might be constructed by adding macro bindings to a **core-syntactic-environment** whose only keyword bindings are those for the primitive keywords.

4.3 Macrologies

The **scheme-syntactic-environment** is a function of the **core-syntactic-environment**. This function is itself a useful abstraction that can be made available as a procedure. We call such functions from syntactic environments to syntactic environments *macrologies*. A macrology is an abstraction that captures the process of defining one language in terms of another.

Given any syntactic environment that assigns meanings to the primitive keywords, the `scheme-macrology` assigns meanings to all of the derived keywords. A programmer who wanted to experiment with an alternate definition of `if` could write

```
(define new-if-syntactic-environment
  (scheme-macrology
    (extend-syntactic-environment
      core-syntactic-environment
      'if
      new-if-expander)))
```

to obtain a syntactic environment in which the derived keywords and, **or**, and **cond** were all defined in terms of the new version of the primitive keyword **if**.

It is common to design a facility that introduces a collection of related syntactic extensions. Such a facility can be conveniently implemented as a macrology. For example, a stack-macrology might be written that extends a given syntactic environment by adding push and pop macros.

4.4 Locally defined macros

Designing a user interface for a macro facility that uses syntactic closures and environments raises a number of questions. While we don't intend to advocate any particular front end, these issues must be addressed by any such interface.

If keywords are subject to the same scoping rules as identifiers, it is natural to have a construct for introducing a local macro definition. For example, the programmer might write

```
(with-macro (push frob stack)
            `(set! ,stack (cons ,frob ,stack))
  ...)
```

to introduce a local push macro. For convenience, the variables frob and stack will be bound to syntactic closures of the argument expressions. The syntactic environment in which the arguments are closed will be the one that was in force where the push-expression occurred. (This is not a very general interface, but it will serve to illustrate the environment issues.)

After the replacement expression has been computed, it should be closed in some relevant syntactic environment. We could decree that some known syntactic environment, such as the scheme-syntactic-environment, is always used. Thus in the example above, the standard definitions of set! and cons would be obtained. But a convincing case can be made that

```
(let ((adjoin cons))
  (with-macro (push frob stack)
              `(set! ,stack (adjoin ,frob ,stack))
    (let ((adjoin +))
      (push (adjoin n m) sum-stack))))
```

should behave the same as

```
(set! sum-stack (cons (+ n m) sum-stack)).
```

That is, the replacement should be closed in the syntactic environment that was in force where the with-macro-expression occurred. The definition of with-macro given in the appendix works this way.

Alternatively, the replacement could be closed in the syntactic environment that is in effect *inside* the body of the `with-macro` expression. This would permit the definition of recursive macros that expand into expressions that employ the same macro again. An appropriate name for this variant of `with-macro` would be `with-macro-rec`, since it bears the same relation to `with-macro` as `letrec` does to `let`.

```
(with-macro-rec (or exp . other-exps)
                (if (null? other-exps)
                    exp
                    `(let ((temp ,exp))
                       (if temp
                           temp
                           (or ,@other-exps))))
  ...)
```

One often needs to introduce several macros simultaneously, especially if they are mutually recursive. `With-macro` and `with-macro-rec` would therefore be more useful if they followed the syntax of `let` and `letrec`.

The expression that computes the replacement raises a different environment issue: where to obtain the syntactic and value environments in which to evaluate it. Using the syntactic environment from where the `with-macro` occurred will not work, because that environment maps identifiers to variables that will not exist until run time. The implementation in the appendix simply uses fixed syntactic and value environments. Thus, the bodies of macros are always written in standard Scheme, even if the program itself is written in a different language.

4.5 Pattern matching

`With-macro`'s limited pattern matching ability would also have to be remedied in any real implementation. As it stands, it assumes that the expression to be expanded always consists of the keyword followed by a sequence of subexpressions, and it assumes that it is correct to close the subexpressions in the environment in which the expression occurred.

A more general solution would have the following capabilities:

- Checking that the input expression is properly formed.
- Selecting between alternatives based on the form of the input expression.

- Destructuring the input expression, to arbitrary depth, and binding variables to its component parts.
- Declaring which components are expressions to be syntactically closed, and in each case, what names are to be left free.

At the same time, this flexibility must be provided in such a way that the most common cases are concise.

The usual approach is to design a pattern matching language. Our experience is that this is as difficult as most other kinds of language design. The pattern language may resemble Lisp, e.g. by employing constructors such as `cons` to indicate destructuring, but this can lead to confusion because not all pattern operations have obvious analogues in Lisp, and not all Lisp constructs make sense in patterns. Alternatively, the pattern language may resemble Lisp data structures, e.g. by using a pair to match a pair (as in [2] and [4]), but this leaves no room to express additional pattern operations without introducing special keywords. Such languages quickly become verbose and baroque.

We regard this area as suitable for future research and do not choose to address it at this time.

5 Previous work

Many users of conventional macro systems are sensitive to scoping problems. Several techniques to ameliorate these situations have been discovered and rediscovered over the years:

- One way to avoid capture problems (like `or`'s problem with `temp`) is to use names so obscure that the macro's client is unlikely to discover them accidentally.
- An improvement on the use of obscure names is to use a `gensym` utility that generates an unlimited supply of names that are guaranteed not to conflict with other names.
- Another way to come up with obscure names is to directly manipulate the mapping from character strings to names. Common Lisp's packages[4] can be used in this manner: when a macro is defined in one package, and a client of the macro resides in a different package, then a given character string in the macro definition is effectively a different name from the same string occurring in the client.

- In [5], Steele advocates the use of thunks to avoid capture problems. He would define or as follows:

```
(define-macro (or exp-1 exp-2)
  `(let ((temp ,exp-1)
         (thunk (lambda () ,exp-2)))
     (if temp temp (thunk))))
```

- Some Lisp dialects provide a mechanism that enables the macro writer to insert absolute references into the replacement expression. Instead of

```
`(cons ,exp-1 ,exp-2)
```

  the macro writer could write

```
`(,(make-absolute-reference-to
      'cons
      standard-scheme-environment)
   ,exp-1
   ,exp-2) .
```

Each of these solutions is incomplete. Clients may unwittingly stumble upon obscure names; packages are not integrated with lexical scoping; thunks can't deal with scoping of keywords; absolute references are clumsy and error prone.

Some Scheme dialects provide interfaces that are similar in spirit to syntactic closures and environments. Syntax tables in MIT Scheme and T contain bindings for keywords, but they do not contain anything corresponding to our identifier-to-variable mapping.

Syntactic environments also bear a strong resemblance to the expansion functions of Dybvig et al.[1] Their expanders follow the same general protocol for processing expressions, and can be used to solve some scoping problems, but they lack the generality provided by syntactic closures.

MIT Scheme also has parsed expressions (called *S-code*) that resemble our syntactic closures in that they do not contain free keyword references. Syntactic closures may additionally leave some keywords free to be determined later.

It has been suggested that if macros were always written in a restricted pattern language, then the implementor of the pattern language could solve scoping problems once and for all. While we believe that it is good to have a convenient notation for defining the most common kinds of macros, we believe that there are occasions in which nothing less than the full power of Lisp will suffice.

"Hygienic macro expansion"[2] is the only other complete solution to the macro scoping problems of which we are aware. Hygienic expansion works by "painting" the entire input expression with some distinctive color before passing it to the expander. Then the returned replacement expression is examined to find those parts that originated from the input expression; these can be identified by their color. The names in the unpainted text are protected from capture by the painted text, and vice versa.

The painting is done without any understanding of the syntax of the input expression: paint is applied to expressions, quoted constants, `cond`-clauses, and the bound variable lists from `lambda`-expressions. This strikes us as being very undisciplined. We prefer a scheme that is everywhere sensitive to the underlying syntactic and semantic structure of the language.

In addition, it is difficult to comprehend how hygienic expansion operates and why it is correct. We feel that syntactic closures solve scoping problems in a natural, straightforward way.

6 Conclusions

The implementation given in the appendix was written for expository purposes. To gain practical experience with syntactic closures, we have also written a complete Scheme system in which macros (including most of Scheme's built-in special forms) are defined using the tools described here. The additional control provided by syntactic closures proved to be quite beneficial in practice. Syntactic closures allowed us to solve scoping problems that have plagued Lisp implementations for years.

The following areas deserve further exploration:

Some versions of Scheme allow one to write definitions (`define` forms) at the beginning of the body of various constructs as a more convenient way of writing a `letrec`. Definitions syntactically resemble expressions, which suggests that macros should be able to expand into them. It also suggests that a `define-macro` form should be permitted in the same contexts. Although these extensions are intuitively appealing, it is difficult to give them a precise meaning.

As noted above, expressive language constructs for defining macros remain to be designed, including possibly a perspicuous pattern matching language.

One might want to do things to syntactic closures other than just inserting them into expressions. For example, the Common Lisp `setf` macro[4] needs to examine an expression that accesses a value in order to determine how to transform it into a corresponding assignment expression. Other macros may need to do more sophisticated kinds of analysis.

We would also like to investigate the application of syntactic closures and environments to problems of programming in the large. When a system is composed of several modules, each consisting of a number of procedure, variable, and macro definitions, it becomes necessary to have a language for describing the interactions between the modules. We believe that syntactic environments must play an important role in any such language.

Syntactic closures are a powerful and convenient tool for solving macro scoping problems. As experienced macrologists, we have found them to be a pleasure to use. They enable one to write correct macros with ease and confidence.

Acknowledgments

David Moon and Kent Pitman made useful comments on drafts of this paper. We also wish to thank the participants at an informal workshop on macros, held at MIT in May 1987: Chris Hanson, Eugene Kohlbecker, Richard Mlynarik, Jim Philbin, Kent Pitman, Bill Rozas, Gerald Jay Sussman, and Mitch Wand.

References

[1] R. Kent Dybvig, Daniel P. Friedman, and Christopher T. Haynes. "Expansion-passing style: Beyond conventional macros." *1986 ACM Conference on Lisp and Functional Programming*, pp. 143–150.

[2] Eugene Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce Duba. "Hygienic macro expansion." *1986 ACM Conference on Lisp and Functional Programming*, pp. 151–159.

[3] Kent M. Pitman. "Special forms in Lisp." *Conference Record of the 1980 Lisp Conference*, pp. 179–187. Reprinted by ACM.

[4] Guy Lewis Steele, Jr. *Common Lisp: The Language.* Digital Press, Burlington MA, 1984.

[5] Guy Lewis Steele Jr. "Rabbit: a compiler for Scheme." MIT Artificial Intelligence Laboratory Technical Report 474, May 1978.

Appendix: an implementation

The implementation consists of two parts: a compiler, and the definition of `scheme-macrology`. The compiler is extremely simple; it generates an expression written in a subset of Scheme. However, the reader should not be misled by this into thinking that we are proposing to standardize a macro expansion procedure; in a practical system the compiler might well generate PDP-10 instructions.

The compiler consists of the implementations for syntactic environments, syntactic closures, and the core syntactic environment.

The implementation of the Scheme macrology is self-contained; it does not depend on any details of the compiler. It illustrates how many standard macros can be written using syntactic closures and includes an implementation of a `with-macro` keyword.

The user visible entry points are:

```
extend-syntactic-environment
make-syntactic-closure
core-syntactic-environment
scheme-macrology
scheme-syntactic-environment
```

all of which are described above.

```
; Utilities:

(define unique-symbol-counter 0)

(define (make-unique-symbol symbol)
  (set! unique-symbol-counter (+ 1 unique-symbol-counter))
  (string->symbol
    (string-append (symbol->string symbol)
                   "@"
                   (number->string unique-symbol-counter
                                   '(heur)))))
```

```
; Simple little run-time system:

; Object code is represented as ordinary Scheme expressions,
; except that combinations are introduced by a CALL
; "keyword".  This makes the run-time system extremely simple.

(define (execute code)
  (eval code user-initial-environment))

(define (call proc . args)
  (apply proc args))



; The compiler:

; A syntactic environment is implemented as a procedure that
; is applied to a second syntactic environment and an
; expression.  It is expected to return the compiled form of
; the expression.  The second syntactic environment is used
; to compile the subexpressions of the expression.

(define (compile syntactic-env exp)
  (syntactic-env syntactic-env exp))

(define (compile-list syntactic-env exps)
  (map (lambda (exp)
         (syntactic-env syntactic-env exp))
       exps))

; Syntactic environments:

(define (extend-syntactic-environment
             outer-syntactic-env keyword expander)
  (lambda (syntactic-env exp)
    (if (and (pair? exp)
             (eq? (car exp) keyword))
        (compile null-syntactic-environment
                 (expander syntactic-env exp))
        (outer-syntactic-env syntactic-env exp))))
```

```
; ADD-IDENTIFIER-LIST is used internally by LAMBDA to
; introduce new identifiers into the syntactic environment.
(define (add-identifier-list syntactic-env identifiers)
  (if (null? identifiers)
      syntactic-env
      (add-identifier (add-identifier-list syntactic-env
                                           (cdr identifiers))
                      (car identifiers))))

(define (add-identifier outer-syntactic-env identifier)
  (let ((variable (make-unique-symbol identifier)))
    (lambda (syntactic-env exp)
      (if (eq? exp identifier)
          variable
          (outer-syntactic-env syntactic-env exp)))))

; FILTER-SYNTACTIC-ENV creates a new syntactic environment in
; which a given list of names take their meaning from one
; syntactic environment, while all other names take their
; meaning from another.
(define (filter-syntactic-env
           names names-syntactic-env else-syntactic-env)
  (lambda (syntactic-env exp)
    ((if (memq (if (pair? exp) (car exp) exp) names)
         names-syntactic-env
         else-syntactic-env)
     syntactic-env
     exp)))
```

```
; The null syntactic environment is used to ensure that the
; expressions returned by expanders are syntactic thunks
; (i.e. have no free names).
(define (null-syntactic-environment syntactic-env exp)
  (if (syntactic-closure? exp)
      (compile-syntactic-closure syntactic-env exp)
      (error "Unclosed expression: ~S" exp)))

; The core syntactic environment is actually a part of the
; compiler, since it determines how code is to be generated
; for the primitive constructs.

(define (core-syntactic-environment syntactic-env exp)
  ((cond ((syntactic-closure? exp) compile-syntactic-closure)
         ((symbol? exp) compile-free-variable)
         ((not (pair? exp)) compile-constant)
         (else (case (car exp)
                 ((quote) compile-constant)
                 ((if begin set!) compile-simple)
                 ((lambda) compile-lambda)
                 (else compile-combination))))
   syntactic-env
   exp))

(define (compile-constant syntactic-env exp)
  exp)

(define (compile-free-variable syntactic-env exp)
  exp)

(define (compile-combination syntactic-env exp)
  `(call ,@(compile-list syntactic-env exp)))

(define (compile-simple syntactic-env exp)
  `(,(car exp) ,@(compile-list syntactic-env (cdr exp))))

(define (compile-lambda syntactic-env exp)
  (let ((syntactic-env (add-identifier-list syntactic-env
                                            (cadr exp))))
    `(lambda ,(compile-list syntactic-env (cadr exp))
       ,@(compile-list syntactic-env (cddr exp)))))
```

```
; Syntactic closures:

; A syntactic closure is implemented as a procedure that is
; marked so that it can be recognized when it is found in an
; expression.  The procedure is applied to the syntactic
; environment in which the closure's free names will be
; resolved.  The procedure returns the compiled form of the
; expression.

(define (make-syntactic-closure syntactic-env free-names exp)
  (vector 'syntactic-closure
          (lambda (free-names-syntactic-env)
            (compile (filter-syntactic-env
                       free-names
                       free-names-syntactic-env
                       syntactic-env)
                     exp))))

(define (make-syntactic-closure-list
           syntactic-env free-names exps)
  (map (lambda (exp)
         (make-syntactic-closure syntactic-env
                                 free-names
                                 exp))
       exps))

(define (syntactic-closure? x)
  (and (vector? x)
       (= 2 (vector-length x))
       (eq? 'syntactic-closure (vector-ref x 0))))

(define (compile-syntactic-closure
           syntactic-env syntactic-closure)
  ((vector-ref syntactic-closure 1) syntactic-env))

; Here ends the compiler.
; The Scheme Macrology:

; The scheme macrology assumes that it is applied to a
; syntactic environment in which the names LAMBDA, QUOTE, IF,
; BEGIN, SET!, MEMV, and MAKE-PROMISE are defined.

(define (scheme-macrology base-syntactic-env)

  (define (let-expander syntactic-env exp)
    (let ((identifiers (map car (cadr exp))))
      (make-syntactic-closure final-syntactic-env '()
        `((lambda ,identifiers
            ,@(make-syntactic-closure-list
                syntactic-env identifiers
                (cddr exp)))
          ,@(make-syntactic-closure-list
              syntactic-env '()
              (map cadr (cadr exp)))))))

  (define (delay-expander syntactic-env exp)
    (let ((delayed (make-syntactic-closure syntactic-env '()
                     (cadr exp))))
      (make-syntactic-closure final-syntactic-env '()
        `(make-promise (lambda () ,delayed)))))
```

```
(define (and-expander syntactic-env exp)
  (let ((operands (make-syntactic-closure-list
                    syntactic-env '()
                    (cdr exp))))
    (cond ((null? operands)
           (make-syntactic-closure final-syntactic-env '()
             '#t))
          ((null? (cdr operands)) (car operands))
          (else
           (make-syntactic-closure final-syntactic-env '()
             `(let ((temp ,(car operands)))
                (if temp
                    (and ,@(cdr operands))
                    temp)))))))

(define (or-expander syntactic-env exp)
  (let ((operands (make-syntactic-closure-list
                    syntactic-env '()
                    (cdr exp))))
    (cond ((null? operands)
           (make-syntactic-closure final-syntactic-env '()
             '#f))
          ((null? (cdr operands)) (car operands))
          (else
           (make-syntactic-closure final-syntactic-env '()
             `(let ((temp ,(car operands)))
                (if temp
                    temp
                    (or ,@(cdr operands)))))))))
```

```
(define (cond-expander syntactic-env exp)
  (make-syntactic-closure final-syntactic-env '()
    (process-cond-clauses syntactic-env (cdr exp))))

(define (process-cond-clauses
           syntactic-env clauses)
  (let ((body (make-syntactic-closure-list
                syntactic-env '()
                (cdar clauses))))
    (cond ((not (null? (cdr clauses)))
           (let ((test (make-syntactic-closure
                         syntactic-env '()
                         (caar clauses)))
                 (rest (process-cond-clauses
                         syntactic-env
                         (cdr clauses))))
             (if (null? body)
                 `(or ,test ,rest)
                 `(if ,test
                      (begin ,@body)
                      ,rest))))
          ((eq? (caar clauses) 'else) `(begin ,@body))
          (else
           (let ((test (make-syntactic-closure
                         syntactic-env '()
                         (caar clauses))))
             (if (null? body)
                 test
                 `(if ,test (begin ,@body))))))))
```

```
(define (case-expander syntactic-env exp)
  (make-syntactic-closure final-syntactic-env '()
    `(let ((temp ,(make-syntactic-closure syntactic-env '()
                                           (cadr exp))))
       ,(process-case-clauses syntactic-env (cddr exp)))))

(define (process-case-clauses syntactic-env clauses)
  (let ((data (caar clauses))
        (body (make-syntactic-closure-list
                syntactic-env '()
                (cdar clauses))))
    (cond ((not (null? (cdr clauses)))
           (let ((rest (process-case-clauses
                         syntactic-env
                         (cdr clauses))))
             `(if (memv temp ',data)
                  (begin ,@body)
                  ,rest)))
          ((eq? data 'else) `(begin ,@body))
          (else `(if (memv temp ',data)
                     (begin ,@body))))))
```

```
(define (with-macro-expander with-macro-syntactic-env exp)
  (let* ((keyword (caadr exp))
         (transformer (execute
                        (compile
                          scheme-syntactic-environment
                          `(lambda ,(cdadr exp)
                             ,(caddr exp)))))
         (expander (lambda (syntactic-env exp)
                     (make-syntactic-closure
                       with-macro-syntactic-env '()
                       (apply transformer
                              (make-syntactic-closure-list
                                syntactic-env '()
                                (cdr exp)))))))
    (make-syntactic-closure final-syntactic-env '()
      `(begin
         ,@(make-syntactic-closure-list
             (extend-syntactic-environment
               with-macro-syntactic-env
               keyword
               expander)
             '()
             (cdddr exp))))))
```

```
(define (with-macro-rec-expander
            with-macro-syntactic-env exp)
  (let* ((keyword (caadr exp))
         (transformer (execute
                        (compile
                          scheme-syntactic-environment
                          `(lambda ,(cdadr exp)
                             ,(caddr exp)))))
         (extended-syntactic-env #f)
         (expander (lambda (syntactic-env exp)
                     (make-syntactic-closure
                       extended-syntactic-env '()
                       (apply transformer
                              (make-syntactic-closure-list
                                syntactic-env '()
                                (cdr exp)))))))
    (set! extended-syntactic-env
          (extend-syntactic-environment
            with-macro-syntactic-env
            keyword
            expander))
    (make-syntactic-closure final-syntactic-env '()
      `(begin
         ,@(make-syntactic-closure-list
             extended-syntactic-env '()
             (cdddr exp))))))
  (define final-syntactic-env #f)

  ; A careful reading of the Scheme report reveals that you
  ; can't put this DO inside the previous DEFINE.
  (do ((syntactic-env base-syntactic-env
                      (extend-syntactic-environment
                        syntactic-env
                        (caar pairs)
                        (cadar pairs)))
       (pairs (list (list 'delay delay-expander)
                    (list 'or or-expander)
                    (list 'and and-expander)
                    (list 'let let-expander)
                    (list 'cond cond-expander)
                    (list 'case case-expander)
                    (list 'with-macro
                          with-macro-expander)
                    (list 'with-macro-rec
                          with-macro-rec-expander)
                    )
              (cdr pairs)))
      ((null? pairs)
       (set! final-syntactic-env syntactic-env)))

  final-syntactic-env

  ) ;end (define (scheme-macrology ...) ...)

(define scheme-syntactic-environment
  (scheme-macrology core-syntactic-environment))
```

END
DATE
FILMED
8-88
DTIC