JFP **16** (3): 253–267, 2006.
doi:10.1017/S0956796805005733 First published online 23 November 2005 Printed in the United Kingdom

EDUCATIONAL PEARL

Automata via macros

SHRIRAM KRISHNAMURTHI*
Brown University, Providence, RI, USA
(*e-mail:* sk@cs.brown.edu)

Abstract

Lisp programmers have long used macros to extend their language. Indeed, their success has inspired macro notations for a variety of other languages, such as C and Java. There is, however, a paucity of effective pedagogic examples of macro use. This paper presents a short, non-trivial example that implements a construct not already found in mainstream languages. Furthermore, it motivates the need for tail-calls, as opposed to mere tail-recursion, and illustrates how support for tail-call optimization is crucial to support a natural style of macro-based language extension.

1 Introduction

The idea of extending a language with domain-specific constructs has a long and distinguished history (Christensen & Shaw, 1969). More recently, the advent of lightweight program-generation techniques in mainstream languages, most notably the templates of C++, have made these ideas accessible to a large number of programmers. Generativity has even been used to build new forms of generic language constructs such as module systems, as witnessed by, for instance, mixin layers (Smaragdakis & Batory, 1998).

In the arena of reflective programming techniques, the Lisp community has long played a leading role in the form of *macros*. Indeed, the Scheme standard (Kelsey *et al.*, 1998) famously begins with the following design manifesto:

> Programming languages should be designed not by piling feature on top of feature, but by removing the weaknesses and restrictions that make additional features appear necessary.

Scheme augments a minimal set of features with a powerful macro system, which enable the creation of higher-level domain-specific and domain-independent language primitives.

How should programmers use macros? Over the years, a few common use-cases have become apparent:

* Partially supported by NSF grant ITR-0218973. An earlier version of this example appeared in the author's thesis (Krishnamurthi, 2001) and was presented at the *Lightweight Languages 1* conference.

- providing cosmetics
- introducing binding constructs
- implementing "control operators", i.e., ones that alter the order of evaluation
- defining data languages

The last of these is perhaps the most subtle. The sub-terms of a macro invocation can contain arbitrary phrases, some of which may not even be legitimate Scheme code. In such cases, the macro construct is essentially creating an abstraction over an implementation choice. For instance, suppose a particular datum can be represented either as a procedure or as a structure. Without macros, the same power can only be obtained by using **quote** followed by *eval*, which perforce defers the representation until evaluation. A macro gives the programmer a consistent, representation-independent means of describing the datum while still resolving the representation before compilation.

Unfortunately, there are relatively few accessible and yet instructive examples of the use of macros. Most examples tend to fall in one of two categories. Some examples are too simple and also too inward-looking: Presenting the definition of a construct like **let** in terms of immediate procedure application is unlikely to impress a programmer accustomed to languages where lexical scope introduction is already built-in. Other examples, such as the embedding of Prolog in Scheme (Haynes, 1987), are too complex to be understood readily (especially since they make heavy use of continuations). This creates a vacuum in the pedagogic space.

The problem of pedagogy is especially clear when trying to teach this material to students. When students are only familiar with traditional procedural programming, their response to being told that macros enable the programmer to create their own control construct is usually, "What other control constructs would I need?"

Furthermore, the approach to building a language by syntactic expansion into a small set of core features is more subtle than most novices realize. Its success in Scheme depends on a potent combination of several forces:

1. a set of very powerful core features;
2. very few restrictions on what can appear where (i.e., values in the language are truly *first-class*, which in turn means the expressions that generate them can appear nearly anywhere); and,
3. a relatively high degree of orthogonality between the core features.

The first means many macros can accomplish their tasks with relatively little effort, the second means the macros can be written in a fairly natural fashion, and the third means that macro writers don't need to worry about too many subtle interactions. A good pedagogic example would, therefore, also illustrate some of these design forces.

This paper presents such a pedagogic example of the use of macros, namely the definition of finite state automata. The example actually features all four of the uses of macros described above. In particular, it's easy to see that the automata are

control constructs, and yet they are not already built into traditional programming languages, thereby providing a constructive answer to the students' question.[1]

2 About the code in this paper

This paper cannot attempt to provide a tutorial on Scheme or its macro system. Please instead consult Dybvig's book (Dybvig, 1996) (available in full on-line; Chapter 8 explains macros in great detail). The key ideas in these systems are hygiene (Kohlbecker *et al.*, 1986; Kohlbecker, 1986), which prevents inadvertent identifier capture, and pattern matching (Kohlbecker & Wand, 1987; Kohlbecker, 1986), which greatly eases the construction and comprehension of macro definitions. Fortunately, both features are intuitive: pattern matching tends to make code look as it should, and hygiene ensures that such code also *works* as it should. Therefore, the reader should be able to comprehend the examples in this paper without understanding these features in detail.

In addition, all the code in this paper is executable, and I encourage readers to run these examples:

1. download and install DrScheme (Findler *et al.*, 2002) from `http://www.drscheme.org/`;
2. change the language level from the default, intended primarily for students, to one for professionals, which supports macros. In version 209, the Language menu's Choose Language ... entry lets you do this. Choose Pretty Big Scheme under the PLT tab.

3 Automata as macros

Suppose we want to define automata manually. Ideally, we should be able to specify the automata once and have different interpretations for the same specification; we also want the automata to be easy to write textually. In addition, we want the automata to execute fairly quickly, and to integrate well with the rest of the code.

Concretely, suppose we want to construct an automaton that recognizes the language $c(ad)^*r$, reminiscent of the Lisp identifier family *car*, *cdr*, *cadr*, *cddr*, *cddar* and so on. We might want to write this textually as

```
automaton init
  init : c -> more
  more : a -> more
         d -> more
         r -> end
  end  :
```

where the state named after the keyword `automaton` identifies the initial state.

[1] Harper's article on proof-directed debugging (Harper, 1999), for instance, uses automata for regular-expression matching as an example, but the automaton is hand-coded in ML, thereby obscuring some of its structure.

https://doi.org/10.1017/S0956796805005733 Published online by Cambridge University Press

How do we implement these automata as programs with dynamic behavior? *I request you, dear reader, to pause now and sketch the details of an implementation in your favorite language before proceeding further.*

One natural implementation of this language of automata is to create a vector or other random-access data structure to represent the states. Each state has an association indicating the actions—implemented as an association list, associative hash table, or other appropriate data structure. The association binds inputs to next states, which are references or indices into the data structure representing states. Given an actual input stream, a program would walk this structure based on the input. If the stream ends, it would accept the input; if no next state is found, it would reject the input; otherwise, it would proceed as per the contents of the data structure. (Of course, other implementations of acceptance and rejection are possible.)

A Scheme implementation of this program would look like this. First we represent the automaton as a data structure:

```
(define machine
  '((init (c more))
    (more (a more)
          (d more)
          (r end))
    (end)))
```

The following program is parameterized over machines and inputs (the primitive *assv* looks up a value in an association list):

```
;; run : automaton × symbol × list(symbol) → boolean
(define (run machine init-state stream)
  (define (walker state stream)
    (cond
     [(empty? stream) true]
     [else
      (let ([in (first stream)]
            [transitions (rest (assv state machine))])
        (let ([new-state (assv in transitions)])
          (if new-state
              (walker (first (rest new-state)) (rest stream))
              false)))]))
  (walker init-state stream))
```

Here are two instances of running this program:

```
> (run machine 'init '(c a d a d d r))
true
> (run machine 'init '(c a d a d d r r))
false
```

This is not the most efficient implementation we could construct in Scheme, but it is representative of the general idea.

https://doi.org/10.1017/S0956796805005733 Published online by Cambridge University Press

While this is a correct implementation of the semantics, it takes quite a lot of effort to get right. It's easy to make mistakes while querying the data structure, and we have to make several data structure decisions in the implementation. Can we do better?

To answer this question affirmatively, let's ignore the details of data structures and understand the *essence* of these implementations:

1. Per state, we need fast conditional dispatch to determine the next state.
2. Each state should be quickly accessible.
3. State transition should have low overhead.

Let's examine these criteria more closely to see whether we can recast them slightly:

fast conditional dispatch This could just be a conditional statement in a programming language. Compiler writers have developed numerous techniques for optimizing properly exposed conditionals (Bernstein, 1985).

rapid state access Pointers of any sort, including pointers to *functions*, would offer this.

quick state transition If only function calls were implemented as `gotos` ...

In other words, the `init` state could be represented by

```
(lambda (stream)
  (cond
    [(empty? stream) true]
    [else
     (case (first stream)
       [(c) (more (rest stream))]
       [else false])]))
```

That is, if the stream is empty, the procedure halts returning a true value; otherwise it dispatches on the first stream element. Note that the boxed expression is invoking the code corresponding to the `more` state. The code for the `more` state would thus be

```
(lambda (stream)
  (cond
    [(empty? stream) true]
    [else
     (case (first stream)
       [(a) (more (rest stream))]
       [(d) (more (rest stream))]
       [(r) (end (rest stream))]
       [else false])]))
```

https://doi.org/10.1017/S0956796805005733 Published online by Cambridge University Press

```
(define machine
  (letrec ([init
            (lambda (stream)
              (cond
                [(empty? stream) true]
                [else
                 (case (first stream)
                   [(c) (more (rest stream))]
                   [else false])]))]
           [more
            (lambda (stream)
              (cond
                [(empty? stream) true]
                [else
                 (case (first stream)
                   [(a) (more (rest stream))]
                   [(d) (more (rest stream))]
                   [(r) (end (rest stream))]
                   [else false])]))]
           [end
            (lambda (stream)
              (cond
                [(empty? stream) true]
                [else
                 (case (first stream)
                   [else false])]))])
    init))
```

Fig. 1. Implementation of an automaton.

Each boxed name is a reference to a state: there are two self-references and one to the code for the `end` state. Finally, the code for the `end` state fails to accept the input if there are any characters in it at all. While there are many ways of writing this, to remain consistent with the code for the other states we write it as

```
(lambda (stream)
  (cond
    [(empty? stream) true]
    [else
     (case (first stream)
       [else false])]))
```

Because there are no matching conditional clauses in the **case** statement this function always returns the value of the fall-through clause.

The full program is shown in Fig. 1. This entire definition corresponds to the machine; the definition of *machine* is bound to *init*, which is the function corresponding to the `init` state, so the resulting value needs only be applied to the

https://doi.org/10.1017/S0956796805005733 Published online by Cambridge University Press

input stream. For instance:

```
> (machine '(c a d a d d r))
true
> (machine '(c a d a d d r r))
false
```

What we have done is actually somewhat subtle. We can view the first implementation as an *interpreter* for the language of automata. This moniker is justified because that implementation has these properties:

1. Its output is an answer (whether or not the automaton recognizes the input), not another program.
2. It has to traverse the program's source as a data structure (in this case, the description of the automaton) repeatedly across inputs.
3. It consumes both the program and a specific input.

It is, in fact, a very classical interpreter. Modifying it to convert the automaton data structure into some intermediate representation would eliminate the overhead in the second clause, but not affect the other criteria.

In contrast, the second implementation given above is the *result of compilation*, i.e., it is what a compiler from the automaton language to Scheme might produce. Not only is the result a program, rather than an answer for a certain input, it also completes the process of transforming the original representation into one that does not need repeated processing.[2]

While this compiled representation certainly satisfies the automaton language's semantics, it leaves two major issues unresolved: efficiency and conciseness. The first owes to the overhead of the function applications. The second is evident because our description has become much longer; the interpreted solution required the user to provide only a concise description of the automaton, and reused a generic interpreter to manipulate that description. What is missing here is the actual compiler that can generate the compiled version, and that is what a macro represents.

3.1 Concision

First, let us slightly alter the form of the input. We assume that automata are written using the following syntax (presented informally):

```
(automaton init
  [init : (c → more)]
  [more : (a → more)
          (d → end)
          (r → end)]
  [end : ])
```

[2] This distinction is closely related to that between *deep* and *shallow* embeddings (Bowen & Gordon, 1995).

https://doi.org/10.1017/S0956796805005733 Published online by Cambridge University Press

```
(define-syntax automaton
  (syntax-rules (: →)        ;; match ':' and '→' literally, not as pattern variables
    [(_ init-state
       (state : (label → target) ···)
       ···)
     (letrec ([state
               (lambda (stream)
                 (cond
                   [(empty? stream) true]
                   [else
                    (case (first stream)
                      [(label) (target (rest stream))]
                      ···
                      [else false])]))]
              ···)
       init-state)]))
```

Fig. 2. A macro for executable automata.

The general transformation we want to implement is clear from the code in Fig. 1:

```
(state : (label → target) ···)   ⇒   (lambda (stream)
                                       (cond
                                         [(empty? stream) true]
                                         [else
                                          (case (first stream)
                                            [(label) (target (rest stream))]
                                            ···
                                            [else false])]))
```

Having handled individual rules, we must make the automaton macro wrap all these procedures into a collection of mutually-recursive procedures. The result is the macro shown in Fig. 2.

To use the automata that result from instances of this macro, we simply apply them to the input:

```
> (define m (automaton init
                       [init : (c → more)]
                       [more : (a → more)
                               (d → more)
                               (r → end)]
                       [end : ]))
> (m '(c a d a d d r))
true
> (m '(c a d a d d r r))
false
```

By defining this as a macro, we have made it possible to genuinely embed automata into Scheme programs. This is certainly true purely at a syntactic level—since the Scheme macro system respects the lexical structure of Scheme, it does not face problems that an external syntactic preprocessor might face. In

https://doi.org/10.1017/S0956796805005733 Published online by Cambridge University Press

addition, an automaton is just another applicable Scheme value. By virtue of being first-class, it becomes just another linguistic element in Scheme, and can participate in all sorts of programming patterns.

In other words, the macro system provides a convenient way of writing compilers from "Scheme+" to Scheme. More powerful Scheme macro systems (Dybvig *et al.*, 1993; Flatt, 2002; Krishnamurthi *et al.*, 1999) allow the programmer to embed languages that are truly different from Scheme, not merely extensions of it, into Scheme. A useful slogan[3] for Scheme's macro system is that it's a *lightweight compiler API*.

3.2 *Efficiency*

The remaining complaint against this implementation is that the cost of a function call adds so much overhead to the implementation that it negates any benefits the **automaton** macro might conceivably manifest. In fact, that's not what happens here at all, and this section examines why not.

Tony Hoare once famously said, "Pointers are like jumps" (Hoare, 1974). What we are seeking here is the reverse of this phenomenon: what is the `goto`-like construct that corresponds to a dereference in a data structure? The answer was given by Guy Steele (Steele, 1977): the *tail call*.

Informally, an invocation of function g within function f is *in tail position relative to* f if f returns whatever value g returns without operating on that value further. Steele's insight was that tail calls do not need to consume stack space. The stack already contains a return address where f should send its result. When f calls g in tail position, the run-time system can replace f's local variables with those for g but leave alone the return address, since the one at the top of the stack is the very one where g's result will go; since f performs no computation on g's return, its stack frame is no longer necessary.[4] In other words, the call just replaces the values of local variables, and a `goto` takes the place of the push-and-pop of a call and the subsequent return. Scheme implementations are *required* to implement tail-calls in a `goto`-like fashion (Kelsey *et al.*, 1998).

Armed with this insight, we can now reexamine the code. Studying the output of compilation, or the macro, we notice that the conditional dispatcher invokes the function corresponding to the next state on the rest of the stream—but does not touch the return value. This is no accident: the macro has been carefully written to only make tail calls.[5]

In other words, the state transition is hardly more complicated than finding the next state (which is statically determinate, since the compiler knows the location of all the local functions) and executing the code that resides there. The code generated from this Scheme source therefore has the features we discussed at the

[3] Due to Matthew Flatt, and quite possibly others too.

[4] A formal definition of tail-calling space behavior is outside the scope of this paper; we refer the reader to the exposition in Clinger's paper (Clinger, 1998), and to Appel's related notion of space safety (Appel, 1991).

[5] Even if the code did need to perform some operation with the result, it is often easy in practice to convert the calls to tail-calls using accumulators, as Abelson and Sussman (Abelson & Sussman, 1985) discuss. In general the conversion to continuation-passing style (Fischer, 1972) converts all calls to tail calls.

beginning of section 3: potentially random access for the procedures, references for state transformation, and some appropriately efficient implementation of the conditional.

The moral of this story is that we get the same representation we would have had to carefully craft by hand virtually for free from the compiler. In other words, *languages represent the ultimate form of reuse*, because we get to reuse everything from the mathematical (semantics) to the practical (libraries), as well as decades of research and toil in compiler construction (Hudak, 1998).

4 Fixing a flaw

The cautious reader must have noticed that the automata defined above are incorrect: they don't quite accept the advertised language. Consider the following interaction:

```
> (m '(c a d a))
true
```

That is, the macro defines automata that accept every substring of the desired language. We can trace the error to the overly generous termination condition, which accepts every terminating string. Indeed, the flaw is built into our definition of automata, because they contain no explicit specification of acceptance.

One fix is to extend the automata to explicitly signal acceptance. We can designate a state as accepting by using the keyword **accept**:

```
(define m2
  (automaton init
    [init : (c → more)]
    [more : (a → more)
            (d → more)
            (r → end)]
    [end : accept]))
```

We must now modify the macro definition accordingly.

Notice that the change affects only the grammar of state transitions, not the outer automaton specification. It is therefore best to first factor the macro into two components, as shown in Fig. 3. This simply explicates the informal factoring we presented in Sect. 3.1.

This refactoring helps isolate the change. Specifically, we must add the keyword and a corresponding rule to the **process-state** macro, as well as modify the code for the existing rule. Fig. 4 shows these changes. The first rule corresponds to the addition, while the boxed code in the second reflects the change. Testing this new version of the macro yields

```
> (m2 '(c a d r))
true
> (m2 '(c a d a))
false
> (m2 '(c a d a r))
true
> (m2 '(c a d a r r))
false
```

https://doi.org/10.1017/S0956796805005733 Published online by Cambridge University Press

```
(define-syntax process-state
  (syntax-rules (→)
    [(_ (label → target) ···)
     (lambda (stream)
       (cond
         [(empty? stream) true]
         [else
          (case (first stream)
            [(label) (target (rest stream))]
            ···
            [else false])]))]))

(define-syntax automaton
  (syntax-rules (:)
    [(_ init-state
        (state : response ···)
        ···)
     (letrec ([state
               (process-state response ···)]
              ···)
       init-state)]))
```

Fig. 3. Refactored implementation.

```
(define-syntax process-state
  (syntax-rules (accept →)
    [(_ accept)
     (lambda (stream)
       (cond
         [(empty? stream) true]
         [else false]))]
    [(_ (label → target) ···)
     (lambda (stream)
       (cond
         [(empty? stream) false]
         [else
          (case (first stream)
            [(label) (target (rest stream))]
            ···
            [else false])]))]))
```

Fig. 4. Modified macro.

Finally, we might like to create a single macro that encapsulates the helper. We can do this by moving the helper inside the **automaton** macro using **let-syntax**. One peculiarity makes this slightly tricky: because the helper is part of the *output* of the initial expansion, the ellipses in its definition must be quoted. We adopt the

https://doi.org/10.1017/S0956796805005733 Published online by Cambridge University Press

```
(define-syntax automaton
  (syntax-rules (:)
    [(_ init-state
        (state : response ...)
        ...)
     (let-syntax
         ([process-state
           (syntax-rules (accept →)
             [(_ accept)
              (lambda (stream)
                (cond
                  [(empty? stream) true]
                  [else false]))]
             [(_ (label → target) (... ...))
              (lambda (stream)
                (cond
                  [(empty? stream) false]
                  [else
                   (case (first stream)
                     [(label) (target (rest stream))]
                     (... ...)
                     [else false])]))])])
       (letrec ([state
                 (process-state response ...)]
                ...)
         init-state))]))
```

Fig. 5. Final version of the macro.

convention, supported by various Scheme implementations, that $(\cdots \ \cdots)$ expands into $\cdots$ in the output of expansion. The final macro definition is shown in Fig. 5.

5 Tail calls versus tail recursion

This example should help demonstrate the often-confused difference between tail *calls* and tail *recursion*. Books such as the first edition of Abelson and Sussman (Abelson & Sussman, 1985) discuss tail recursion, which is a special case where a function makes tail calls to *itself*. They point out that, because implementations must optimize these calls, using recursion to encode a loop results in an implementation that is really no less efficient than using a looping construct. They use this to justify, in terms of efficiency, the use of recursion for looping.

These descriptions unfortunately tell only half the story. While their comments on using recursion for looping are true, they obscure the subtlety and importance of optimizing all tail *calls*, which permit a family of functions to invoke each other without experiencing penalty. This leaves programmers free to write readable programs without paying a performance penalty—a rare "sweet spot" in the readability-performance trade-off. Traditional languages that offer only looping

https://doi.org/10.1017/S0956796805005733 Published online by Cambridge University Press

constructs and no tail calls force programmers to artificially combine procedures, or pay in terms of performance.

The functions generated by the **automaton** macro are a good illustration of this. If the implementation did not perform tail-call optimization but the programmer needed that level of performance, the macro would be forced to somehow combine all the three functions into a single one that could then employ a looping construct. This leads to an unnatural mangling of code, making the macro much harder to develop and maintain.

As a parting puzzle for our readers from more traditional languages, we ask you to consider how you would write complex, mutually-recursive procedures with loops instead. A good illustrative example of such a problem is the filesystem exercise that constitutes section 16, "Development through Iterative Refinement", in the first edition of *How to Design Programs* (Felleisen *et al.*, 2001), available on-line at `http://www.htdp.org/`.

6 Perspective

In the introduction, we placed several demands on the automaton example. How did it fare?

cosmetics This should be obvious. The only improvement to this notation would be to display it graphically.

binding construct Automata bind state names to their corresponding behavior. This is different than binding identifiers to values, but it is binding nonetheless.

control operator An executable automaton is clearly a kind of control construct.

data language We have shown two different implementations of automata, one as a direct interpreter and another by compilation to mutually-recursive functions. The macro for automata enables a programmer to ignore this distinction, while still leaving both implementations a possibility.

The example has also shown how automata exploit the ability to write procedures at will, and depend on tail-calls to make this non-standard loop actually have the informal semantics that a programmer associates with a "looping" construct. In the process, it shows how features that would otherwise seem orthogonal, such as macros and tail-calls, are in fact intimately wedded together; in particular, the absence of the latter would greatly complicate use of the former. Supporting the primitives that macros need is analogous to offering garbage collection: it frees the generator programmer from a number of important but ultimately low-level concerns, leaving more cognitive capacity free to deal with the concerns of the domain.

Acknowledgements

For sharing their wisdom on macros, I thank Dan Friedman, Mayer Goldberg, John Lacey, Kent Dybvig, Bruce Duba, Matthias Felleisen, Matthew Flatt, and Oleg Kiselyov. Thanks to Matthias Felleisen for numerous editorial remarks, to Christopher Dutchyn for his very careful reading, to Greg Sullivan for prompting

me to speak at Lightweight Languages 1, to *Dr. Dobb's Journal* for publishing the resulting talk on the Web, and to the several people who commented on the material in that talk.

References

Abelson, H. and Sussman, G. J. (1985) *Structure and Interpretation of Computer Programs*. MIT Press.

Appel, A. (1991) *Compiling with Continuations*. Cambridge.

Bernstein, R. L. (1985) Producing good code for the case statement. *Software—Practice & Exper.* **15**(10), 1021–1024.

Bowen, J. P. and Gordon, M. J. C. (1995) A shallow embedding of Z in HOL. *Infor. & Softw. Tech.* **37**(5–6), 269–276.

Christensen, C. and Shaw, C. J. (eds). (1969) *Proceedings of the Extensible Languages Symposium*. ACM. (Appeared as *SIGPLAN Notices*, **4**(8), 1–62, August 1969.)

Clinger, W. D. (1998) Proper tail recursion and space efficiency. *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 174–185.

Dybvig, R. K. (1996) *The Scheme Programming Language*, 2nd ed. Prentice-Hall.

Dybvig, R. K., Hieb, R. and Bruggeman, C. (1993) Syntactic abstraction in Scheme. *Lisp & Symbolic Computation*, **5**(4), 295–326.

Felleisen, M., Findler, R. B., Flatt, M. and Krishnamurthi, S. (2001) *How to Design Programs*. MIT Press.

Findler, R. B., Clements, J., Flanagan, C., Flatt, M., Krishnamurthi, S., Steckler, P. and Felleisen, M. (2002) DrScheme: A programming environment for Scheme. *J. Funct. Program.* **12**(2), 159–182.

Fischer, M. J. (1972) Lambda calculus schemata. *ACM SIGPLAN Notices*, **7**(1), 104–109.

Flatt, M. (2002) Composable and compilable macros. *ACM SIGPLAN International Conference on Functional Programming*.

Harper, R. (1999) Proof-directed debugging. *J. Funct. Program.* **9**(4), 463–470.

Haynes, C. T. (1987) Logic continuations. *J. Logic Program.* **4**, 157–176.

Hoare, C. A. R. (1974) Hints on programming language design. In: Bunyan, C. (editor), *Computer Systems Reliability*, pp. 505–534. Pergamon Press.

Hudak, P. (1998) Modular domain specific languages and tools. *International Conference on Software Reuse*.

Kelsey, R., Clinger, W. and Rees, J. (1998) Revised^5 report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, **33**(9).

Kohlbecker, E. E. and Wand, M. (1987) Macros-by-example: Deriving syntactic transformations from their specifications. *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 77–84.

Kohlbecker, E. E., Friedman, D. P., Felleisen, M. and Duba, B. F. (1986) Hygienic macro expansion. *ACM Symposium on Lisp and Functional Programming*, pp. 151–161.

Kohlbecker, Jr., E. E. (1986) *Syntactic extensions in the programming language Lisp*. PhD thesis, Indiana University.

Krishnamurthi, S. (2001) *Linguistic reuse*. PhD thesis, Department of Computer Science, Rice University.

Krishnamurthi, S., Felleisen, M. and Duba, B. F. (1999) From macros to reusable generative programming. *International Symposium on Generative and Component-based*

Software Engineering: Springer Lecture Notes in Computer Science 1799, pp. 105–120. Springer-Verlag.

Smaragdakis, Y. and Batory, n. (1998) Implementing layered designs and mixin layers. *European Conference on Object-oriented Programming*, pp. 550–570.

Steele, Jr., G. L. (1977) Debunking the "expensive procedure call" myth, or procedure call implementations can be considered harmful, or Lambda, the ultimate GOTO. *ACM Conference Proceedings*, pp. 153–162.