Chapter 36

Macros and their Impact on Language Design

36.1 Language Design Philosophy

The *Revised*^{5} *Report on the Algorithmic Language Scheme* 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 language primitives. This approach can only work, however, with a carefully designed target language for expansion. Its success in Scheme depends on a potent combination of two forces:

- A set of very powerful core features.
- 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).

The first means many macros can accomplish their tasks with relatively little effort, and the second means the macros can be written in a fairly natural fashion.

This manner of structuring a language means that even simple programs may, unbenownst to the programmer, invoke macros, and tools for Scheme must be sensitive to this fact. For instance, DrScheme is designed to be friendly to beginners. Even simple beginner programs expand into rather complicated and relatively mangled code, many of whose constructs the beginner will not understand. Therefore, when reporting errors, DrScheme uses various techniques to make sure this complexity is hidden from the programmer.

Building a language through macros does more than just complicate error reporting. It also has significant impact on the forms of generated code that the target implementation must support. Programmers who build these implementations make certain assumptions about the kinds of programs they must handle well; these are invariably based on what "a normal human would write". Macros, however, breaks these unwritten rules. They produce unusual and unexpected code, resulting in correctness and, particularly, performance errors. Sometimes these are easy to fix; in many other cases they are not. We will study examples to illustrate instances where macros crucially depend on the target language's handling of certain key code patterns.

36.2 Example: Pattern Matching

We will now examine a rather unusual construct that a programmer would never write, and explain why an implementation should nevertheless search for instances of it and handle it efficiently. To set the stage, consider the Scheme construct **let**, which binds names to values in a local lexical context. Though this (or an equivalent way of introducing local scope) would be a language primitive in most languages, in Scheme this is expressible as a rather simple macro in terms of first-class functions. That is,

```
(let ((v e) ···) b)
```

can be implemented by expanding into[1]

```
((lambda (v ···) b) e ···)
```

where (**lambda** (*v* ···) *b*) introduces an (anonymous) procedure with argument list *v* ··· and body *b*, and the outer parentheses apply this procedure to the argument expressions *e* ···. The application binds the variables *v* ··· to the values of the expressions *e* ···, and in that extended environment evaluates the body *b*—exactly what we would intend as the semantics for **let**. For instance, the program

```
(let ([x 3]
      [y 2])
   (+ x y))
```

which evaluates to $2 + 3$, i.e., 5, is transformed into

```
((lambda (x y)
    (+ x y))
  3 2)
```

This macro is, in fact, quite easy to implement: thanks to hygiene and pattern matching, the implementer of **let** merely needs to write

```
(define-syntax let
  (syntax-rules ()
    [(let ([v e] ···) b)
     ((lambda (v ···) b) e ···)]))
```

The Scheme pre-processor finds all bodies of the form (**let** ···), matches them against the input pattern (here, (**let** ([*v e*] ···) *b*)), binds the pattern variables (*v*, *e* and *b*) to the corresponding sub-expressions, and replaces the body with the output pattern in which the pattern variables have been replaced by the sub-expressions bound to them.

[1]For simplicity, we assume the body has only one expression. In reality, Scheme permits multiple expressions in the body, which is useful in imperative programs.

There is, however, a significant performance difference between the two forms. A compiler can implement **let** by extending the current activation record with one more binding (for which space can be pre-allocated by the creator of the record). In contrast, the expanded code forces the compiler to both create a new closure and then apply it—both relatively more expensive operations.

Given this expense, you might think it silly for a Scheme system to implement the **let**-to-**lambda** macro: why take an efficient source-language instruction, whose intent is apparent, and make it less efficient behind the programmer's back? Yet at least one Scheme compiler (Chez Scheme) does precisely this. Furthermore, in the back end, it finds instances of ((**lambda** ···) ···) and effectively handles them as it would have **let**.

Why would a compiler behave so perversely? Surely no human would intentionally write ((**lambda** ···) ···), so how else could these arise? The operative phrase is, of course, "no human". Scheme programs are full of program-generating programs, and by treating this odd syntactic pattern as a primitive, *all* macros that resolve into it benefit from the compiler's optimizations.

Consider a simple symbol-based conditional matcher: the user writes a series of symbol and action pairs, such as

```
(switch [off 0]
        [on  1])
```

The matcher performs the symbol comparison and, when a symbol matches, executes the corresponding action (in this case, the actions are already numerical values). The entire (**switch** ···) expression becomes a function of one argument, which is the datum to compare. Thus, a full program might be

```
(define m
  (switch [off 0]
          [on  1]))
```

with the following interactions with the Scheme evaluator:

```
> (m 'off)
0
> (m 'on)
1
```

To implement **switch**, we need a macro rule when there are one or more cases:

```
(switch                     ⇒         (lambda (v)
  [sym0 act0]                           (if (symbol=? v (quote sym0))
  [pat-rest act-rest]                       act0
  ···)                                      ((switch
                                               [pat-rest act-rest]
                                               ···)
                                              v)))
```

This yields a function that consumes the actual value (*v*) to match against. The matcher compares *v* against the first symbol. If the comparison is successful, it invokes the first action. Otherwise it needs to invoke the

```
(define-syntax switch
  (syntax-rules ()
    [(switch) (lambda (v) false)]
    [(switch [sym0 act0]
             [pat-rest act-rest]
             ···)
     (lambda (v)
       (if (symbol=? v (quote sym0))
           act0
           ((switch
             [pat-rest act-rest]
             ···)
            v)))]))
```

Figure 36.1: Simple Pattern Matcher

pattern-matcher on the remaining clauses. Since a matcher is a function, invoking it is a matter of function application. So applying this function to *v* will continue the matching process.[2]

For completeness, we also need a rule when no patterns remain. For simplicity, we define our matcher to return **false**[3] (a better response might be to raise an exception):

```
(switch)    ⇒    (lambda (v) false)
```

Combining these two rules gives us the complete macro, shown in Figure 36.1.

Given this macro, the simple use of **switch** given above generates

```
(lambda (v0)
  (if (symbol=? v0 (quote off))
      0
      ((lambda (v1)
         (if (symbol=? v1 (quote on))
             1
             ((lambda (v2)
                false)
              v1)))
       v0)))
```

(I've used different names for each *v*, as the hygienic expander might, to make it easy to keep them all apart. Each *v* is introduced by another application of the **switch** macro.)

[2]The ··· denotes "zero or more", so the pattern of using one rule followed by a ··· is common in Scheme macros to capture the potential for an unlimited number of body expressions.

[3]In many Scheme systems, **true** and **false** are written as `#t` and `#f`, respectively.

While this expanded code is easy to generate, its performance is likely to be terrible: every time one clause fails to match, the matcher creates and applies another closure. As a result, *even if the programmer wrote a pattern matching sequence that contained no memory-allocating code, the code might yet allocate memory*! That would be most unwelcome behavior.

Fortunately, the compiler comes to the rescue. It immediately notices the ((**lambda** ···) ···) pattern and collapses these, producing effectively the code:

```
(lambda (v0)
  (if (symbol=? v0 (quote off))
     0
     (let ([v1 v0])
       (if (symbol=? v1 (quote on))
          1
          (let ([v2 v1])
            false)))))
```

In fact, since the compiler can now see that these **let**s are now redundant (all they do is rename a variable), it can remove them, resulting in this code:

```
(lambda (v0)
  (if (symbol=? v0 (quote off))
     0
     (if (symbol=? v0 (quote on))
        1
        false)))
```

This is pretty much exactly what you would have been tempted to write by hand. In fact, read it and it's obvious that it implements a simple conditional matcher over symbols. Furthermore, it has a very convenient interface: a matcher is a first-class function value suitable for application in several contexts, being passed to other procedures, etc. The macro produced this by recursively generating lots of functions, but a smart choice of compiler "primitive"—((**lambda** ···) ···), in this case—that was sensitive to the needs of macros reduced the result to taut code. Indeed, it now leaves the code in a state where the compiler can potentially apply further optimizations (e.g., for large numbers of comparisons, it can convert the cascade of comparisons into direct branches driven by hashing on the symbol being compared).

36.3 Example: Automata

Next, we examine another optimization that is crucial for capturing the intended behavior of many programs. As an example, 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 as easy as possible to write (here, we stick to textual notations). In addition, we want the automata to execute fairly quickly, and to integrate well with the rest of the code (so they can, for instance, be written in-line in programs).

Concretely, suppose we want to write a simple automaton that accepts only patterns of the form $(01)^*$. We might want to write this textually as

```
automaton see0
  see0 : 0 -> see1
  see1 : 1 -> see0
```

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

Consider a slightly more complex automaton, one that recognizes the Lisp identifier family *car*, *cdr*, *cadr*, *cddr*, *cddar* and so on. That is, it should recognize the language $c(ad)^*r$. Its automaton might look like

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

We leave defining a more formal semantics for the automaton language as an exercise for the reader.

It is easy to see that some representation of the textual description suffices for treating the automata statically. How do we implement them as programs with dynamic behavior? *We request you, dear reader, to pause now and sketch the details of an implementation before proceeding further.*

A natural implementation of this language 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.)

One 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:

```
(define (run machine init-state stream)
  (define (walker state stream)
    (or (empty? stream)      ;; if empty, return true, otherwise . . .
        (let ([transitions (cdr (assv state machine))]
              [in (first stream)])
          (let ([new-state (assv in transitions)])
                  (if new-state
                      (walker (cadr new-state) (rest stream))
                      false)))))
    (walker init-state stream))
```

Here are two instances of running this:

```
> (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.

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 (which we have done only poorly above). 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.

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

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

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

```
(lambda (stream)
  (or (empty? stream)
      (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 similarly be

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

Each underlined 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)
  (or (empty? stream)
      (case (first stream)      ;; no matching clauses, so always false
        [else false])))
```

The full program is shown in Figure 36.2. 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 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 second overhead in the second clause, but would still leave open 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.

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.

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

Figure 36.2: Alternate Implementation of an Automaton

36.3.1 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 → more)
          (r → end))
  (end  : ))
```

The general transformation we want to implement is quite clear from the result of compilation, above:

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

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

Figure 36.3: A Macro for Executable Automata

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 Figure 36.3. 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 truly embed automata into Scheme programs. This is 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 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 allow the programmer to embed languages that are truly different from Scheme, not merely extensions of it, into Scheme. A useful slogan (due to Matthew Flatt and quite possibly others) for Scheme's macro system is that it's a *lightweight compiler API*.

36.3.2 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"[4]. 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: the *tail call*.

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. Indeed, the code generated from this Scheme source looks an awful lot like the automaton representation we discussed at the beginning of section 36.3: 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.

Tail Calls versus Tail Recursion

This example should help demonstrate the often-confused difference between tail *calls* and tail *recursion*. Many books 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 one another 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 constructs and no tail calls force programmers to artificially combine procedures, or pay via 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

[4]The context for the quote is pejorative: "Pointers are like jumps, leading wildly from one part of the data structure to another. Their introduction into high-level languages has been a step backwards from which we may never recover."

[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. In general, as we have seen, the conversion to continuation-passing style converts all calls to tail calls.

construct. This leads to an unnatural mangling of code, making the macro much harder to develop and maintain.

36.4 Other Uses

Scheme macros can do many more things. *datum→syntax-object* can be used to manufacture identifiers from syntax supplied to the macro. Macros can also define other macros! You might find such examples as you bgin to employ macros in your work. You might find Kent Dybvig's *The Scheme Programming Language* and Paul Graham's *On Lisp* useful books to understand these paradigms better.

36.5 Perspective

We have now seen several examples of Scheme's macro system at work. In the process, we have seen how features that would otherwise seem orthogonal, such as macros, first-class procedures and tail-calls, are in fact intimately wedded together; in particular, the absence of the latter two would greatly complicate use of the former. In this sense, the language's design represents a particularly subtle, maximal point in the design space of languages: removing any feature would greatly compromise what's left, while what is present is an especially good notation for describing algorithms.