Chapter 34

Macros as Compilers

For these notes, please use the PRETTY BIG language level.

34.1 Language Reuse

We have so far implemented languages as interpreters. In the real world, however, programming languages are defined not only by their implementation but also by their toolkit: think of the times you've disliked programming in a language because you didn't like the default editor or the debugger or the lack of a debugger or .... Therefore, when we set out to implement a fresh language implementation, we run the risk that we'll upset our users if we don't provide all the programming tools they're already accustomed to.

One way around this is to not create an entire implementation from scratch. Instead, we could just *compile* the new language into an existing language. If we do that, we can be fairly sure of reusing most of the tools built for the existing language. There is one problem, which is that feedback such as error messages may not make too much sense to the programmer (since she is expecting messages in terms of the constructs of the DSL, while the messages are in terms of the constructs of the target language). This is a real concern, but it is ameliorated some by the tools we will use.

Many languages provide a syntactic preprocessor that translates terms before handing them off to the evaluator. In languages like C and Scheme they're called *macros*, while in C++ they're called *templates*. We will now study the Scheme macro system in some depth. By default, the Scheme macro system permits programmers to add constructs to Scheme, thereby effectively providing a compiler from Scheme+ (the extended Scheme language) to Scheme itself.

34.1.1 Example: Measuring Time

Suppose we want to add a construct to the language that measures the time elapsed while evaluating an expression. That is, we want (**my-time** *e*) which returns the time it took (in milliseconds, say) to evaluate *e*. (The actual *time* command in Scheme also returns the value, but this version suffices for now. We'll use **my-time** for our attempts to avoid clashing with the vesion built-in.)

This is easy; here's the code:

(**define** (**my-time** *e*)

```
(let ([begin-time (current-milliseconds)])
  (begin
    e
    (- (current-milliseconds) begin-time))))
```

Let's test it:

```
> (my-time (+ 1 2))
0
```

Good; that's about what we'd expect. Even for slightly more computationally expensive expressions, we get

```
> (my-time (expt 2 1000))
0
```

Well, that's because DrScheme is really fast, see. How about:

```
> (my-time (expt 2 10000))
0
```

Hmm. Zero *milliseconds*? Maybe not. So let's try

```
> (my-time (expt 2 1000000))
0
```

This time DrScheme noticeably gives pause—we can tell from a wristwatch—so something is afoot.

The problem is that we defined **my-time** to be a procedure, and Scheme is an eager language. Therefore, the entire expression reduced to a value before the body of **my-time** began to evaluate. As a result, the difference in time was always going to be a constant. On different machines we might get different values, but the value isn't going to change, no matter what the expression!

How do we define **my-time**? There are three options.

First would be to introduce lazy evaluation into the language. This may be tempting, but it's going to make a mess overall, because it'd be impossible to determine when an expression is going to reduce, and an expression that has already been reduced to a value may need to have not been reduced later. This is not a viable solution.

The second is to make **my-time** take a thunk (recall: a procedure of no arguments). That is, we would have

```
(define (my-time e-thunk)
  (let ([begin-time (current-milliseconds)])
    (begin
      (e-thunk)
      (- (current-milliseconds) begin-time))))
```

so that

```
> (my-time (lambda () (expt 2 10000)))
0
> (my-time (lambda () (expt 2 100000)))
```

```
60
> (my-time (lambda () (expt 2 1000000)))
2023
```

This may be sufficient, but it's certainly not satisfactory: we've introduced an absolutely unnecessary syntactic pattern into the code for which we have no explanation other than that's just what the language demands. This is not an acceptable abstraction.

Finally, another is to accomplish the effect of textual substitution by using... textual substitution. In Scheme, we can instead write

```
(define-syntax my-time
  (syntax-rules ()
    [(time e)
     (let ([begin-time (current-milliseconds)])
       (begin
         e
         (- (current-milliseconds) begin-time)))]))
```

When we test this, we find

```
> (my-time (expt 2 1000))
0
```

Hmm! But ever hopeful:

```
> (my-time (expt 2 10000))
10
> (my-time (expt 2 100000))
70
> (my-time (expt 2 1000000))
2053
```

which is what we expect.

How does this version of **my-time** work? The Scheme *macro system* trawls the program source and gathers all the syntax definitions. It then substitutes all the uses of these syntax definitions with the bodies, where each syntax definition is defined by pattern-matching (we'll see several more examples). Only after finishing all the substitution does it hand the program to the Scheme evaluator, which therefore doesn't need to know anything about the syntax definitions. That is, given the above syntax definition and the program (**my-time** (*expt* 2 10000)), the program that the Scheme evaluator actually sees is

```
(let ([begin-time (current-milliseconds)])
  (begin
    (expt 2 10000)
    (- (current-milliseconds) begin-time)))
```

This is the right-hand-side of the first (and only) clause in the list of rules, except *e* has been substituted with the exponentiation expression. This is now an ordinary Scheme expression that the evaluator can reduce to a

value. Notice that the current time is now measured before and after the expression evaluates, thus ensuring that we do in fact clock its evaluation.[1]

34.1.2 Example: Local Definitions

We saw earlier this semester that

```
{with {var val} body}
```

could be rewritten as

```
{{fun {var} body} val}
```

by a preprocessor, so our core evaluator did not need to implement `with` directly. The same is true of the **let** construct in Scheme. Here's a simple macro for **let** (again, we'll use the **my-** convention to avoid any clashes):

```
(define-syntax my-let-1
  (syntax-rules ()
    [(let (var val) body)
     ((lambda (var) body) val)]))
```

Sure enough,

```
> (my-let-1 (x 3) (+ x 4))
7
> (my-let-1 (x 3) (my-let-1 (y 4) (+ x y)))
7
```

In full Scheme, however, the **let** construct is a bit more complex: it permits binding several identifiers at the same time (as we saw in a homework assignment regarding `with`). Therefore, the true translation should be regarded as something along these lines:

(let ([*var val*] ...) *body*) $\Longrightarrow$ ((**lambda** (*var* ...) *body*) *val* ...)

That is, we want each of the variables to remain in the same order, and likewise each of the value expressions—except we don't know how many we will encounter, so we use ... to indicate "zero or more".

How are we going to define this macro? In fact, it couldn't be easier. A researcher, Eugene Kohlbecker, observed that numerous extensions to Scheme had this same "zero or more" form, and noticed that people always wrote them informally using the stylized notation above. He therefore simply defined a macro system that processed that notation:

```
(define-syntax my-let
  (syntax-rules ()
    [(my-let ([var val] ...) body)
     ((lambda (var ...) body) val ...)]))
```

[1] Technically, this isn't exactly the expression that evaluates. We'll return to this in a bit.

Therefore (**my-let** ([*x* 3] [*y* 4]) (+ *x y*)) translates into ((**lambda** (*x y*) *body*) 3 4) which, sure enough, reduces to 7. Notice how the macro system is smart enough to treat the ([*var val*] ⋯) pattern as being the composite of the *var* ⋯ and *val* ⋯ patterns.[2] In particular, if no identifiers are bound, then this turns into an immediate application of a thunk to no arguments, which just evaluates the body.

34.1.3 Example: Nested Local Definitions

In a **let**, all the named expressions are bound in the same scope, which doesn't include any of the bound names. Sometimes, it's useful to bind names sequentially so later bindings can refer to earlier ones. Scheme provides the construct **let**∗ for this task:

```
(let* ([a 5]
       [b 12]
       [a^2 (* a a)]
       [b^2 (* b b)]
       [a^2+b^2 (+ a^2 b^2)])
  (sqrt a^2+b^2))
```

(Think of what this would evaluate to with **let** instead of **let**∗.)

We can implement **let**∗ very easily by unraveling it into a sequence of **let**s:

$$(\textbf{let}* ([var\ val] \cdots)\ body) \Longrightarrow (\textbf{let}\ ([var_0\ val_0])\ (\textbf{let}\ ([var_1\ val_1])\ \cdots\ (\textbf{let}\ ([var_n\ val_n])\ body)))$$

There is a stylized way of writing such macros in Scheme, which is to split them into two cases: when the sequence is empty and when the sequence has one or more elements. When there are no identifiers being bound, then **let**∗ does the same thing as **let** (which is to reduce to the expression itself):

(**let**∗ () *body*) ⟹ *body*

Since each ⋯ means "zero or more", we need to use a more refined pattern to indicate "one or more":

(**let**∗ ([*var0 val0*] [*var-rest val-rest*] ⋯) *body*)

The rewrite rule then becomes

$$(\textbf{let}* ([var0\ val0]\ [var\text{-}rest\ val\text{-}rest] \cdots)\ body) \Longrightarrow (\textbf{let}\ ([var0\ val0])\ (\textbf{let}* ([var\text{-}rest\ val\text{-}rest]\ \vdots)\ body))$$

That is, we apply the macro for **let**∗ recursively. Written in Scheme syntax, this is expressed as (notice the two cases):

[2] The use of brackets versus parentheses is purely stylistic.

```
(define-syntax my-let*
  (syntax-rules ()
    [(my-let* () body)
     body]
    [(my-let* ([var0 val0]
               [var-rest val-rest] ...)
       body)
     (let ([var0 val0])
       (my-let* ([var-rest val-rest] ...)
         body))]))
```

There is nothing in Scheme that prevents a runaway expansion. Therefore, it's possible to write a misbehaving macro that expands forever, so that evaluation never even begins. However, most macros follow the simple stylistic pattern above, which guarantees termination (the recursion is over the bound identifiers, and each time through, one more identifier-value pair is taken off).

34.1.4 Example: Simple Conditional

Let's say we want a simplified form of conditional that has only two branches and one conditional. This is effectively the same as **if**:

(**cond2** [*t e1*] [*else e2*]) $\Longrightarrow$ (**if** *t e1 e2*)

We might try the following macro:

```
(define-syntax cond2
  (syntax-rules ()
    [(cond2 (t e1) (else e2))
     (if t e1 e2)]))
```

This correctly evaluates expressions such as

```
(cond2 [(even? (current-seconds)) 'even]
       [else                      'odd])
```

Unfortunately, this also permits expressions such as

```
(cond2 [(even? (current-seconds)) 'even]
       [(odd? (current-seconds))  'odd])
```

This shouldn't be syntactically legal, because **cond2** permits only one conditional; in place of the second, we require programmers to write **else**. We can see that this second expression doesn't get evaluated at all by writing something atrocious:

```
(cond2 [false  'even]
       [(/ 1 0) 'odd])
```

which evaluates to 'odd.

What we want is for the **cond2** macro to simply reject any uses that don't have **else** in the second question position. This is where the mystical () after **syntax-rules** comes in: it lists the *keywords* in the macro. That is, we should instead define the macro as

```
(define-syntax cond2
  (syntax-rules (else)
    [(cond2 (t e1) (else e2))
     (if t e1 e2)]))
```

Then, we get the following interaction:

```
> (cond2 [false  'even]
         [(/ 1 0) 'odd])
cond2: bad syntax in: (cond2 (false (quote even)) ((/ 1 0) (quote odd)))}
```

Without the keyword designation, Scheme has no way of knowing that **else** has a special status; naturally, it makes no sense to build that knowledge into the macro system. Absent such knowledge, it simply treats *else* as a macro variable, and matches it against whatever term is in that position. When we put **else** in the keyword list, however, the expander no longer binds it but rather expects to find it in the right position—or else rejects the program.

34.1.5 Example: Disjunction

Let's consider one more example from Scheme lore. In Scheme, conditionals like **or** and **and** *short-circuit*: that is, when they reach a term whose value determines the result of the expression, they do not evaluate the subsequent terms. Let's try to implement **or**.

To begin with, let's define the two-arm version of **or**

```
(define (my-or2-fun e1 e2)
  (if e1
      e1
      e2))
```

Sure enough, a very simple example appears to work

```
> (my-or2-fun false true)
#t
```

but it fails on a more complex example:

```
> (let ([x 0])
    (my-or2-fun (zero? x)
                (zero? (/ 1 x))))
/: division by zero
```

whereas a short-circuiting evaluator would not have permitted the error to occur. The problem is, once again, Scheme's eager evaluation regime, which performs the divison before it ever gets to the body of *my-or2-fun*. In contrast, a macro does not have this problem:

```
(define-syntax my-or2
  (syntax-rules ()
    [(my-or2 e1 e2)
     (if e1 e1 e2)]))
```

which yields

```
> (my-or2 false true)
#t
> (let ([x 0])
    (my-or2 (zero? x)
            (zero? (/ 1 x))))
#t
```

In particular, the second expression translates into

```
(let ([x 0])
  (if (zero? x)
      (zero? x)
      (zero? (/ 1 x))))
```

(just replace *e1* and *e2* consistently).

As this expansion begins to demonstrate, however, this is an unsatisfying macro. We evaluate the first expression twice, which has the potential to be inefficient but also downright wrong. (Suppose the first expression were to output something; then we'd see the output twice. If the expression wrote a value into a database and returned a code, executing it a second time may produce a different result than the first time.) Therefore, we'd really like to hold on to the value of the first evaluation and return it directly if it's not false. That is, we want

```
(define-syntax my-or2
  (syntax-rules ()
    [(my-or2 e1 e2)
     (let ([result e1])
       (if result
           result
           e2))]))
```

This expands the second expression into

```
(let ([x 0])
  (let ([result (zero? x)])
    (if result
        result
        (zero? (/ 1 x)))))
```

Since Scheme is eager, the expression in the *e1* position gets evaluated only once. You should construct test cases that demonstrate this.

34.1.6 Example: For Loops

Many languages provide a looping construct for iterating through integers sequentially. Scheme doesn't for three reasons:

1. Because most such loops are anyway inappropriate: the indices only exist to traverse sequential data structures. Uses of *map* or *filter* over a list accomplish the same thing but at a higher level of abstraction.

2. Because recursion in the presence of tail calls has the same computational effect.

3. Because, if we really crave a more traditional syntax, we can define it using a macro!

We'll build up a loop macro in three stages.

Loops with Named Iteration Identifiers

Here's our first attempt at a **for** loop macro.[3] We've generously embellished it with keywords to increase readability:

```
(define-syntax for0
  (syntax-rules (from to in)
    [(for0 ⟨var⟩ from ⟨low⟩ to ⟨high⟩ in ⟨bodies⟩ ···)
     (local ([define loop (lambda (⟨var⟩)
                            (if (> ⟨var⟩ ⟨high⟩)
                                'done
                                (begin
                                  ⟨bodies⟩ ···
                                  (loop (+ ⟨var⟩ 1)))))])
       (loop ⟨low⟩))]))
```

This lets us write programs such as

```
(for0 x
      from 2
      to 5
      in (display x))
```

which prints 2, 3, 4 and 5. However, when we try this on a program like this

```
(for0 x
      from 2
      to (read)
      in (display x))
```

[3] We're using the convention of wrapping macro pattern-variables in ⟨···⟩ to emphasize their relationship to BNF.

we notice an unpleasant phenomenon: the program reads the upper-bound of the loop *every time through the loop*. To correct it, we should make sure it evaluates the upper-bound expression only once, which we can do with a small change to the macro:

```
(define-syntax for1
  (syntax-rules (from to in)
    [(for1 ⟨var⟩ from ⟨low⟩ to ⟨high⟩ in ⟨bodies⟩ ···)
     (local ([define high-value ⟨high⟩]
             [define loop (lambda (⟨var⟩)
                            (if (> ⟨var⟩ high-value)
                                'done
                                (begin
                                  ⟨bodies⟩ ···
                                  (loop (+ ⟨var⟩ 1)))))])
       (loop ⟨low⟩))]))
```

In general, we must be very careful with macros to ensure expressions are evaluated the right number of times. In this instance, ⟨*low*⟩ is going to be evaluated only once and ⟨*var*⟩ is only an identifier name, but we have to make sure ⟨*high*⟩ is evaluated only once.

In fact, however, this version is *also* buggy! If there is a (*read*) in the ⟨*low*⟩ position, that's going to get evaluated second instead of first, which is presumably not what we wanted (though notice that we didn't formally specify the behavior of **for**, either). So to get it right, we really need to evaluate ⟨*low*⟩ and bind its value to an identifier first.

In general, it's safer to bind all expression positions to names. Scheme's eager evaluation semantics ensures the expressions will only be evaluted once. We don't *always* want this, but we want it so often that we may as well do it by default. (The times we accidentally bind an expression too early—for instance, the conditional expression of a **while** loop—we will usually discover the problem pretty quickly by testing.) In addition we must be sure to do this binding in the right order, mirroring what the user expects (and what our documentation for the new language construct specifies). (Notice that the problematic expression in this example is (*read*), which has the side-effect of prompting the user. Of course, we may want to limit evaluation for efficiency reasons also.)

Loops with Implicit Iteration Identifiers

When we define a loop such as the one above, we often have no real use for the loop variable. It might be convenient to simply introduce an identifier, say **it**, that is automatically bound to the current value of the loop index. Thus, the first loop example above might instead be written as

```
(for2 from 2
      to 5
      in (display it))
```

Here's a proposed macro that implements this construct:

```
(define-syntax for2
  (syntax-rules (from to in)
```

```
[(for2 from ⟨low⟩ to ⟨high⟩ in ⟨bodies⟩ ...)
 (local ([define high-value ⟨high⟩]
         [define loop (lambda (it)
                        (if (> it high-value)
                            'done
                            (begin
                              ⟨bodies⟩ ...
                              (loop (+ it 1)))))])
   (loop ⟨low⟩))])
```

Notice that in place of ⟨*var*⟩, we are now using *it*. When we run this in DrScheme, we get:

```
> (for2 from 2 to 5 in (display it))
reference to undefined identifier: it
```

Oops! What happened here?

Actually, the macro system did exactly what it should. Remember hygiene? This was supposed to prevent *inadvertent capture* of identifiers across the macro definition/macro use boundary. It just so happens that in this case, we really do want **it** written in the macro *use* to be bound by **it** in the macro *definition*. Clearly, here's a good example of where we want to "break" hygiene, intentionally.

Unfortunately, the simple **syntax-rules** mechanism we've been using so far isn't quite up to this task; we must instead switch to a slightly more complex macro definition mechanism called *syntax-case*. For the most part, this looks an awful lot like **syntax-rules**, with a little more notation. For instance, we can define *for3* to be the same macro as **for1**, except written using the new macro definition mechanism instead:

```
(define-syntax (for3 x)
  (syntax-case x (from to in)
    [(for3 ⟨var⟩ from ⟨low⟩ to ⟨high⟩ in ⟨bodies⟩ ...)
     (syntax
       (local ([define high-value ⟨high⟩]
               [define loop (lambda (⟨var⟩)
                              (if (> ⟨var⟩ high-value)
                                  'done
                                  (begin
                                    ⟨bodies⟩ ...
                                    (loop (+ ⟨var⟩ 1)))))])
         (loop ⟨low⟩)))]))
```

To convert any **syntax-rules** macro definition into a corresponding one that uses *syntax-case*, we must make the three changes boxed above (adding a parameter to the macro name, providing the parameter as an explicit argument to *syntax-case*, and wrapping the entire output expression in (**syntax** ...).

We can similarly define *for4*:

```
(define-syntax (for4 x)
  (syntax-case x (from to in)
    [(for4 from ⟨low⟩ to ⟨high⟩ in ⟨bodies⟩ ...)
```

```
(syntax
 (local ([define high-value ⟨high⟩]
         [define loop (lambda (it)
                        (if (> it high-value)
                            'done
                            (begin
                              ⟨bodies⟩ ...
                              (loop (+ it 1)))))])
   (loop ⟨low⟩))))])
```

This does not solve the hygiene problem; it simply enables it by converting the macro to use *syntax-case*. The reason is that *syntax-case* provides additional capabilities. In particular, it provides a procedure called *datum→syntax-object*, which takes an arbitrary Scheme datum and a term in the macro body, and "paints" the datum with the same colors as those on the macro body. This has the effect of persuading the hygiene mechanism to treat the introduced term as if it were written by the programmer. As a result, it gets renamed consistently. Thus, we must write

```
(define-syntax (for4 x)
  (syntax-case x (from to in)
    [(for4 from ⟨low⟩ to ⟨high⟩ in ⟨bodies⟩ ...)
     (with-syntax ([it (datum→syntax-object (syntax for4) 'it)])
       (syntax
        (local ([define high-value ⟨high⟩]
                [define loop (lambda (it)
                               (if (> it high-value)
                                   'done
                                   (begin
                                     ⟨bodies⟩ ...
                                     (loop (+ it 1)))))])
          (loop ⟨low⟩)))))]))
```

The *with-syntax* construct introduces new pattern variables for use in the output. The first argument to *datum→syntax-object* identifies which expression the identifier the expander must pretend "introduced" the identifier. The second, in this case, is the symbol that will be painted appropriately. Therefore, the result of expansion on our running example will look something like

```
(local ([define high-value 5]
        [define loop (lambda (g1729)
                       (if (> g1729 high-value)
                           'done
                           (begin
                             (display g1729)
                             (loop (+ g1729 1)))))])
  (loop 2))
```

Notice how the uses of **it** are all renamed consistently. (In practice, other bound identifiers such as *high-value* and even *loop* will also acquire fresh names, but we don't show that here to keep the code more readable.) Indeed, this mechanism is sufficiently robust that it will even do the right thing with nested loops:

```
(for4 from 2 to 5 in
      (for4 from 1 to it in
            (display it))
      (newline))
```

generates

```
12
123
1234
12345
```

In the inner loop, notice that the **it** in the loop bound (**from** 1 **to it**) is the iteration index for the *outer* loop, while the **it** in (*display* **it**) is the index for the inner loop. The macro system associates each **it** appropriately because each use of **for4** gets a different coat of colors. Unfortunately, we have lost the ability to refer to the outer iteration in the inner loop.

Combining the Pieces

A better design for an iteration construct would be to combine these ways of specifying the iteration identifier (explicitly and implicitly). This is easy to do: we simply have two rules.[4] If an identifier is present, use it as before, otherwise bind **it** and recur in the macro.

```
(define-syntax (for5 x)
  (syntax-case x (from to in)
    [(for5 from ⟨low⟩ to ⟨high⟩ in ⟨bodies⟩ ···)
     (with-syntax ([it (datum→syntax-object (syntax for5) 'it)])
       (syntax
        (for5 it from ⟨low⟩ to ⟨high⟩ in ⟨bodies⟩ ···)))]
    [(for5 ⟨var⟩ from ⟨low⟩ to ⟨high⟩ in ⟨bodies⟩ ···)
     (syntax
      (local ([define high-value ⟨high⟩]
              [define loop (lambda (⟨var⟩)
                             (if (> ⟨var⟩ high-value)
                                 'done
                                 (begin
                                   ⟨bodies⟩ ···
                                   (loop (+ ⟨var⟩ 1)))))])
        (loop ⟨low⟩))))]))
```

[4] When defining such macros, be very sure to test carefully: if an earlier rule subsumes a later rule, the macro system will not complain, but the code will never get to a later rule! In this case we need not worry since the two rules have truly different structure.

This passes all the expected tests: both the following expressions print the numbers 2 through 5:

```
(for5 x from 2 to 5 in (display x))
(for5 from 2 to 5 in (display it))
```

while this

```
(for5 x from 2 to 5 in
      (for5 from 1 to x in
            (printf "[~a, ~a] " x it))
      (newline))
```

prints

```
[2, 1] [2, 2]
[3, 1] [3, 2] [3, 3]
[4, 1] [4, 2] [4, 3] [4, 4]
[5, 1] [5, 2] [5, 3] [5, 4] [5, 5]
```

There are still ways to many ways of improving this macro. First, we might want to make sure ⟨*var*⟩ is really a variable. We can use *identifier?* for this purpose. The *syntax-case* mechanism also permits *guards*, which are predicates that refine the patterns and don't allow a rule to fire unless the predicates are met. Finally, the following program does not work:

```
(for5 x from 2 to 5 in
      (for5 from 1 to it in
            (printf "[~a, ~a] " x it))
      (newline))
```

It reports that the boxed **it** is not bound (why?). Try to improve the macro to bind **it** in this case.

34.2 Hygiene

Now what if the use of **my-or2** really looked like this?

```
(let ([result true])
     (my-or2 false
             result))
```

which should evaluate to true. The expansion, however, is

```
(let ([result true])
  (let ([result false])
    (if result
        result
        result)))
```

which evaluates to false!

What happened here? When we look at just the input expression, we do not see only one binding of *result*. Reasoning locally to that expression, we assume that **my-or2** will evaluate the first expression and, finding it false, will evaluate the second; since this is *result*, which is bound to true, the overall response should also be true. Instead, however, the use of *result* within the macro definition interferes with *result* in the context of its use, resulting in the incorrect result.

The problem we see here should seem awfully familiar: this is exactly the same problem we saw under a different guise when trying to understand scope. Here, *result* in the second arm of the disjunction is bound in the **let** just outside the disjunction. In contrast, *result* inside the macro is bound inside the macro. We as programmers should not need to know about all the names used within macros—*just as we don't need to know the names of identifiers used within functions!* Therefore, macros should be forced to obey the scoping rules of the language.

Just to be sure, let's try this expression in our evaluator:

```
> (let ([result true])
    (my-or2 false result))
#t
```

We get true! This is because Scheme's macro system is *hygienic*. That is, it automatically renames identifiers to avoid accidental name clashes. The expression that actually evaluates is something like

```
(let ([result true])
  (let ([g1729 false])
    (if g1729
        g1729
        result)))
```

where *g1729* is a uniquely-generated identifier name. Notice that only the *result*s within the macro definition get renamed. In fact, because **let** is itself a macro, its identifiers also get renamed (as do those introduced by **lambda** and other binding forms), so the real program sent to the evaluator might well be

```
(let ([g4104 true])
  (let ([g1729 false])
    (if g1729
        g1729
        g4104)))
```

Many macro systems, such as that of C, are not hygienic. Programmers sometimes try to circumvent this by using hideous identifier names, such as `__macro_result_`. *This is not a solution!*

1. Not only is a it painful to have to program this way, small typos would greatly increase development time, and the macro would be much harder to decipher when a programmer tries to modify or correct it later.

2. This solution is only as good as the programmer's imagination; the problem still persists, lying in wait for just the right (wrong!) identifier to be bound in the context of use. Indeed, while a programmer may choose a sufficiently obscure name from the perspective of other programmers, not all source is

written by humans. A tool generating C code (such as a Scheme-to-C compiler) may happen to use exactly this naming convention.

3. This name is only obscure "upto one level". If the macro definition is recursive, then recursive instances of the macro may interfere with one another.

4. If you use this macro to debug the source that contains the macro (e.g., compiling the C compiler using itself), then your carefully-chosen "obscure" name is now *guaranteed* to clash!

In short, to return to a theme of this course: we should view these kinds of contortions by programmers as a symptom of a problem that must be addressed by better language design. Don't settle for mediocrity! In this case, hygiene is that solution.[5]

Notice, by the way, that we needed hygiene for the proper execution of our very first macro, because **my-time** introduced the identifier *begin-time*. At the time, we never even gave a thought to this identifier, which means in the absence of hygiene, we had a disaster waiting to happen. With hygiene, we can program using normal names (like *begin-time* and *result*) and not have to worry about the consequencs down the line, just as with static scope we can use reasonable names for local identifiers.

34.3 Comparison to Macros in C

Macro systems have a bad rap in the minds of many programmers. This is invariably because the only macro system they have been exposed to is that of C. C's macros are pretty awful, and indeed used to be worse: macros could contain *parts* of lexical tokens, and macro application would glue them together (e.g., the identifier *list-length* could be assembled by a macro that generated *lis*, another generating *t-le* and yet another generating *ngth*). C macros are not hygienic. Because C has no notion of local scope, C macros could not easily introduce local identifiers. Finally, C macros are defined by the C pre-processor (`cpp`), which operates on files a *line at a time*. Therefore, to apply a macro over a multi-line argument, a C programmer would have to use a \ at the end of each line to fool the pre-processor into concatenating the adjacent line with the present one. Failing to remember to use the line-continuation character could lead to interesting errors.

In contrast, Scheme's macros operate over parenthesized expressions instead of pestering programmers with lines. They respect lexical boundaries (to create a new identifier, you must do so explicitly—it cannot happen by accident). Scheme macros are hygienic. They have many more features that we haven't discussed here. In short, they correct just about every mistake that C's macro system made.

34.4 Abuses of Macros

When shouldn't a programmer use macros?

As you can see, macros provide a programmer-controlled form of *inlining*, that is, directly substituting the body of an abstraction in place of its use. Compilers often inline small procedures to avoid paying the cost of procedure invocation and return. This permits programmers to define abstractions for simple

[5]The algorithm, in effect, "paints" each expression on expansion, then consistently renames identifiers that have the same paints.

operations—such as finding the corresponding matrix element in the next row when the matrix is stored linearly, or performing some bit-twiddling—without worrying about the overhead of function invocation. Normally, inlining is done automatically by a compiler, after it has examined the size of the procedure body and determined whether or not it is cost-effective to inline.

Unfortunately, early compilers were not savvy enough to inline automatically. C programmers, looking to squeeze the last bit of performance out of their programs, therefore began to replace function definitions with macro definitions, thereby circumventing the compiler. Invariably, compilers got smarter, architectures changed, the cost of operations altered, and the hard-coded decisions of programmers came to be invalidated. Nowadays, we should *regard the use of macros to manually implement inlining as a programming error*. Unfortunately, many C programmers still think this is the primary use of macros (and in C, it's not useful for a whole lot else), thereby further despoiling their reputation. (The previous sentence is intentionally ambiguous.)

Another bad use of macros is to implement laziness. *Macros do not correspond to lazy evaluation.* Laziness is a property of when the implementation evaluates arguments to functions. Macros are not functions. For instance, in Scheme, we cannot pass a macro as an argument: try passing **or** as an argument to *map* and see what happens. Indeed, macro expansion (like type-checking) happens in a completely different phase than evaluation, while laziness is very much a part of evaluation. So please don't confuse the two.

34.5 Uses of Macros

When should a programmer use macros?

providing cosmetics Obviously, macros can be used to reduce the syntactic burden on programmers. These are perhaps the least interesting use; at least, a macro that does this should also fulfill one of the other uses.

introducing binding constructs Macros can be used to implement non-standard binding constructs. We have seen two examples, **let** and **let**$*$, above. If these were not already in the language, we could easily build them using macros. They would be impossible to define *as language contructs* in most other languages.

altering the order of evaluation Macros can be used to impose new orders-of-evaluation. For instance, we saw *time* suspend evaluation until the clock's value had been captured. The **or** construct introduced short-circuit evaluation. Often, programmers can obtain the same effect by thunking all the sub-expressions and thawing (the opposite of thunking) them in the desired order, but then the programmer would be forced to write numerous **lambda** () $\cdots$)'s—replacing one intrusive, manual pattern with another. (In particular, if a programmer fails to obey the pattern faithfully, the behavior may become quite difficult to predict.)

defining data languages Sometimes the sub-terms of a macro application may not be Scheme expressions at all. We have seen simple instances of this: for example, in (**my-let** ([**x** 3] [*y* 4]) (+ **x** *y*)), neither the parentheses wrapping the two bindings, nor those surrounding each name-value pair, signify applications. In general, the terms may have arbitrary structure, even including phrases that would be

meaningless in Scheme, such as (*my-macro* (**lambda** (**x**))) that would be syntactic errors otherwise. We can get some of the same benefit from using quotations, but those are run-time values, whereas here the macro can traverse the sub-terms and directly generate code.

In particular, suppose you wish to describe a datum without choosing whether it will be represented as a structure or as a procedure. In ordinary Scheme, you have to make this decision up front, because you cannot "introduce a **lambda**" after the fact. Designating the datum using a macro lets you hide this decision, deferring the actual representation to the macro.