CS1102: Introduction to Macros

Kathi Fisler and Charles Rich, WPI

September 22, 2016

1 The Need to Improve Our Slideshow Language

Last we looked at our slideshow package, we were left with an unsettled feeling that we hadn't really created a language. We created a collection of data definitions for programs and an interpreter to process those definitions, but the structures didn't give us something that looked like a conventional programming language. Recall that at the end of the last lecture I explained that we actually HAVE created a language (the hard part, that is), we just hadn't put the cleaner notation on top of it. Today, we want to see what we need to do to handle this final step.

As a starting point, let's recall what our current talk programs look like:

```
(define talk1
  (let ([intro-slide (lambda () (make-slide ...))]
        [arith-eg-slide
         (lambda ()
           (make-slide
            (next-example-title)
            (make-pointlist (list "(+ (* 2 3) 6)" "(+ 6 6)" "12") false)))]
        [func-eg-slide (lambda () (make-slide ...))]
        [summary-slide (lambda () (make-slide ...))])
    (make-talk
     (list (make-display intro-slide)
           (make-timecond 5
                          (list (make-display arith-eg-slide))
                          empty)
           (make-display func-eg-slide)
           (make-display summary-slide)))
```

If we wanted to clean up this syntax, what might we want to do?

1. We probably want to get rid of parts of the code that are too Scheme-specific (**lambda**, *make-* from *define-structs*, etc).
2. We want to get rid of details that the implementation needs, but that don't contribute information about the computation that the programmer wants to perform.

For example:

- The (**lambda** () ...) around each slide is annoying. The **lambda** is definitely Scheme-specific. Furthermore, someone who is writing talks shouldn't have to remember to wrap a **lambda** around every slide.
- All of the *list* commands are annoying. We have to write them even if we have only one item to put in the list. Also, these expose the language implementation to the programmer.
- The program really isn't self-contained. We use Scheme **let** to specify the slides, then use the slide names in the actual body of the talk. This isn't a big deal, but it is something we would ideally like to address.

- Having to write *true* and *false* in each pointlist isn't as annoying as the other issues, but it is error-prone and mildly irksome. If someone is writing a program in this language, they have to keep recalling whether *true* yields a numbered or a bulleted list.

In general, details that are extraneous to the program someone wants to write are problematic for two reasons:

- The programmer might forget to write them down, leading to program errors.
- The whole point of creating a new language is to give programmers constructs that make it easier to write certain kinds of programs. The more a language has extraneous details, the harder programs are to write, and the less useful the language becomes.

With this in mind, let's work on improving our programming language.

1.1 Fixing Pointlist Specifications

How might we augment the language to save programmers from remembering the correspondence between *true/false* and the numbering scheme? We can easily address this by adding some functions to our program that set the numbered? field for us:

```
;; pointlist-numbered : list[string] → pointlist
;; create numbered pointlist with given points
(define (pointlist-numbered points)
  (make-pointlist points true))

;; pointlist-bulleted : list[string] → pointlist
;; create non-numbered pointlist with given points
(define (pointlist-bulleted points)
  (make-pointlist points false))
```

Using these functions, we could rewrite our talk program as:

```
(define talk1
  (let ([intro-slide (lambda () (make-slide ...))]
        [arith-eg-slide
         (lambda ()
           (make-slide
            (next-example-title)
            (pointlist-bulleted (list "(+ (* 2 3) 6)" "(+ 6 6)" "12"))))]
        [func-eg-slide (lambda () (make-slide ...))]
        [summary-slide (lambda () (make-slide ...))])
    (make-talk
     (list (make-display intro-slide)
           (make-timecond 5
                          (list (make-display arith-eg-slide))
                          empty)
           (make-display func-eg-slide)
           (make-display summary-slide)))
```

This example demonstrates one easy way to make a program notation more readable: *introduce functions to supply data that the programmer might otherwise forget how to specify*.

An aside

If you wanted to get rid of the *list* argument to the pointlists, you could instead write these new helpers as functions taking an arbitrary number of arguments as follows:

```
;; pointlist-bulleted2 : list[string] → pointlist
;; create bulleted pointlist with given points
(define pointlist-bulleted2
  (lambda points
    (make-pointlist points false)))
```

we would call this as (*pointlist-bulleted2* "(+ (∗ 2 3) 6)" "(+ 6 6)" "12").

In general, Scheme turns (**define** (*foo x*) ...) into (**define** *foo* (**lambda** (*x*) ...)). This style creates an anonymous function (using **lambda**) then uses **define** to name that function. This version of pointlist-numbered goes one step farther and takes the parens off around the parameter specification. When you do this, Scheme bundles all the arguments (however many there are) into a list and passes that list as the actual parameter. You can require some number of arguments by using (**lambda** (*arg1 arg2 . rest-of-args*) ...) – this requires a minumum of 2 arguments, with all args after the first two bundled into a list (the list would be empty if the function is called with only two arguments).

I included this just for fun. You won't be tested on arbitrary-argument functions.

1.2 Removing Lambdas on Slide Specifications

Let's use a similar approach to clean up how we write down slides. Instead of having the programmer write down the **lambda**s, let's write a function that takes the data for a slide and returns the appropriate **lambda**. I'll call the function *myslide* so that we don't create a conflict with our current use of the name *slide* in the **define-struct**:

```
;; myslide : string slide-body → (→ slide)
;; return a function to make a slide
(define (myslide title body)
  (lambda ()
    (make-slide title body)))
```

Using this function, our slide program would look like:

```
(define talk1
  (let ([intro-slide (myslide ...)]
        [arith-eg-slide
         (myslide
          (next-example-title)
          (pointlist-bulleted (list "(+ (∗ 2 3) 6)" "(+ 6 6)" "12")))]
        [func-eg-slide (myslide ...)]
        [summary-slide (myslide ...)])
    (make-talk
     (list (make-display intro-slide)
           (make-timecond 5
                          (list (make-display arith-eg-slide))
                          empty)
           (make-display func-eg-slide)
           (make-display summary-slide)))
```

If we made this change, we would notice an odd behavior – our example numbering is off again! If we delay advancing off the first slide, the second slide has "Example 2" as a title, which is what our *timecond* supposedly fixed. What happened?

Think about this before reading further.

Why did we add the **lambda** in the first place? We needed to prevent Scheme from getting the value of *next-example-title* until run-time. In the call to *myslide*, we removed that protection. Scheme evaluates all arguments to functions before calling the functions. Scheme therefore calls *next-example-title*, passes the result to *myslide*, and then buries the *value* in the lambda. We don't want to bury the value, though, we want to bury the *expression that computes the value*. Our *myslide* function therefore defeats the entire purpose of adding the **lambda** in the first place.

In short, functions won't work here because Scheme always evaluates arguments to functions. In this instance, we need something that adds code **without** evaluating its arguments. In other words, we need macros.

2 What's A Macro?

Think of macros like special rules or patterns that take an expression and rewrite it into another expression *without evaluating any of the pieces*. This is exactly what we want in the *myslide* case. We want to be able to write

```
(myslide
 (next-example-title)
 (pointlist-bulleted (list "(+ (* 2 3) 6)" "(+ 6 6)" "12")))
```

and have Scheme convert it into

```
(lambda ()
  (make-slide
   (next-example-title)
   (pointlist-bulleted (list "(+ (* 2 3) 6)" "(+ 6 6)" "12"))))
```

without evaluating the subexpressions.
How do we do this in Scheme? Let's get the code down first, and then analyze it:

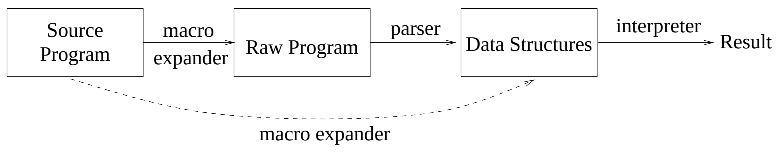
```
(define-syntax myslide
  (syntax-rules ()
    [(myslide title body)
     (lambda ()
       (make-slide title body))]))
```

First, look at the last three lines: we see a pair of square brackets surrounding two expressions. The first expression is the **myslide** expression that we tried to implement using a function; the second expression is the translation that we'd like to have for the first expression. Above that are two lines that introduce new Scheme keywords. The first line says that expressions starting with **myslide** should not be evaluated. The second line is required syntax for defining the translation rules. Never mind the () for now – we'll explain what that's for in due time. For now, just make sure it follows the **syntax-rules**.

This kind of expression that transforms one expression into another is called a *macro*. We'll be learning a lot about macros and how to use them effectively as we continue our exploration of languages.

How do macros work? When you hit Execute, Scheme does a first pass over your code translating all the macros into their corresponding expressions (this is called *macro-expansion*). During macro expansion, Scheme will translate all expressions of the form (**myslide** <*title-expression*> <*body-expression*>) with the corresponding (**lambda** () (*make-slide* ...)) expression without evaluating the <*title-expression*> or the <*body-expression*>. It's that simple. After the macro-expansion pass, Scheme will load your program and Execute it as you are used to so far.

Put more visually, the following diagram shows the stages that happen when going from a program to its execution. A full-fledged language implementation takes the top path; in 1102, we will follow the lower path, basically bypassing the parser (take a compilers course if you want to understand the missing stage better).



Returning to Slideshow

If we add the **myslide** macro definition to the slideshow package, we can now write our talk as we wanted to before, without introducing errors in the example numbering:

```
(define talk1
  (let ([intro-slide (myslide ...)]
        [arith-eg-slide
         (myslide
          (next-example-title)
          (pointlist-bulleted (list "(+ (* 2 3) 6)" "(+ 6 6)" "12")))]
        [func-eg-slide (myslide ...)]
        [summary-slide (myslide ...)])
    (make-talk
     (list (make-display intro-slide)
           (make-timecond 5
                          (list (make-display arith-eg-slide))
                          empty)
           (make-display func-eg-slide)
           (make-display summary-slide)))
```

To finish the slideshow example, we have to see how many of the other annoyances we listed at the beginning of the lecture can be removed either using functions or using macros. Before we do that, though, let's get a better understanding of, and practice with, macros.

2.1 Some Other Macro Examples

As a rule of thumb, we use macros whenever we want to write expressions that *look* like functions, but that don't evaluate in the same way (eval the args first, then eval a body). A macro is just a rule for rewriting one pattern into another. Let's look at two other macro examples, one of which you've been using all term.

2.1.1 Or

Let's pretend that **or** was not a built-in operator, and that you wanted to define it. Here's an attempt using a function (I give it the name *my-or* to avoid conflicts with the built-in **or** operator).

```
;; or : boolean boolean → boolean
;; return true if and only if one of the inputs is true
(define (my-or e1 e2)
  (cond [e1 true]
        [else e2]))
```

What would be some good test cases for *my-or*? Let's try a few and see how this looks:

```
> (my-or (= 3 3) (> 3 3))
true

> (my-or (> 3 4) (= 4 4))
true
```

```
> (my-or (= 3 4) (> 3 4))
false
```

Looks good, right? Let's try one more example: (*my-or* (= 3 3) (= 3 'a)). What answer should you get on this? You should get true, since the first argument evalutes to true. What do you get? You get an error, since you can't use = on a symbol argument. If you tried this same expression using Scheme's **or** instead of *my-or*, you'd get *true*.

From other languages, many of you know that **or** "short-circuits" – as soon as it finds an argument that evaluates to true, it returns true *without evaluating the remaining arguments*. This requirement tells you that **or** can't be implemented as a regular function. It has to be something special. It is; it's a macro.

```
(define-syntax my-or
  (syntax-rules ()
    [(my-or e1 e2)
     (cond [e1 true]
           [else e2])]))
```

How does this macro solve our short-circuit problem? If you recall how Scheme evaluates conditionals, it will only evaluate *e2* if *e1* (the first test) was false. We relied on our knowledge of how Scheme evaluates expressions to implement this macro properly.

2.1.2 Time

Often, we want to determine how long a particular computation took to complete (for performance analysis, for example). For this, it's useful to have a *time* operator that takes an expression and prints out the amount of time spent executing that expression. Given a particular expression, such as (*run-talk talk1*), we could compute the execution time using the following expression:

```
(let ([start-time (current-seconds)])
  (let ([result (run-talk talk1)])
    (begin (printf "Time used: ~a~n" (- (current-seconds) start-time))
           result)))
```

Since we might want to time any number of expressions, we want to parameterize this expression over the computation to time. We have two options: functions and macros. Which should we use and why? If we used a function, we'd write something like:

```
(define (time expr)
  (let ([start-time (current-seconds)])
    (let ([result expr])
      (begin (printf "Time used: ~a~n" (- (current-seconds) start-time))
             result))))

(time (run-talk talk1))
```

This would have the same problem we encountered earlier: Scheme would evaluate (*run-talk talk1*) before calling the time function. That's bad in this case because we don't start measuring the execution time until we get into the body of *time*, after which the expression has already been evaluated (and our chance to measure the time lost).

Let's write this as a macro then:

```
(define-syntax time
  (syntax-rules ()
    [(time expr)
     (let ([start-time (current-seconds)])
       (let ([result expr])
         (begin (printf "Time used: ~a~n" (- (current-seconds) start-time))
                result)))]))
```

```
(time (run-talk talk1))
```

Again, this works because we don't start evaluating *expr* until after we've saved the *start-time* and are actually timing the computation. The difference between macro-expansion time and run-time saves us here.

But wait – couldn't we have written time as a function if we'd used **lambda** to delay when we evaluate the expression? For example, why wouldn't the following non-macro solution have worked?

```
(define (time expr-func)
  (let ([start-time (current-seconds)])
    (let ([result (expr-func)])
      (begin (printf "Time used: ~a~n" (- (current-seconds) start-time))
             result))))

(time (lambda () (run-talk talk1)))
```

What do you think? Would this work?

This approach would indeed let us measure the evaluation time (a performance purist would note that we add a bit of extra time to our measurement though, because we incur the cost of calling the *expr-func* expression inside the body). It sort of misses the point, however, because we'd rather not have to remember to wrap the **lambda** around the function before timing it. If you forget the **lambda**, it's not like the programmer gets an error message, they just get an inaccurate time measurement! Our goal is always to support the programming task, and thereby programmers, as best we can.

Okay, so we still want the macro to make the code cleaner, but we still could have used the **lambda** version though, couldn't we? As in, couldn't we have written **time** with a combination of the function and the macro, as:

```
(define (time-as-func expr-func)
  (let ([start-time (current-seconds)])
    (let ([result (expr-func)])
      (begin (printf "Time used: ~a~n" (- (current-seconds) start-time))
             result))))

(define-syntax time
  (syntax-rules ()
    [(time expr)
     (time-as-func (lambda () expr))]))

(time (run-talk talk1))
```

Could you do this, yes. Does it make sense? No. The macro achieves the same thing as the **lambda**, and notice we needed a bit more code infrastructure to make this work. The macro version is smaller, cleaner, and therefore preferable in this case.