CS1102: Introduction to Macros, part 2

Kathi Fisler, WPI

September 30, 2008

1 Using Macros to Fix More of the Slideshow Language

Now that we have a better idea of how macros work, let's return to cleaning up the slideshow language. We wanted to clean up several parts of the code. We've already gotten rid of the **lambda**s around slides with the *myslide* macro we wrote earlier. We fixed the problem with specifying the booleans in the pointlists by adding some helper functions. That leaves us with two main issues to address: the lists, and the let statement for defining the slides.

1.1 Cleaning Up the Talk Specification

Here's the current talk program, using the format of the posted code:

```
(define talk2
  (let ([intro-slide (myslide ...)]
        [arith-eg-slide
         (myslide
          (make-next-example-title)
          (pointlist-bulleted (list "(+ (* 2 3) 6)" "(+ 6 6)" "12")))]
        [func-eg-slide (myslide ...)]
        [summary-slide (myslide ...)])
    (make-talk
     (list (make-display make-intro-slide)
           (make-timecond (lambda (time-in-talk) (> 5 time-in-talk))
                          (list (make-display make-arith-eg-slide))
                          empty)
           (make-display make-func-eg-slide)
           (make-display make-summary-slide)))))
```

First, let's clean up the talk specification to hide the details of the lists. We want to write a macro *mytalk* that takes a sequence of commands as arguments and adds the outer list structure. The following macro achieves this:

```
(define-syntax mytalk
  (syntax-rules ()
    [(mytalk cmd1 ...)
     (make-talk
      (list cmd1 ...))]))
```

This introduces a new feature of macros: the ellipses. What do they mean and let us do? The ellipses say "there can be any number of the pattern immediately preceding me". In this case, they say "there can be any number of commands following the **mytalk**". Down in the macro body, we can use the ellipses again to say "take all the remaining commands and drop them into the list". We can now rewrite the talk body using **mytalk**, and Scheme will convert it into the same talk (*talk2*) that we had previously.

```
(define talk3
```

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

[**Note** the — in *talk3* are shorthand for the rest of the slide specifications that I've left out of the notes to reduce clutter. They are not any form of valid Scheme, even with macros.]

1.2 Moving Slides Into the Talk

This leaves one last major change to make: we want to make the slide specifications part of the talk, rather than relying on Scheme let in the program text. Here, I'm going to show you the desired program syntax first, then we'll work on the macro to get us there.

```
(define talk4
  (talk-with-slides
   ((slide intro-slide
           "Hand Evals in DrScheme"
           "Hand evaluation helps you learn how Scheme reduces programs to values")
    (slide arith-eg-slide
           (make-next-example-title)
           (pointlist-bulleted (list "(+ (* 2 3) 6)" "(+ 6 6)" "12")))
    (slide func-eg-slide
           (make-next-example-title)
           (pointlist-bulleted (list "(define (foo x) (+ x 3))" "(* (foo 5) 4)"
                                     "(* (+ 5 3) 4)" "(* 8 4)" "32")))
    (slide summary-slide
           "Summary: How to Hand Eval"
           (pointlist-numbered (list "Find the innermost expression"
                                     "Evaluate one step"
                                     "Repeat until have a value"))))
   (make-display intro-slide)
   (make-timecond (lambda (time-in-talk) (> 5 time-in-talk))
                  (list (make-display arith-eg-slide))
                  empty)
   (make-display func-eg-slide)
   (make-display summary-slide)))
```

If we compare this version of the talk to our earlier versions, what will the new *talk-with-slides* macro need to do in order to transform *talk4* into *talk3*?

- It needs to introduce a let where the slide names become the bound variables and the other slide data is passed into *myslide*.

- It needs to use **mytalk** around the commands after the slide specification.
- It needs to handle an arbitrary number of slides and commands.

Let's develop the macro. First, let's write down the part of the macro that defines the input pattern:

```
(define-syntax talk-with-slides
  (syntax-rules ()
    [(talk-with-slides ((slide name1 title1 body1) ...)
                       cmd1 ...)
     <TRANSLATION PATTERN GOES HERE>]))
```

Notice here that we have two sets of ellipses: one to let us specify an arbitrary number of slides and another to specify an arbitrary number of commands. It's important that the slides be wrapped in a pair of parens (or some other syntax) so that the first set of ellipses knows when to stop matching code – in other words, the set of parens around the slides separates the slide specifications from the command specifications.

What goes into the output pattern? Again, look at *talk3* and write down what you need to reproduce the pattern. At the outermost level, we need a **let** statement and a **mytalk** statement in the **let** body.

```
(define-syntax talk-with-slides
  (syntax-rules ()
    [(talk-with-slides ((slide name1 title1 body1) ...)
                       cmd1 ...)
     (let (<FILL IN HERE>)
       (mytalk cmd1 ...))]))
```

Next, ask yourself what goes into the let. The variable names match the variable "name1" in the input pattern:

```
(define-syntax talk-with-slides
  (syntax-rules ()
    [(talk-with-slides ((slide name1 title1 body1) ...)
                       cmd1 ...)
     (let ([name1 <FILL IN HERE>])
       (mytalk cmd1 ...))]))
```

What does the expression corresponding to each name look like? In *talk3*, its a *myslide* pattern.

```
(define-syntax talk-with-slides
  (syntax-rules ()
    [(talk-with-slides ((slide name1 title1 body1) ...)
                       cmd1 ...)
     (let ([name1 (myslide <FILL IN HERE>)])
       (mytalk cmd1 ...))]))
```

What are the arguments to *myslide*? The title and body, both of which have names in our input pattern.

```
(define-syntax talk-with-slides
  (syntax-rules ()
    [(talk-with-slides ((slide name1 title1 body1) ...)
                       cmd1 ...)
     (let ([name1 (myslide title1 body1)])
       (mytalk cmd1 ...))]))
```

Is that all? Not quite. We've put the first slide into the output pattern, but we haven't used the ellipses for the slides (to allow us to specify an arbitrary number of slides. We insert the ellipses at the point in the output pattern where we want to repeat how we handle the slide inputs. Since each additional slide introduces a new **let** variable, we put the slide ellipses inside the let variable declaration area:

```
(define-syntax talk-with-slides
  (syntax-rules ()
    [(talk-with-slides ((slide name1 title1 body1) ...)
                       cmd1 ...)
     (let ([name1 (myslide title1 body1)]
           ...)
       (mytalk cmd1 ...))]))
```

With this, we can write and run *talk4* with our existing interpreter.

1.3 Cleaning Up Timecond

How does our current language look? A lot better. There's one more thing to fix (that will, handily enough, finish our introduction to macros). The *timecond* is still a little clumsy because we have to include the *empty* even when we have no commands in our else case. Let's add macros that make the empty else case optional. First, let's write the macro for when we do have an else case:

```
(define-syntax time-branch
  (syntax-rules ()
    [(time-branch test cmdlist1 cmdlist2)
     (make-timecond test cmdlist1 cmdlist2)]))
```

We can do omit the else case by giving multiple input patterns to **time-branch** with different numbers of arguments. The macro expander will use the first transformation that matches the input pattern. Here's how the macro looks:

```
(define-syntax time-branch
  (syntax-rules ()
    [(time-branch test cmdlist)
     (make-timecond test cmdlist empty)]
    [(time-branch test cmdlist1 cmdlist2)
     (make-timecond test cmdlist1 cmdlist2)]))
```

This gives us yet another revised version of the talk program (this version, available in the posted code, also introduces a function definition to rename *make-display* to *display-slide*, since the latter sounds more like a command than a data structure).

```
(define talk5
  (talk-with-slides
   ((slide intro-slide
           "Hand Evals in DrScheme"
           "Hand evaluation helps you learn how Scheme reduces programs to values")
    (slide arith-eg-slide
           (make-next-example-title)
           (pointlist-bulleted (list "(+ (* 2 3) 6)" "(+ 6 6)" "12")))
    (slide func-eg-slide
           (make-next-example-title)
           (pointlist-bulleted (list "(define (foo x) (+ x 3))" "(* (foo 5) 4)"
                                     "(* (+ 5 3) 4)" "(* 8 4)" "32")))
    (slide summary-slide
           "Summary: How to Hand Eval"
           (pointlist-numbered (list "Find the innermost expression"
                                     "Evaluate one step"
                                     "Repeat until have a value"))))
   (disp-slide intro-slide)
   (time-branch (lambda (time-in-talk) (> 5 time-in-talk))
```

```
                (list (disp-slide arith-eg-slide)))
  (disp-slide func-eg-slide)
  (disp-slide summary-slide)))
```

1.4 What's Left?

We've made a lot of progress on the language since *talk1* (and it didn't take that much work, once you understand how to write macros)! A couple of minor issues lurk in *talk5*, such as the remaining list commands and the **lambda** in the **time-branch**. You could eliminate the lists with some additional macros or tweaks to our current macros.

The **lambda** is harder to get rid of. It's tempting to leave the lambda off the time-branch test and let the macro insert the lambda, as follows:

```
(define-syntax time-branch
  (syntax-rules ()
    [(time-branch test cmdlist)
     (make-timecond (lambda (time-in-talk) test) cmdlist empty)]
    [(time-branch test cmdlist1 cmdlist2)
     (make-timecond (lambda (time-in-talk) test) cmdlist1 cmdlist2)]))

(time-branch (> 5 time-in-talk)
             (list (disp-slide arith-eg-slide)))
```

Unfortunately, this won't work due to a technical issue with macros. You can't have the source code use an undefined identifier and have the **define-syntax** macro introduce that name as a parameter (or a let/local variable). This is by design, and it's a good idea, because otherwise you could have clashes between names introduced in code and names introduced in macros (getting this feature of macros right was an unsolved research problem for a long time in the languages community). It is possible to write macros that do capture names, but that requires another, more complicated, style of macros that isn't worth going into in this course. If you're interested, I'd be glad to point you towards more information on this topic.

2 Recap

These notes introduced you to macros and showed you how to use them to put a cleaner syntax/interface on your data definitions for a language. Here's a summary of the main points you need to take away from this presentation:

- Macros are different from functions. When Scheme evaluates a function call, it evaluates the arguments before evaluating the body. When Scheme expands a macro, it just rewrites one pattern of code into another, *without evaluating anything*.
- Whenever you want a construct that needs to delay evaluating its arguments (such as **or** or *time*, you must use a macro.
- We define macros using **define-syntax**, and a macro specification consists of pairs of input patterns and the output patterns to translate them into.
- Ellipses are used in macros to handle arbitrarily many instances of input patterns.
- Macros can handle multiple forms of the same notation (as we saw in **time-branch**, but each form must start with the same macro name and be distinguishable based on the pattern of the syntax (i.e., you can't rely on *number?* or *symbol?* to tell one pattern from another in a multi-armed macro.

In terms of skills for the course, I will expect that you are able to:

- Identify when you need a macro to implement a particular construct.

- Write your own macros of similar complexity to the ones introduced here.

We will gain more practice with macros as we go through the remaining language exercises in the course.