CS2135: Introduction to Macros

Kathi Fisler, WPI

February 4, 2003

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
            (format "Example ~a" example-index)
            (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 (lambda (time) (> 10 time))
                          (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
            (format "Example ~a" example-index)
            (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 (lambda (time) (> 10 time))
                          (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*.

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
          (format "Example ~a" example-index)
          (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 (lambda (time) (> 10 time))
                          (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! Somehow, both example slides have the title "Example 0". 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 *example-index* until run-time. In the call to *myslide*, we removed that protection. Scheme evaluates all arguments to functions before calling the functions. Scheme therefore gets the value of *example-index*, passes that value 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 a way to write a function that **doesn't** evaluate its arguments first. 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
 (format "Example ~a" example-index)
 (pointlist-bulleted (list "(+ (* 2 3) 6)" "(+ 6 6)" "12")))
```

and have Scheme convert it into

```
(lambda ()
  (make-slide
   (format "Example ~a" example-index)
   (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:

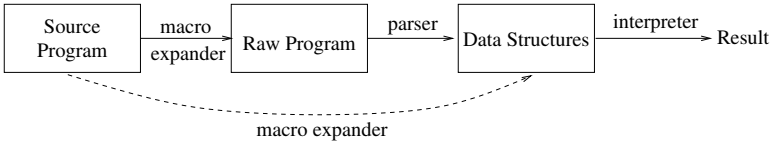
```
(define-syntax myslide
  (syntax-rules ()
    [(myslide title body)
     (lambda ()
       (make-slide title body))]))
```

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 2135, 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
          (format "Example ~a" example-index)
          (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 (lambda (time) (> 10 time))
                          (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.

3 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.

> **Note:** at this point I'm going to switch over to the version of the talk language that includes the section structures, as shown in the posted code. This is for consistency with those files, so we can type in and try these macros with the existing body of code.

3.1 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
          (format "Example ~a" example-index)
          (pointlist-bulleted (list "(+ (* 2 3) 6)" "(+ 6 6)" "12")))]
        [func-eg-slide (myslide ...)]
        [summary-slide (myslide ...)])
    (make-talk
     (list
      (make-section
       (list (make-display make-intro-slide)
             (make-timecond (lambda (time-in-talk) (> 5 time-in-talk))
                            (make-section (list (make-display make-arith-eg-slide)))
                            (make-section 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 and the sections. We want to write a macro *mytalk* that takes a sequence of commands as arguments and adds all the outer list and section structure. The following macro achieves this:

```
(define-syntax mytalk
  (syntax-rules ()
    [(mytalk cmd1 ...)
     (make-talk
      (list
       (make-section
        (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
          (format "Example ~a" example-index)
          (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))
                    (make-section (list (make-display make-arith-eg-slide)))
                    (make-section empty))
     (make-display make-func-eg-slide)
     (make-display make-summary-slide))))
```

[**Note** that the ellipses in *talk3* are **NOT** macro ellipses – they are informal shorthands for the rest of the slide specifications that I've left out of the notes for sake of space. Scheme can only process ellipses when they occur inside macro definitions.]

3.2 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))
                  (make-section (list (make-display arith-eg-slide)))
                  (make-section 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.

3.3 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. We have to remember to put in the *section* commands. Let's introduce a macro *time-branch* to handle the section commands for us:

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

With this, I can write the *timecond* expression as

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

This is a bit cleaner, but wouldn't it be nice if we could just leave off the empty case entirely, and have *timebranch* automatically insert the *empty* if we only provide one list of commands? We can do this 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 (make-section cmdlist) (make-section empty))]
    [(time-branch test cmdlist1 cmdlist2)
     (make-timecond test (make-section cmdlist1) (make-section 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)))
```

3.4 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, for reasons that are a bit too complicated to explain now. If you want the details, come by my office and I'll be glad to explain them to you.

4 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 have an operator 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.