CS1102: Extending the Slideshow Language

Kathi Fisler and Charles Rich, WPI

September 18, 2016

1 Introduction: Design by Successive Refinement

In the last lecture, we finished implementing a prelimary version of a slideshow program. The language is pretty simple – all we can do is issue commands to display slides in a linear order. Still, we had to develop a lot of infrastructure code to get this much of the system working. We now have a simple interface (which you would have had to develop yourself if you were designing this language), data structures for slides, and the core of a program that can handle slide displays. You'd need all of this to develop the full program anyway, so we're off to a good start.

We are developing our slideshow language through a process known as *iterative refinement*. We didn't tackle the entire system at once. Instead, we carved out a small yet cohesive part of the full slideshow program, then designed, implemented and tested it. We finished this implementation before adding more features. This approach is valuable for many reasons:

- It helps you concentrate on one part of the system.
- It forces you to understand some of the issues that will affect your entire system design, but without the complexity of the whole system.
- You end up with a collection of working code that you can try to reuse as you add more features to the system. Note that you can't always reuse your existing code (depending on what features you add later), but you can often reuse a lot of it and you can almost always reuse the *insights* you developed while working on that code.

Basically, iterative refinement encourages you to develop a system in small stages, where each stage builds upon earlier stages. Such early-stage systems are often called *prototypes*. As you develop languages for this course (and your entire career, for that matter), practice isolating small pieces to implement first and building on those pieces later.

With that said, let's extend the slideshow language!

2 The First Refinement: Timed Conditionals

Imagine you are giving a talk using your slideshow package. You're not really sure how much material to prepare – if your audience asks a lot of questions, you may be running short on time, but you want to have enough material to keep talking in case they don't ask many questions. Ideally, you'd like your talk program to be conscious of the time and to skip over certain collections of slides depending upon how much time remains.

Let's extend our slideshow package with this functionality. We need to add a *timecond* construct to our language. This construct is like an **if**-statement from other languages you've studied. It will perform some test on the time that's elapsed since you started speaking, then choose which set of slides to run next depending on the result of the test.

What might this look like in our talk program? Recall that our previous talk program looked like:

```
(make-talk
 (list (make-display intro-slide)
       (make-display arith-eg-slide)
       (make-display func-eg-slide)
       (make-display summary-slide)))
```

(where the various slides were defined in the previous set of notes, and in the posted code). Let's say we want a different talk that would skip the *arith-eg-slide* if more than 10 seconds has elapsed since we started talking. How might we write that talk down? We need something like an **if**-statement in our language:

```
(make-talk
 (list (make-display intro-slide)
       (make-timecond <TEST>
                      (make-display arith-eg-slide)
                      <OTHERWISE>)
       (make-display func-eg-slide)
       (make-display summary-slide)))
```

This is the right idea, but we have to fill in the TEST and OTHERWISE details. In our example scenario, we don't want to run anything in the OTHERWISE case – we just want to go onto the *func-eg-slide*. What could we put in place of OTHERWISE? We could use something like *false*, but we're better off using *empty* and letting each branch of the *timecond* be a list. Why? Maybe we have a group of related slides, and we have to choose to show or skip all of them at once. Lists would let us do that. This gives us the program:

```
(make-talk
 (list (make-display intro-slide)
       (make-timecond <TEST>
                      (list (make-display arith-eg-slide))
                      empty)
       (make-display func-eg-slide)
       (make-display summary-slide)))
```

Now, we just need to figure out what we'd put in the TEST position. A first temptation is to put a number there. In our motivating scenario, we wanted to skip the slide if we had spoken for more than 10 seconds, so maybe we should write:

```
(make-talk
 (list (make-display intro-slide)
       (make-timecond 10
                      (list (make-display arith-eg-slide))
                      empty)
       (make-display func-eg-slide)
       (make-display summary-slide)))
```

and have the *timecond* statement run the first set of slides if the elapsed time is less than the given number and the second set of slides otherwise.

This is a good initial approach for adding *timecond* to the language. But this just shows how we want to add it to our programs – we need to edit the data definitions and interpreter to handle this change.

Augmenting the Data Definition

Where should *timecond* go in our data definition? From the example, it appears to belong with the commands, since a talk contains a list of commands (notice that *timecond* is at the same level as *display*, so it should go in a similar place in the data definition). How would you have known this if we hadn't started with the example? *Any time you add a new operator or construct, it's going to go into the command area of your data definition* (every language definition needs a data definition for its commands).

Here's our previous data definition for commands:

```
;; A cmd is
;; - (make-display slide)
(define-struct display (slide))
```

We need to add a space for *timecond* that matches the example we developed above:

```
;; A cmd is
;; - (make-display slide)
;; - (make-timecond number list[cmd] list[cmd])

(define-struct display (slide))
(define-struct timecond (seconds within-time over-time))
```

[Note: the posted code introduces a new data definition called *section*, and uses that in the definition of *timecond* instead of *list*[*cmd*]. Either would work. I created the *section* structure in case we later wanted to add other attributes to individual sections of a talk.]

Augmenting the Interpreter

Now that we have the data definition for *timecond*, we can extend the interpreter to support it. Where should we make the change? Notice we have edited only one data definition: the one for commands. All the other data definitions have remained intact. This tells us *exactly* where to edit the existing interpreter code: the *run-cmd* function, the template function that processes the command data definition, has to change.[1]

How do we get started? Same way we always do: fill in the template and see what that suggests:

```
;; run-cmd : cmd → void
;; executes the given command
(define (run-cmd cmd)
  (cond [(display? cmd)
         (begin (print-slide (display-slide cmd))
                (await-click))]
        [(timecond? cmd)
         (timecond-seconds cmd) ...
         (run-cmdlist (timecond-within-time cmd)) ...
         (run-cmdlist (timecond-over-time cmd)) ... ]))
```

Note that the template gives us the calls to *run-cmdlist* on the two sets of slides – the template code takes care of running those commands for us. We just have to determine when to run those commands.

What is (*timecond-seconds cmd*)? The data definition tells us that it is a number. What do we want to do with it? We want to compare it to the current elapsed time. Let's assume we can write a function to give us that number (call that function *elapsed-time*). Add that to our working code:

```
;; run-cmd : cmd → void
;; executes the given command
(define (run-cmd cmd)
  (cond [(display? cmd)
         (begin (print-slide (display-slide cmd))
                (await-click))]
        [(timecond? cmd)
         (elapsed-time) ...
         (timecond-seconds cmd) ...
         (run-cmdlist (timecond-within-time cmd)) ...
         (run-cmdlist (timecond-over-time cmd)) ... ]))
```

How do we combine the time limit with the elapsed time? Less than or equal!

```
;; run-cmd : cmd → void
```

[1]This example helps illustrate why we put so much emphasis on templates in this course. If you design programs to have the same structure as your data, it's easy to decide where to edit your code when the definitions change.

```
;; executes the given command
(define (run-cmd cmd)
  (cond [(display? cmd)
         (begin (print-slide (display-slide cmd))
                (await-click))]
        [(timecond? cmd)
         (<= (elapsed-time) (timecond-seconds cmd)) ...
         (run-cmdlist (timecond-within-time cmd)) ...
         (run-cmdlist (timecond-over-time cmd)) ... ]))
```

The result of the comparison is a boolean. What does it tell us? Which set of commands to run. This suggests that we need a **cond** to finish writing *run-cmd*:

```
;; run-cmd : cmd → void
;; executes the given command
(define (run-cmd cmd)
  (cond [(display? cmd)
         (begin (print-slide (display-slide cmd))
                (await-click))]
        [(timecond? cmd)
         (cond [(<= (elapsed-time) (timecond-seconds cmd))
                (run-section (timecond-within-time cmd))]
               [else (run-section (timecond-over-time cmd))])]))
```

That's it! We now have time conditionals implemented in our language. Grab the file `ppt-stage2.rkt` from the webpage to try this code out. The code also shows you how to implement the *elapsed-time* function.

3 The Second Refinement: Dynamic Content

Run the talk program with *timecond* twice: once exercising each branch of the conditional. If you look closely, you'll notice a new problem: if we do skip the *arith-eg-slide*, the first example shown (the *func-eg-slide*) has title "Example 2", even though there was no example 1. This is problematic – the whole point of adding *timecond* is to let a speaker change the talk content "behind the scenes" (instead of frantically pressing the space bar as many people do when using powerpoint when they are running out of time). For the talk to look truly smooth, however, we'll need to be able to adjust the example numbers as well.

If we want to achieve example numbers that match the sequencing under *timecond*, when can we assign example numbers to slides? There are two choices: when we write the program, or when we run the program (the latter is called *run-time*). We need to set the example numbers at run-time. You could try to write a program that copied the body of the *func-eg-slide* with different slide numbers and put the copies in the right places in the talk program, but a sufficiently large program with many *timecond* statements would quickly show you that this approach doesn't scale. Since the example number depends on speaker (run) time, we will need to decide on those numbers at run-time.

How do we do this in the program? Let's go back to our talk program and try some alternatives:

```
(define talk1
  (let ([intro-slide (make-slide ...)]
        [arith-eg-slide
         (make-slide
          "Example 1"
          (make-pointlist (list "(+ (* 2 3) 6)" "(+ 6 6)" "12") false))]
        [func-eg-slide (make-slide ...)]
        [summary-slide (make-slide ...)])
      (make-talk
       (list (make-display intro-slide)
        (make-timecond 20
                       (list (make-display arith-eg-slide))
                       empty)
        (make-display func-eg-slide)
        (make-display summary-slide)))))
```

Clearly, the number 1 has to come out. What can we use in its place? How about a variable for the count of examples that we have displayed so far (we'll call this variable *example-index*). Can we just replace the 1 with *example-index* as follows?

```
(define talk1
  (let ([intro-slide (make-slide ...)]
        [arith-eg-slide
         (make-slide
          "Example example-index"
          (make-pointlist (list "(+ (* 2 3) 6)" "(+ 6 6)" "12") false))]
        [func-eg-slide (make-slide ...)]
        [summary-slide (make-slide ...)])
    (make-talk ...)))
```

Of course not. This puts *example-index* as a sequence of characters into the string, and we want to put the *contents* of *example-index* into the string. We could just take the variable name out of the string as in:

```
(define talk1
  (let ([intro-slide (make-slide ...)]
        [arith-eg-slide
         (make-slide
          "Example" example-index
          (make-pointlist (list "(+ (* 2 3) 6)" "(+ 6 6)" "12") false))]
        [func-eg-slide (make-slide ...)]
        [summary-slide (make-slide ...)])
    (make-talk ...)))
```

This is problematic because it changes the number of arguments to make-slide, and we don't want to have to make two kinds of slides (example slides and non-example slides) [*why not?*].

Racket provides an operator called *format* that builds a string out of data values. The following code creates a single string containing the string Example followed by the example-index (see the DrRacket helpdesk for more information on *format*).

```
(define talk1
  (let ([intro-slide (make-slide ...)]
        [arith-eg-slide
         (make-slide
          (format "Example ~a" example-index)
          (make-pointlist (list "(+ (* 2 3) 6)" "(+ 6 6)" "12") false))]
        [func-eg-slide (make-slide ...)]
        [summary-slide (make-slide ...)])
    (make-talk ...)))
```

How does this solution look? It matches the data definition of a slide because *format* returns a string. It uses the value of *example-index* to build the string. Assuming we defined the variable *example-index* and increment it each time we display an example slide, this looks pretty good.

Or does it? What value would *example-index* have when we make the title string for the *arith-eg-slide*? Or, a related question, *when would Racket call the format command*? We want that call to happen at run-time, just before we display the slide. Is that when it happens?

To answer this, you need to remember how Racket evaluates functions. Racket will define *talk1* when you hit execute – this is *before* you run the program (at what we call *compile-time* – when your programming environment loads or processes your program prior to running it). So, Racket will call *format* and grab the value of *example-index* when you hit Execute, **not** when you run the talk1 program through the interpreter. This is not what we wanted to have happen! How can we prevent it?

We said that Racket evalutes expressions and defines variables when you hit Execute. Consider what happens if you write a function like

```
;; square : number → number
(define (square n)
  (* n n))
```

and hit Execute. Does Racket perform the multiplication at this time? No, it can't, because you haven't yet called the function, you only defined it. This is a crucial point for you to understand about how programs are run.

> *As a general rule, languages do not evaluate the bodies of functions until you call them. In particular, function bodies are not evaluated at function-definition (compile) time.*[2]

How does this help us? If we put the reference to *example-index* inside a function, and we don't call that function until its time to display the slide, then Racket will use the current run-time value of *example-index* to format the slide titles. In other words, somewhere we need to wrap a **lambda** around the use of *example-index*.

Where can we put the lambda? Putting it just around the *example-index*, as in (*format* "Example ~a" (**lambda** () *example-index*)) won't work because then *format* would try to put the lambda, not the number, into the string. We want to put the **lambda** someplace that keeps our data definition for slides consistent across all of the slides. Using this criterion, the best place for the **lambda** is outside the call to *make-slide*, as follows:

```
(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 ...)))
```

What did this change accomplish? Each slide is now a function, not a structure. When Racket evaluates the definition for *talk1*, it will not go into the bodies of the lambdas, so it will not look at the value for *example-index*. In other words, *we are using lambda to delay the evaluation of the slide contents until run-time*. Remember we discussed this idea when we talked about functions and **lambda**s a couple of weeks ago: functions are useful for delaying computations that we know now, but want to perform later.

But wait – doesn't this change our data definition for slides? And don't we have to actually *call* these new functions somewhere? Yes on both counts. What's changed in the data definition? The display commands now take a function that takes no arguments and *returns* a slide, rather than a slide structure:[3]

```
;; A cmd is
;; - (make-display (→ slide))
;; - (make-timecond number list[cmd] list[cmd])
```

How does this change impact the interpreter? Remember the correspondence between data definitions and code: the change goes in the *display?* case of *run-cmd*. What does the change entail? Before we call *print-slide*, we need to

[2]This idea that programs are processed in different, identifiable times, is one theme that will recur through this class – keep this in mind.

[3]You could also change the definition of a slide to be a function and leave the definition of display intact. That would be better in some ways, but for consistency with the code handed out in class we'll stick to changing the definition of display for now.

run the function in the display to get the slide structure (notice the extra set of parens around the call to *display-slide* in the *display?* case now):

```
;; run-cmd : cmd → void
;; executes the given command
(define (run-cmd cmd)
  (cond [(display? cmd)
         (begin (print-slide ((display-slide cmd)))
                (await-click))]
        [(timecond? cmd)
         (cond [(<= (elapsed-time) (timecond-seconds cmd))
                (run-cmdlist (timecond-within-time cmd))]
               [else (run-cmdlist (timecond-over-time cmd))])]))
```

The posted code (`ppt-stage3.rkt`) shows the full code along with the code that increments the *example-index* variable.

4 Recap

We've done a brain-full of work in this lecture. Let's step back for a moment, recall what we did and what techniques we learned.

What We Did

1. We added time conditionals to our slideshow language.
2. We modified the slideshow program to generate some data (in this case the example numbers) at run-time.

Techniques We Learned

1. Add new constructs to the command data definition for our language (every language we define will have a command data definition).
2. When you add a new construct, think about what information a programmer needs to write down for that construct, develop some examples, then write data definitions to support your ideas.
3. By using **cond** in our implementation, we can implement conditional-like expressions in our language.
4. We can use **lambda** to delay a computation until run-time. Simply wrapping **lambda** around an expression prevents Racket from evaluating that expression. However, if we do this we need to edit our code to call the new function when we do want to evaluate the expression.
5. Adding new constructs to a language has fairly local impact on code – we only edit the code in the places corresponding to where we edited the data definitions.

Download the code. Run it. Move the lambdas around and see what breaks. Try the approaches we discussed but didn't take and make sure you understand why we didn't take them. Hopefully, the techniques and approaches we used in this example will become clearer that way.

5 Are We Done Yet?

We have finished implementing our first software package using a languages-like approach. How close have we come to implementing a real programming language? We've done the heart of the work, but some small issues still remain:

- *The syntax still looks funny.* What does a program look like in this language? If you wanted your roommate to program in your language, what would (s)he have to write? Currently, something like:

```
(make-talk
 (list (make-display intro-slide)
       (make-timecond 10
                      (list (make-display arith-eg-slide))
                      empty)
       (make-display func-eg-slide)
       (make-display summary-slide)))
```

  That doesn't look very much like a programming language – the notation is pretty clumsy and relies on all these make constructs. Real languages don't look much like this, so what's missing?

  What's missing is a cleaner notation (syntax) for writing down slide programs, and a translator from that syntax into our data definitions. In a real language, you might want to write something more like:

```
(talk (display intro-slide)
      (before 10 (display arith-eg-slide))
      (display func-eg-slide)
      (display summary-slide))
```

  (or, maybe even something without parentheses!) You would then need a program (called a *parser*) that would translate this cleaner notation into the *make-display*, etc version that we've been using. We will get into this a little in the next couple of weeks (though this is a big topic and we can't do it justice in 1102).

  That said, the work we've done so far includes everything EXCEPT the nicer notation and the parser. Just under the syntax of every language lies a series of data definitions like the ones we developed here. Interpreters are written to process those data definitions. You have done a real language implementation, minus prettying up the notation.

- *Given that we used **lambda** in our programs, aren't we relying pretty heavily on Racket here?* Yes and no. Yes in that we are definitely relying on Racket (C++ doesn't have **lambda**), but no in that every language that you would implement the slideshow package in has constructs that would achieve what **lambda** did for you here. The general techniques we're learning (such as delaying computation to get run-time content generation) apply across the board. You just need to learn how to implement those techniques in your implementation language of choice.

  This also ties into the previous question: once you put the prettier notation on front of the language, that notation would transfer across more implementation languages than our current *make-display*, etc notation would. But that's another topic for another day.