CS2135: Implementing the Slideshow Language

Kathi Fisler, WPI

January 30, 2003

0.1 Implementing the Prototype

Last class, we developed a data definition for programs in a slideshow language. The notes for that lecture show the data definitions. Now, we want to implement the language.

To implement the language, we need a program that consumes a talk and runs it – let's call this function *run-talk*. In the posted code file (ppt-stage1.scm), you'll find a small collection of functions for providing a text-based interface to slideshows. Any unknown Scheme functions in the following code are given in the posted file, and just handle various forms of printing. What's the contract on *run-talk*?

```
;; run-talk : talk → void
;; executes the commands in a talk then displays end-of-show message
```

What now? *Fall back on the templates!* We know that a talk contains a list of commands, so we'll need a function that runs all the commands in a list:

```
;; run-talk : talk → void
;; executes the commands in a talk then displays end-of-show message
(define (run-talk a-talk)
  (begin
    (run-cmdlist (talk-cmds a-talk))
    (end-show)))

;; run-cmdlist : list[cmd] → void
;; executes every command in a list
(define (run-cmdlist cmd-lst)
  (cond [(empty? cmd-lst) void]
        [(cons? cmd-lst)
         (begin
           (run-cmd (first cmd-lst))
           (run-cmdlist (rest cmd-lst)))]))

;; run-cmd : cmd → void
;; executes the given command
(define (run-cmd cmd)
  (cond [(display? cmd) ...]))
```

Note we still haven't figured out how to implement the display command – we're just writing down templates and creating new functions to process all of the functions involved in the datatype for programs. Now that we're down to *run-cmd* though, we need to figure out how to implement the display command.

What should the display command do? Print out the slide contents, then wait for a click from the user before proceeding. We can easily fill in the code with these operations:

```
;; run-cmd : cmd → void
;; executes the given command
(define (run-cmd cmd)
  (cond [(display? cmd)
```

```
        (begin (print-slide (display-slide cmd))
               (await-click))]))
```

Finally, we just need to implement *print-slide*. For that, we use the library of printing routines, and follow the data definition for slides so we know what we need to print;

```
;; print-slide-title : string → void
;; displays title of slide on screen
(define (print-slide-title title-str)
  (print-string (string-append "Title: " title-str))
  (print-newline))

;; print-slide-body : slide-body → void
;; displays contents of body on screen
(define (print-slide-body body)
  (cond [(string? body) (print-string body)]
        [(pointlist? body)
         (cond [(pointlist-numbered? body)
                (print-numbered-strings (pointlist-points body) 1)]
               [else (print-unnumbered-strings (pointlist-points body))])]))

;; print-slide : slide → void
;; displays contents of slide on screen
(define (print-slide aslide)
  (begin
    (print-string "————————————————————")
    (print-slide-title (slide-title aslide))
    (print-slide-body (slide-body aslide))
    (print-string "————————————————————")))
```

The *run-talk* program and all of its subprograms collectively form an *interpreter*: they interpret programs (talks) by executing them.

Recap

Now that we have this mass of code, let's step back and summarize what we did:

1. Identify the data, operations, control, and programs for the domain we're trying to write a language for.
2. Create data definitions for programs, which requires data definitions for the data, operations, and control.
3. Implement a program (the interpreter) that takes a program (from our data definitions) and executes it according to what we want that language to do. If you followed the templates, much of this code wrote itself, saving you to just think about how to implement the individual operations (which is the interesting part anyway).

   Every time you implement a language, you will go through these steps!

And Finally ...

Congratulations! You've implemented your first language. In the next couple of lectures, we'll refine our prototype by adding new features and seeing how those affect the implementation.