Overview of Web Programming

Kathi Fisler, WPI

February 23, 2004

This lecture looks at programming for web scripts and some of the challenges it raises. These notes accompany a powerpoint presentation that is also available on the syllabus page.

1 The Problem of Web Programming

Work through the Powerpoint file through the slide called "A Central Problem"

Ideally, we would like to draw analogies between web programs and the non-web programs that you write. Some analogies are obvious:

- Boxes on forms correspond to requesting inputs (like *read* in Scheme).
- Web pages/scripts are like functions: they request a number of inputs from a user and perform some operations on the input, possibly calling other functions (pages/scripts) for additional processing.
- Pressing buttons on web pages calls functions.

The "central problem" slide points out a critical way in which this analogy breaks down. Unlike functions, web scripts terminate as soon as they request inputs from users (as soon as they display the boxes for the web form and read the inputs). The user must press a (submit) button to continue the computation. This means that you have to change how you write programs when you implement web programs.

Let's illustrate this with a simple example. Suppose you wanted to write a program that asks a user to input their age, then displays some information about their ability to vote. Keeping with the idea that we need one page to request input, then another to display the output, we might write a Scheme version of this program as follows:[1]

```
;; request-age-page : → void
;; prompts user to input their age
(define (request-age-page)
  (begin (printf "Enter your age: ")
         (read)))

;; age-page-nonweb : → void
;; displays ability to vote based on user's age
(define (age-page-nonweb)
  (local ((define age (request-age-page)))
    (cond [(>= age 18) (printf "Don't forget to vote!")]
          [else (printf "You'll be able to vote in ~a years" (- 18 age))])))
```

If this were a real web program, the user would get a page with a box in which to enter their age and a submit button. Pressing submit would bring up a new page with the appropriate message from the cond statement. Running it in Scheme would yield the following interaction:

[1] we will write our web programs in Scheme because not everyone in the class knows CGI or PHP programming. It's not hard to translate these programs into equivalent ones in your favorite web programming language.

```
> (age-page-nonweb)
Enter your age: 16
You'll be able to vote in 2 years
```

This program has functions that correspond to pages in a web program. Those functions don't follow the termination behavior of web scripts though. We said that web scripts print out pages, read inputs, and then terminate. The *request-age-page* appears to do that (since nothing happens after the *read*). Terminate is a stronger condition though: *web programs don't even return control flow back to the programs that called them!*

In order to study web programs through Scheme, we need a way to define scripts that look like Scheme functions, but abort when they are done, rather than returning to the programs that called them. This is a change to the way Scheme usually handles functions, so we need a macro for defining scripts. The following code achieves this task. **I do not expect you to understand how this macro works — just copy it into your Scheme file when you are experimenting with scripts**. (Of course, if you *want* to know how it works, stop by my office sometime.)

```
(define abort #f)
(let/cc grab-abort
  (set! abort grab-abort))

(define-syntax define-script
  (syntax-rules ()
    [(define-script (script-name arg ...) body)
     (define (script-name arg ...)
       (abort body))]))
```

Let's use the new macro to define our age program as scripts instead of scheme functions. To do this, change **define** to **define-script** on both functions (I also edited the name of the main function so we can tell them apart).

```
;; request-age-page-script : → void
;; prompts user to input their age
(define-script (request-age-page-script)
  (begin (printf "Enter your age: ")
         (read)))

;; age-page : → void
;; displays ability to vote based on user's age
(define-script (age-page-web)
  (local ((define age (request-age-page-script)))
    (cond [(>= age 18) (printf "Don't forget to vote!")]
          [else (printf "You'll be able to vote in ~a years" (- 18 age))])))
```

Now, let's run the program again. The script version should yield the same answers as the original version:

```
> (age-page)
Enter your age:
```

What happened? The *request-age-page-script* program aborted as soon as it finished, rather than return control to the *age-page* program. The **local** in *age-page-web* never finished, because the program aborted at the end of *request-age-page-script*.

This is the problem of programming on the web. Perhaps this looks bizarre to you, but this really is how web scripts work in practice. Over the next three classes, we will show you how to program in this style, and a step-by-step process you can follow to convert programs to ones that will work as scripts.

2 Fixing the Age-Page Program

Let's try to figure out how to fix the script versions of the *age-page* program so that they behave the same way as the original program. If we want the **local** to execute, we have to make sure it gets invoked before *request-age-page-script* terminates. One obvious way to do this is to move the **local** inside *request-age-page-script*, as follows:

```
;; request-age-page-script : → void
;; prompts user to input their age
(define-script (request-age-page-script)
  (begin (printf "Enter your age: ")
         (local ((define age (read)))
           (cond [(>= age 18) (printf "Don't forget to vote!")]
                 [else (printf "You'll be able to vote in ~a years" (- 18 age))]))))

;; age-page : → void
;; displays ability to vote based on user's age
(define-script (age-page)
  (request-age-page-script))
```

This violates the spirit of web scripts though, because scripts are only supposed to request inputs or display messages based on information entered in previous pages. Put another way, by putting the **local** in the same function as the request for input, we've taken out the "submit" button from the web page. We need another way to do this.

We said earlier that submit buttons resemble calling functions. Let's capture this in the code my moving the **local** into another script that gets called after we ask the user for input:

```
;; submit-age : → void
;; reads the age the user entered and displays the voting status
(define-script (submit-age age)
   (cond [(>= age 18) (printf "Don't forget to vote!")]
         [else (printf "You'll be able to vote in ~a years" (- 18 age))]))

;; request-age-page-script : → void
;; prompts user to input their age
(define-script (request-age-page-script)
  (begin (printf "Enter your age: ")
         (submit-age (read))))

;; age-page : → void
;; displays ability to vote based on user's age
(define-script (age-page)
  (request-age-page-script))
```

Running this version yields the desired interaction:

```
> (age-page)
Enter your age: 16
You'll be able to vote in 2 years
```

Why did this version work? Notice that before *request-age-page-script* can terminate, it must call *submit-age*, which continues the computation. The *submit-age* script displays the voting information to the user, then aborts. Control never gets back to *request-age-page-script*, but that's okay, because it didn't have more work to do anyway! This example illustrates how to program for the web: each script calls another script to continue the computation just before it would otherwise terminate. No other computation can depend on the answer from a script.

3 Returning to Adding Numbers

The Powerpoint slides that we used to start the lecture were doing an example of a simple adder that requests two numbers on two separate pages and displays their sum. Let's try to write sufficient scripts for this example. We'll start with a conventional version of the program.

```
;; request-num1-page : → number
;; requests the first number
```

```
(define (request-num1-page)
  (begin (printf "Enter first number: ")
         (read)))

;; request-num2-page : → number
;; requests the second number
(define (request-num2-page)
  (begin (printf "Enter second number: ")
         (read)))

;; adder-page : → void
;; requests two numbers from user and displays their sum
(define (adder-page)
  (local ((define n1 (request-num1-page)))
    (local ((define n2 (request-num2-page)))
      (printf "sum: ~a~n" (+ n1 n2)))))
```

To convert this to a web program, we need to make sure that each request page starts the next stage of the computation before it finishes, and we need to change each **define** to **define-script**. In the age program, we sent each *read* to a *submit* function that continued the computation. Let's do the same here:

```
;; request-num1-page : → number
;; requests the first number
(define-script (request-num1-page)
  (begin (printf "Enter first number: ")
         (submit1 (read))))
```

What should *submit1* do? If we look at the original *adder-page* function, after we request the first number we request the second number (and send it off to a script as well)

```
(define (submit1 n1)
  (begin (printf "Enter second number: ")
         (submit2 (read))))
```

What should *submit2* do? Since both numbers have been requested, it can now print the sum:

```
(define-script (submit2 n2)
  (printf "sum: ~a~n" (+ n1 n2)))
```

Running this version yields the following interaction:

```
> (adder-page-web)
Enter first number: 5
Enter second number: 8

[BUG] reference to undefined identifier: n1
```

Where's the problem? Notice that *submit2* tries to print the sum of *n1* and *n2*, but it doesn't have *n1* (which was read in as part of *submit1*). To fix this, we pass *n1* along as a parameter to *submit2*:

```
;; adder-page-web : → void
;; requests the first number
(define (adder-page-web)
  (begin (printf "Enter first number: ")
         (submit1 (read))))

;; submit1 : number → void
;; requests the second number
(define (submit1 n1)
  (begin (printf "Enter second number: ")
         (submit2 n1 (read))))

;; submit2 : number number → void
;; get second number from user and display sum
(define-script (submit2 n1 n2)
  (printf "sum: ~a~n" (+ n1 n2)))
```

This version runs as expected.

For those of you with web programming experience, *n1* would be handled as a hidden variable. HTML supports hidden variables for precisely this reason: breaking programs into web scripts requires a way to pass values between scripts.

4 Onward

We've seen a couple of simple examples of converting programs to the web. The problem gets harder when the programs involve more conditionals, recursive functions, and other features. You'll see how to address these problems systematically in the next two lectures.

For now, return to the PPT presentation and follow it along to the end.