1 Introduction

This series of lectures explores how Web programming is intimately tied to material we cover in this course. We start off with some warm-up material in this lecture, then get progressively more complex as the week progresses.

We’ll begin with a solution to Part 1 of the first assignment. Leaving out a few small details, our program might have looked like the one shown in figure 1.

The structure of this program should be pretty self-evident. We run the program by invoking (go). At each stage, an invocation of prompt-read-many returns one or more values. The following series of let’s associates helpful names to the values read. Finally, we display the result of enrolling.

Now compare this to your solutions to part 2 of the homework. Once you strip away the syntax and minor linguistic differences, you should still notice a good deal is different. Can you catalog these differences?

• You probably wrote several programs, not one. If you did write one program, it probably had to check (probably by examining which fields had values) whether it was being invoked to supply personal information or enrollment choices. Once it determined this, it branched off into different functions. Notice the program above doesn’t do this.

• Your program had to involve a loop somewhere, to read multiple course selections. In Scheme, we recursively invoke select-courses. The recursion in the Web program looks rather different, and just a little strange, since it involves putting the right URL in the action field of a form.

• In the Scheme program, the key function, select-courses, consumes no arguments. Your Web equivalent had to consume several implicit arguments in the form of hidden fields (or some other such technique). Where did these arguments disappear in Scheme or, equivalently, why did they materialize on the Web?

Depending on the nature of your solution, you will probably notice even more differences.

All this forces us to ask a fundamental question: are these two kinds of interactive programs essentially different, or are they really just the same program
;; Eliding definitions for filter and prompt-read

(define (go)
  (let ([answers (prompt-read-many (list "Enter your name: 
                      "Enter your street address: 
                      "Enter your email address: " ))])
    (let ([name (first answers)])
      [street (second answers)]
      [email (third answers)])
    (local
      [(define ALL-COURSES '(cs16 cs18 cs22 cs31 cs32))
       (define (display-result courses)
         (printf "Name: "n"Street: "n"Email: "n"Final Choice: "n"
                    name street email courses))
       (define (select-courses)
         (local
           [(define (helper taking)
               (begin
                 (printf "Name: "n"Street: "n"Email: "n"
                    name street email)
                 (printf "Enrolled: "n" taking)
                 (printf "Remaining: "n" (filter (lambda (course) 
                                               (not (member course taking)))
                                               ALL-COURSES)))
               (let ([answers (prompt-read-many (list "Next operation (add/drop/quit): 
                                               "Course name: "))])
                 (let ([next-op (first answers)])
                   [course (second answers)])
                 (cond
                   [((symbol=? next-op 'quit)
                      taking]
                   [((symbol=? next-op 'add)
                      (helper (cons course taking)))]
                   [((symbol=? next-op 'drop)
                      (helper (filter (lambda (curr) 
                                       (not (symbol=? curr course)))
                                       taking)))]
                 (helper empty))]])
               (let ([courses (select-courses)])
                 (display-result courses)))])))

Figure 1: Interactive Version
in different guises? If they’re different, why? If they’re the same, why don’t they appear that way—and, more importantly, given one (hopefully the simple version above), can we generate the other (hopefully the Web version)?
These are the questions we will tackle in the next few days.

2 Comparison and Translation

Web programs are large, ugly and unwieldy. Instead of manipulating the Web version, we’ll start with the Scheme program above and transform it into something that should resemble your Web solution (it certainly resembles one of mine). At each step, we’ll justify why we’re performing that transformation.

2.1 Lifting Definitions

Just as warm-up, we’ll move a few definitions outside the outer local. We’ll pick on three in particular: ALL-COURSES, display-result and select-courses. The first is really easy. The third is a bit tricky, but also fairly large. The second is tricky in the same way, but a lot smaller, so it’s easier to see what we need to do when we lift it. We refer to this process of moving definitions from an inner to an outer scope lifting. See figure 2.

You know to always exercise caution when lifting heavy objects. In this case, if we’d lifted select-courses or display-result relatively blindly, we’d have gotten into trouble, because three of the variables bound in their bodies—name, street and email—would no longer be bound! Therefore, we have to pass these along explicitly. In general, when we lift the function to a sufficiently rarified scope, we need to pass variables that used to be bound, but have now become free, along as arguments.

2.2 Extracting the Loop

Okay, that was easy. But the heart of select-courses is select-helper, and that’s still within its body. As a result, select-courses is still rather large. Also, while it seems rather atomic, it buries the difference between the first time we select courses—when we’re starting with some default course list (presumably empty)—and subsequent passes, when we really are the masters of our coursework destiny. This becomes clearer once we lift the helper function out: see figure 3.

2.3 Currying

Okay, so now it’s clear that the first time around, the registrar’s program gets to pick the default course selection (here, not surprisingly, empty), while on subsequent turns it’s obviously our choice. (Note how, when you lift a function outside a narrow scope, you may want to give it a more descriptive name, reminiscent of the scope from which it emerged.)
(define ALL-COURSES '(cs16 cs18 cs22 cs31 cs32))

(define (display-result name street email courses)
  (printf "Name: \nStreet: \nEmail: \nFinal Choice: \n  name street email courses))

(define (select-courses name street email)
  (local
    [(define (helper taking)
      (begin
        (printf "————–\nName: \nStreet: \nEmail: \n    name street email)
        (printf "Enrolled: \n    taking)
        (printf "Remaining: \n    (filter (lambda (course)
            (not (member course taking)))
    ALL-COURSES))
    (let ([answers (prompt-read-many (list "Next operation (add/drop/quit): "
                                               "Course name: "))])
      (let ([next-op (first answers)]
            [course (second answers)])
        (cond
          [(symbol=? next-op 'quit) taking]
          [(symbol=? next-op 'add)
           (helper (cons course taking))]
          [(symbol=? next-op 'drop)
           (helper (filter (lambda (curr)
                           (not (symbol=? curr course)))
                           taking))]
        (helper empty)))))
  (define (go)
    (let ([answers (prompt-read-many (list "Enter your name: 
                                           "Enter your street address: 
                                           "Enter your email address: "))])
      (let ([name (first answers)]
            [street (second answers)]
            [email (third answers)])
        (let ([courses (select-courses name street email)])
          (display-result name street email courses)))))

Figure 2: Preliminary Abstractions
;;; Eliding ALL-COURSES, display-result and go

(define (select-helper name street email taking)
  (begin
    (printf "————–\n")
    (printf "Name: \nStreet: \nEmail: \n" name street email)
    (printf "Enrolled: \n" taking)
    (printf "Remaining: \n" (filter (lambda (course) (not (member course taking))) ALL-COURSES))
    (let ([answers (prompt-read-many (list "Next operation (add/drop/quit): " "Course name: "))])
      (let ([next-op (first answers)]
        [course (second answers)])
        (cond
          [(symbol=? next-op 'quit) taking]
          [(symbol=? next-op 'add) (select-helper name street email (cons course taking))]
          [(symbol=? next-op 'drop) (select-helper name street email (filter (lambda (curr) (not (symbol=? curr course))) taking))])])
      (define (select-courses name street email)
        (select-helper name street email empty)))

Figure 3: Loop Extracted
(define (display-result name street email)
  (lambda (courses)
    (printf "Name: ~a
Street: ~a
Email: ~a
Final Choice: ~a"
      name street email courses))))

(define (select-helper name street email)
  (lambda (taking)
    (begin
      (printf "—————
"
      (printf "Name: ~a
Street: ~a
Email: ~a"
        name street email)
      (printf "Enrolled: ~a
" taking)
      (printf "Remaining: ~a
" (filter (lambda (course)
                    (not (member course taking)))
                    ALL-COURSES))
    (let ((answers (prompt-read-many (list "Next operation (add/drop/quit): 
" "Course name: "))))
      (let ((next-op (first answers))
            (course (second answers))
          (cond
            [(symbol=? next-op 'quit) taking]
            [(symbol=? next-op 'add) ((select-helper name street email)
                                      (cons course taking))]
            [(symbol=? next-op 'drop) ((select-helper name street email)
                                        (filter (lambda (curr)
                                                  (not (symbol=? curr course)))
                                                  taking))]]))))

(define (select-courses name street email)
  ((select-helper name street email)
   empty))

(define (go)
  (let ((answers (prompt-read-many (list "Enter your name: 
" "Enter your street address: 
" "Enter your email address: "))))
    (let ((name (first answers))
          (street (second answers))
          (email (third answers)))
      (let ((courses (select-courses name street email)))
        ((display-result name street email) courses)))))

Figure 4: Curried Version
There is, however, something pretty unsatisfying about the type of `select-helper`. It’s not like we can drop any of the arguments—if we did, the function would signal an error because it wouldn’t have some of the data it needs—but it seems unfair to treat all four arguments on par. After all, the first three are only there because of lifting. Only the fourth is really a product of the function’s iteration.

We can express this idea directly in the code itself. We split every argument list into “actual” arguments (ones it had in the original program) and arguments introduced due to lifting. To separate them, we use a function abstraction. That is, we *curry* the function, which means we split the arguments up in the function definition, and apply them as necessary at the call site. (Fans of the theory of computation will recognize this as Kleene’s “s-m-n” theorem.)

This forced us to change calls at several loci, but the result is a lot more satisfying. We can distinguish between the arguments that are really a part of the computation, and those that we’re forced to simply carry along. Figure 4 demonstrates this.

### 2.4 Splitting

The function `select-helper` is pretty large, and it really does two logically disparate tasks: printing the current status to request action, and determining what action to take based on the user’s choices. It’s natural to split this into two distinct functions, as figure 5 shows, where `process-response` handles the user’s input.

Naturally, we will want to curry `process-response` also. This affects both caller and callee. Indeed, we might want to curry `process-response` twice: once for the initial three arguments (which were curried in `select-helper` already), and once more for *taking*, which is the principal parameter to `select-helper`, but now needs to be passed as an argument since we’ve split the scope. We could do this, but it isn’t strictly necessary; using just one level of nesting functions is still enough to demonstrate arguments essential to the function’s behavior and those ensuing from lifting and the splitting of scope. See figure 6 for the simpler version.

### 2.5 A Few Last Steps

The same argument we made about needing to split `select-helper` could be leveled against `go` also. We may as well apply the same steps, breaking it into three logical parts: reading personal information, getting course selections, and printing the final result. This last step is really pretty simple for our application, but we’ll just go ahead and define the abstraction anyway, just to be anal about it, as in figure 7.

Now, whenever we get some kind of user response, typically by invoking `prompt-read-many` but, at one higher level of abstraction, also from invoking `select-courses`, we immediately use a function to process the responses.
(define (select-helper name street email)
  (lambda (taking)
    (begin
      (printf "————–˜n"
      (printf "Name: ˜aˆnStreet: ˜aˆnEmail: ˜aˆn" name street email)
      (printf "Enrolled: ˜aˆn" taking)
      (printf "Remaining: ˜aˆn" (filter (lambda (course)
        (not (member course taking)))
      ALL-COURSES))
      (let ([answers (prompt-read-many (list "Next operation (add/drop/quit): "
            "Course name: ")]))
        (process-response name street email taking answers))))

(define (process-response name street email taking answers)
  (let ([next-op (first answers)]
         [course (second answers)])
    (cond
      [(symbol=? next-op 'quit)
        taking]
      [(symbol=? next-op 'add)
        ((select-helper name street email)
          (cons course taking))]
      [(symbol=? next-op 'drop)
        ((select-helper name street email)
          (filter (lambda (curr)
            (not (symbol=? curr course)))
            taking))])

Figure 5: Split Version

8
(define (select-helper name street email)
  (lambda (taking)
    (begin
      (printf "——————–n")
      (printf "Name: \" name \n"
        name street email)
      (printf "Enrolled: \" taking)
      (printf "Remaining: \" Remaining (filter (lambda (course) (not (member course taking)))
        ALL-COURSES))
      (let ([answers (prompt-read-many (list "Next operation (add/drop/quit): 
        "Course name: ")])
        (process-response name street email taking)
        answers)))))

(define (process-response name street email taking)
  (lambda (answers)
    (let ([next-op (first answers)]
      [course (second answers)])
      (cond
        [(symbol=? next-op 'quit)
          taking]
        [(symbol=? next-op 'add)
          ((select-helper name street email)
            (cons course taking))]
        [(symbol=? next-op 'drop)
          ((select-helper name street email)
            (filter (lambda (curr) (not (symbol=? curr course)))
              taking))])))

Figure 6: Split and Curried Further
3 Perspective

It might seem like we’ve gone to a lot of trouble here. We claim that each step will somehow make the program simpler or clearer, but it’s not clear that has happened at all. Indeed, it hasn’t, really. The final version of the program, while identical in behavior, is a heck of a lot more complex with its arguments and currying than the original program was.

So why did we do this? Because that’s what the Web demands. Indeed, what we’ve just done is essentially transform our simple interactive program into its Web equivalent. Let’s examine this piece-by-piece.

First, we have the library functions, in this case filter and prompt-read-many. Neither one is particularly difficult to design. It’ll emerge that prompt-read-many is actually quite interesting, but we’ll pass over that for now. filter certainly isn’t interesting. So we can just assume these library functions are always present.

This brings us to the first non-library function, go. This corresponds to the first script, which gets the personal information. It then invokes a (conceptually) second script to handle the rest of the computation. This is shown in figure 8.

The second script, shown in figure 9, does the bulk of the work. It invokes select-helper to print the current enrollment status (which necessitates adding one more parameter, which is initially empty). Providing an answer to this form invokes process-response, which handles the options. That is, process-response is the front-end of the second script, the first thing run on script invocation. In contrast, select-helper is simply the display engine, which generates the next HTML page.

This phase is done when the user doesn’t choose an option that runs process-response. At that point, the program is ready to conclude the transaction, so it invokes conclude. This leads to the very slim (conceptual) third script—which you may have in-lined into your second—which simply summarizes the courses, shown in figure 10.

There are two key lessons to learn from this exercise.

1. First, each script seems to be conceptually isolated, invoking other scripts to do what might correspond to a function call in a regular interactive program. We’ll return to this point in much more detail in the coming days.

2. Second, when the scripts invoke one another (or themselves), they need to carry around a good deal of data. This should sound a bit familiar—it’s a bit like a turtle... Indeed, the curried structure of the code clarifies exactly what’s going on. Web programming forces us to lift functions to the top-level because we can’t resume a computation after we’ve halted it, which the CGI specification demands. In the process, we have to use hidden fields or other machinery to convey all this information.

Indeed, we can make this relationship between “lifted” and “new” data very explicit:
(define (conclude name street email)
  (lambda (courses)
    ((display-result name street email)
     courses)))

(define (select-and-conclude answers)
  (let ([name (first answers)]
         [street (second answers)]
         [email (third answers)])
    (let ([courses (select-courses name street email)])
      ((conclude name street email)
       courses))))

(define (go)
  (let ([answers (prompt-read-many (list "Enter your name: 
                                        "Enter your street address: 
                                        "Enter your email address: 
                                        
                                        (select-and-conclude answers)))]))

Figure 7: Final Changes

(define (go)
  (let ([answers (prompt-read-many (list "Enter your name: 
                                        "Enter your street address: 
                                        "Enter your email address: 
                                        
                                        (select-and-conclude answers)))]))

Figure 8: First Script
(define (process-response name street email taking)
  (lambda (answers)
    (let ([next-op (first answers)]
       [course (second answers)])
      (cond
       [(symbol=? next-op 'quit) taking]
       [(symbol=? next-op 'add) ((select-helper name street email) (cons course taking))]
       [(symbol=? next-op 'drop) ((select-helper name street email) (filter (lambda (curr) (not (symbol=? curr course))) taking))]))))

(define (select-helper name street email)
  (lambda (taking)
    (begin
      (printf "————–\n"
      (printf "Name: \nStreet: \nEmail: \n" name street email)
      (printf "Enrolled: \n" taking)
      (printf "Remaining: \n" (filter (lambda (course) (not (member course taking))) ALL-COURSES))
      (let ([answers (prompt-read-many (list "Next operation (add/drop/quit):" "Course name: 
"))]
        ((process-response name street email taking) answers))))))

(define ALL-COURSES '({cs16 cs18 cs22 cs31 cs32}))

(define (select-courses name street email)
  ((select-helper name street email) empty))

(define (select-and-conclude answers)
  (let ([name (first answers)]
     [street (second answers)]
     [email (third answers)])
    (let ([courses (select-courses name street email)]
      ((conclude name street email) courses)))))

Figure 9: Second Script
<table>
<thead>
<tr>
<th>Web script</th>
<th>function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Web form fields</td>
<td>argument values</td>
</tr>
<tr>
<td>Form submission</td>
<td>function invocation</td>
</tr>
<tr>
<td>Hidden fields</td>
<td>environment</td>
</tr>
<tr>
<td>Web form with hidden fields</td>
<td>closure</td>
</tr>
<tr>
<td>Bookmark</td>
<td>function name</td>
</tr>
</tbody>
</table>

This last line illustrates exactly the problem we face when we try to bookmark a generated page: the bookmark itself saves only a URL, which is simply the name of the function. In other words, bookmarking implements *dynamic scoping*. Fortunately, each bookmark is launched in an “empty environment”, so we simply get script errors such as “submitted form didn’t contain necessary fields”. What we really want to save is the function with its lexical environment, namely the closure, so when we resume the computation, we have all the necessary information. One way to do this is to save the entire Web page, including its hidden fields, as the “bookmark”.
(define (conclude name street email)
  (lambda (courses)
    ((display-result name street email)
      courses))))

(define (display-result name street email)
  (lambda (courses)
    (printf "Name: ˜a˜nStreet: ˜a˜nEmail: ˜a˜nFinal Choice: ˜a˜n"
      name street email courses))))

Figure 10: Third Script