In our previous lecture, we studied how difficult it was to correctly implement even a program like

\[ (\text{display} (\text{+} (\text{web-read} \ \text{"First number: "})) \ (\text{web-read} \ \text{"Second number: "}))) \]

on the Web. We agreed that we were better off implementing a “Web compiler” that would correctly transform this to a form that was safe in the face of Web interactions. Today we will examine the essence of writing such a compiler.

We identified the key problem that such a compiler needs to face, namely that every time a Web program generates a form, it must halt execution. What this means is that any pending computation at that point is lost. That is, if we convert the string "First number" into a form and produce it as the output of the Web program, then we must halt the computation after issuing that output. As a result, the system cannot know that it needs to perform the rest of the computation.

What is this “rest of the computation”? In words, it is to take the result from the form, generate a form for the second number, add them, then display their result. We’re going to need to talk about these kinds of “rests of computation” a lot, and English (or any other natural language) is clearly an unwieldy medium for doing so. But we have a perfectly good language for expressing computations, so let’s use it! How would we express this in Scheme? We might write

\[ (\text{display} (\text{+} \square \ (\text{web-read} \ \text{"Second number: "}))) \]

What kind of thing is \( \square \)? Well, it’s what we’re using to stand in place of the result of the user’s form submission. “... to stand in place of a result”: that sounds mighty evocative of an identifier, and indeed it has the benefit that we already have identifiers in Scheme, so we don’t need to think about adding another concept to the language. The problem with treating it as an identifier, however, is that it’s a free (or “unbound”) identifier. How do we bind it?

We can bind it the way we bind all identifiers:

\[ (\text{lambda} \ \square \ (\text{display} (\text{+} \square \ (\text{web-read} \ \text{"Second number: "})))) \]

That is, we can rewrite the initial program as

\[ (\text{web-read} \ \text{"First number: "} \ (\text{lambda} \ \square \ (\text{display} (\text{+} \square \ (\text{web-read} \ \text{"Second number: "}))))) \]

Actually, this doesn’t quite make sense, because \text{web-read} takes only one argument, not two. Instead, we’ll pretend we have a procedure named \text{web-read/k}, which takes two arguments. The way \text{web-read/k} functions is as follows. It uses the first argument to generate a Web form. It stashes the second argument, the procedure or receiver, somewhere in the server. When the user submits a number, the server applies the receiver to the user-supplied number.

Using this new primitive, how do we rewrite the program above? We might be tempted to write something like this:

\[ (\text{display} (\text{+} \ (\text{web-read/k} \ \text{"First number: "} \ (\text{lambda} \ \square \ (\text{\ldots})))) \ (\text{\ldots})) \]
but that doesn’t work! Everything waiting to be run after the web-read/k that doesn’t get put in the receiver procedure never gets run because the program halts and forgets all the pending computations. This forces us to move all the remaining computation into the receiver, like so:

(web-read/k "First number: "
(lambda ()
  (display (+
    (web-read "Second number: ")))))

This binds the first number input by the user to [], and the computation continues normally . . .

. . . Or not. The problem is, the program still halts prematurely: it just does so on the second web-read instead of the first. The problem analysis is the same as for the first user input, so we can use the same solution. Having read the first number, we have to read the second number before we can add the two and display their output. Therefore:

(web-read/k "First number: "
(lambda (λ1)
  (web-read/k "Second number: "
    (lambda (λ2)
      (display (+ λ1 λ2)))))

Now, when the program finally generates the sum, it can safely halt without having registered any receivers, because there aren’t any computations left to perform.¹

Suppose we have a server that does not terminate. Then web-read/k can behave as follows. It can store the receiver procedures in a hash table, associating each with a uniquely-generated key. It the generates a form that includes the key in its “action” field (which identifies a receiver). When the server gets a request for that particular key, it extracts the receiver from the hash table and invokes it on the value that the user provides.

Suppose we can’t figure out how to modify the innards of a Web server to do all this. We still don’t need to waste the effort above. We just need to convert the program into a more traditional form, namely lift the nested procedures to the top-level:

(define (f2 λ2)
  (display (+ λ1 λ2)))

(define (f1 λ1)
  (web-read/k/action "<form action=f2>Second number:<</form>" f2))

(web-read/k/action "<form action=f1>First number:<</form>" f1)

Unfortunately, by sloppily lifting the procedures to the top-level, we’ve reintroduced the problem we saw in the previous lecture: [] is a free identifier!

Puzzle

How would you solve this free variable problem in the Web compiler?

Some Implications

Notice three implications of the transformation we have implicitly employed above.

1. We have had to make decisions about the order of evaluation. That is, we had to choose whether to evaluate the left or the right argument of addition first. Notice that we had to make the same decision when we added a store to the interpreter! Fortunately, we made the same decision here that we did there, namely that we will evaluate the left argument before the right argument.

2. The transformation we use is global, namely it (potentially) affects all the procedures in the program. In this, too, it is similar to the changes we had to make to support the store.

¹To be entirely pedantic, there is one thing left to do, which is to invoke an instruction such as exit. We leave extending the above sequence of transformation to handle this as an exercise to the reader.
3. This transformation *sequentializes* the program. Given a nested expression, it forces the programmer to choose which sub-expression to evaluate first (a consequence of the first point above); further, every subsequent operation lies in the receiver, which in turn picks the first expression to evaluate, pushing all other operations into its receiver; and so forth. The net result is a program that looks an awful lot like a traditional procedural program: we can easily imagine writing the addition server’s final version in the following form instead:

\[
\begin{align*}
\text{Box}_1 &= \text{web-read/k} \ "First number: " \\
\text{Box}_2 &= \text{web-read/k} \ "Second number: " \\
\text{display} &\ (\text{Box}_1 + \text{Box}_2)
\end{align*}
\]

In fact, we can easily derive the above procedural pseudo-code by mechanically transforming the final version of the addition server. This suggests that this series of transformations can be used to *compile* a program in a language like Scheme into one in a language like C . . .