

A Better Web API

sk and dbtucker

2002-10-11

In our previous lecture, we saw that the lack of a proper API (Application Programmer Interface) makes Web programs considerably more complex and error-prone. In this lecture, we will study a new API that enormously raises the level of abstraction at which programmers can write Web programs.

To explain this new API, we return to our canonical Web program:

```
(display (+ (web-read "First number: ")
           (web-read "Second number: ")))
```

Ideally, we would like an interface where this code—or something very close to it—is the entire Web program, with an API to handle the mismatch between this code and the Web's current API. Recall that the difficulty was that a Web program was forced to exit after executing each output.

1 A Digression on Generating HTML

We will need to generate HTML, the markup language of the Web, so let's first study a convenient way to describe it in a Scheme program.

Here's a sample HTML document:

```
<html>
  <title>
    <head>Enter a Number</head>
  </title>
  <body>
    <p>Enter a number here.</p>
  </body>
</html>
```

You may not have seen closing tags such as `</p>`, because many Web browsers don't enforce their need. But a proper HTML document should balance its tags.¹

Now let's think about *representing* this document in Scheme. Here's a pretty natural representation:

```
(list 'html
      (list 'title
            (list 'head
                  "Enter a number ")))
      (list 'body
            (list 'p
                  "Enter a number here. ")))
```

It's easy to see that this corresponds to the HTML version. Where did the closing tags go? The closing parentheses of the Scheme operations took their place, preserving the meaning but greatly reducing the quantity of text!

This is still a pretty unwieldy way of writing long lists. Fortunately, Scheme offers a shorter way of expressing them. We have seen this before when writing our parsers: In reality, Scheme permits programmers to quote any *s-expression*. Thus, we can write the above list more concisely as

¹Actually, HTML does not require this, but XML does. In this course, we'll pretend we're using a sanitized version of HTML, called XHTML, that obeys XML's conventions.

```
'(html
  (title
    (head "Enter a number"))
  (body
    (p "Enter a number here")))
```

At this point, it's simply a matter of invoking the right API procedure to convert this expression into the corresponding HTML, pointy-brackets and all. DrScheme provides such procedures,² but for what we're going to do today, we won't need even those.

Before we return to Web programming, let's think about this HTML representation a little more. There are two details we've overlooked in the presentation above.

The first is that HTML tags have *attributes*. For instance, a Web page designer can write

```
<body bgcolor="red">
  The reds are coming!
</body>
```

to dictate that the document's background should be red. We need a way to record this in our HTML representation. There are in fact many ways we can record this; the one we'll choose is the following. We put attributes nested in parentheses after the tag and before the tag's body. Thus, the example above becomes

```
'(body ([bgcolor "red"])
  "The reds are coming!")
```

(The use of brackets is entirely a matter of convention.) We can represent two attributes easily enough:

```
<body bgcolor="red"
  vlink="yellow">
  The reds are coming!
</body>
```

becomes

```
'(body ([bgcolor "red"]
  (vlink "yellow"))
  "The reds are coming!")
```

In the special case where a tag has no attributes, we can write either

```
'(body ()
  "The reds are coming!")
```

or, equivalently,

```
'(body
  "The reds are coming!")
```

to reduce the volume of typing.

The second detail is this. Remember that in the end, we're trying to represent a Web page. And the whole point of generating Web pages dynamically is that they rarely have the same content. Therefore, we sometimes want to be able to change some parts of a page. Indeed, the important detail is that we often want to be able to change *parts* of a page, leaving the scaffolding intact. How can we do that here?

Ideally, a Web page template is simply a Scheme function. That is, we'd like to write

```
(define (generate-page nth)
  '(html
    (title
      (head "Enter a number"))
    (body
      (p "Enter " nth " number here"))))
```

²Indeed, we use these libraries to maintain the cs173 Web pages, as well as many other Web sites.

followed by invocations such as

```
(generate-page "first")  
(generate-page "second")
```

Unfortunately, this doesn't have the desired effect at all. The name `nth` in the body is a quoted symbol, not the identifier `nth` (note carefully the typographic conventions!). In effect, we'd like to "escape" back into Scheme so that the `nth` in the body can instead refer to the identifier `nth`.

We can do this easily enough in Scheme. We'll present the technique in two stages. The first is to use `'`, pronounced *quasiquote* (or *backquote* by the verbally challenged), in place of `'` (*quote*). That is, the function becomes

```
(define (generate-page nth)  
  `(html  
    (title  
      (head "Enter a number"))  
    (body  
      (p "Enter " nth " number here"))))
```

This in itself, however, accomplishes nothing. A quasiquote is the same as a quotation ...
... Unless it contains a `,` (*unquote*). Thus, the function we really want is

```
(define (generate-page nth)  
  `(html  
    (title  
      (head "Enter a number"))  
    (body  
      (p "Enter " ,nth " number here"))))
```

The unquote tells quasiquote that the following expression (in this instance, the identifier `nth`) is really a bit of Scheme code, not literal data. Invoking this version of *generate-page* correctly generates the desired representation of the HTML terms.

Puzzle

Type in and experiment with the function above. Then try to create an HTML template for your own Web site. Finally, try to use `,@` (*unquote-splicing*) in place of unquote and see what happens!

2 Web Programming

With that in hand, we can now return to studying Web programming. Let's first transform our program so it actually uses HTML to create forms. The final output of the program is a Web page displaying the sum, so we can write this as follows:

```
(html (head (title "Sum"))  
      (body ([bgcolor "white"]  
            (p "The sum is "  
              ,(number→string (+ (request-number "first")  
                                 (request-number "second"))))))))
```

This is fairly simple as Web pages go: its body contains a single paragraph that prints the result of addition. Therefore, we can focus on *request-number*.

The procedure *request-number* is going to have to first generate a page that contains a form. Here's how we can write this form:

```
(define (build-request-page which-number)  
  `(html (head (title "Enter a Number to Add"))  
        (body  
          (form ([action "???" ] [method "post"])
```

```
(p
  "Enter the " ,which-number " number to add: "
  (input ([type "text" ] [name "number" ] [value " "]))
  (input ([type "submit" ] [name "enter" ] [value "Enter" ]))))))
```

(Be sure to notice the unquotation before *which-number* in the paragraph body.)

This looks reasonable, but a careful inspection should reveal that it's not! In particular, we've left out a very important bit of information, namely the URL of the form's action handler, namely the program that will receive and process the form's field values. What should we use here?

In general, when we're not sure of what value to write in the body of a procedure, we can always make it a parameter and hope for the best later on. That's what we'll do now. We'll make the action URL an argument to the function, and hope that we can supply the right one later. For reasons we'll see later, we're going to call that argument *k-url*, and supply it separately from the value for *which-number*:

```
; build-request-page : str → url → response
(define (build-request-page which-number)
  (lambda (k-url)
    `(html (head (title "Enter a Number to Add"))
      (body ([bgcolor "white" ])
        (form ([action ,k-url ] [method "post" ])
          "Enter the " ,which-number " number to add: "
          (input ([type "text" ] [name "number" ] [value " "]))
          (input ([type "submit" ] [name "enter" ] [value "Enter" ]))))))))))
```

What kind of value does *build-request-page* produce? We don't know for now, so we'll just call it a *response*, and figure it out later.

Having done all that, we're finally ready to tackle *request-number* itself. Clearly, the procedure has the following type and header:

```
; request-number : str → num
(define (request-number which-number)
  ...)
```

Furthermore, we know that in its body, it must invoke *build-request-page*:

```
... (build-request-page which-number) ...
```

What does this procedure invocation do? It returns a procedure that is expecting a URL. That URL will be the one that knows how to process the result of submitting the form, i.e., the one that will resume the suspended computation.

In our previous lecture, we wrote,

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 then 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.

Let's suppose we had such a mythical procedure, which we'll call **send/suspend** (because it sends a Web page, then suspends computation awaiting a response). This procedure generates the right kind of URL for resuming the computation. In particular, it consumes a procedure expecting a resumption URL, generates such a URL, and feeds it to the given procedure. That is, we need to write

```
... (send/suspend (build-request-page which-number)) ...
```

which is why we gave that somewhat odd type to *build-request-page*.

At this point, what's left is mainly bookkeeping. The value that **send/suspend** returns is of type *response*; we can now use a series of API invocations to extract the value of a field:

```
⋮
(extract-binding/single
```

```
'number
  (request-bindings (send/suspend (build-request-page which-number))))
:
:
```

extracts the value associated with the field named "number". From our Web programming experience, we know that these APIs return strings, but we're writing a procedure whose return type is a number. Therefore, we need to convert this into a number, for which we can employ *string→number*. Assembling the pieces gives us

```
(define (request-number which-number)
  (string→number
   (extract-binding/single
    'number
    (request-bindings (send/suspend (build-request-page which-number))))))
```

What is perhaps somewhat surprising about this development is that we are now looking at fully-executable code in the PLT Scheme Web server. In particular, the primitive unique to its API is the command **send/suspend**. The effect of **send/suspend** is to automatically generate the receivers of each Web interaction *without forcing the programmer to manually convert the program* to the special form we studied in the previous class! Let us see what these receivers look like. This is the receiver for the first invocation of **send/suspend**:

```
(lambda (□)
  (html (head (title "Sum "))
        (body ([bgcolor "white"])
              (p "The sum is "
                 ,(number→string (+ (string→number
                                       (extract-binding/single
                                        'number
                                        (request-bindings □)))
                                     (request-number "second"))))))))
```

The receiver for the second invocation depends on the value that the user provides as the first input; this shows that the receivers must be constructed on the fly, and cannot be determined entirely statically. Specifically, assuming the programmer provided 1729 as the first number, the second receiver is

```
(lambda (□)
  (html (head (title "Sum "))
        (body ([bgcolor "white"])
              (p "The sum is "
                 ,(number→string (+ 1729
                                       (string→number
                                        (extract-binding/single
                                         'number
                                         (request-bindings □))))))))))
```

Henceforth, we will call these receivers *continuations*, because they represent how the computation must continue. The question is how to generate the continuations automatically, without having to hand-convert the programs. The primitive **send/suspend** does this automatically; it is virtually unique to Scheme. Programming the Web is giving us an intuition into how continuations behave and how we can employ them; next we will study how to add them to a programming language.