Simulating a Web Server

HTTP protocol is like calling a function:

```
(define total 0)

(define (a)
  `(("Current value:" ,total)
    "Call a2 to add 2"
    "Call a3 to add 3"))

(define (a2)
  (set! total (+ total 2))
  (a))

(define (a3)
  (set! total (+ total 3))
  (a))
```

Simulating a Web Server

Stateless variant is functions with arguments:

```
(define (b)
  (do-b 0))

(define (do-b total)
  `(("Current value:" ,total)
    "Call b2 with " ,total " to add 2"
    "Call b3 with " ,total " to add 3"))

(define (b2 total)
  (do-b (+ total 2)))

(define (b3 total)
  (do-b (+ total 3)))
```

Simulating a Web Server

For complex data, use **`remember`** and **`lookup`** to make a simple key:

```
(define (c)
  (do-c "*"))

(define (do-c total)
  (local [(define key (remember total))]
    `(("Current value:" ,total)
      "Call c2 with " ,key " to append \"hello\""
      "Call c3 with " ,key " to append \"goodbye\"")))

(define (c2 key)
  (do-c (string-append (lookup key) " hello")))

(define (c3 key)
  (do-c (string-append (lookup key) " goodbye")))
```

Simulating a Web Server

```
(define table empty)

(define (remember v)
  (local [(define n (length table))]
    (begin
      (set! table (append table
                          (list v)))
      n)))

(define (lookup key)
  (list-ref table key))
```

Direct Interactive Programs

But normally we write code more like this:

```
(define (d)
  (do-d 0))

(define (do-d total)
  (begin
    (printf "Total is ~a\nAdd 2 next?\n" total)
    (do-d (+ total
             (if (read) 2 3)))))
```

Direct Interactive Programs

Or like this:

```
(define (f)
  (do-f 0))

(define (num-read prompt)
  (begin
    (printf "~a\n" prompt)
    (read)))

(define (do-f total)
  (do-f (+ (num-read
            (format "Total is ~a\nNext number...\n"
                    total))
           total)))
```

We'd like to have a **`web-read`**...

Interactive Web Programs

Can we make this work?

```
(define (g)
  (do-g 0))

(define (web-read prompt)
  `(,prompt
    "To continue ..."))

(define (do-g total)
  ... (web-read
       (format "Total is ~a\nNext number...\n"
               total))
  ...)
```

`web-read` should not be specific to **`g`**

Interactive Web Programs

```
(define (g)
  (do-g 0))

(define (web-read prompt)
  (local [(define key (remember ...))]
    `(,prompt
      "To continue, call resume with" ,key "and value")))

(define (resume key val)
  ...)

(define (do-g total)
  ... (web-read
       (format "Total is ~a\nNext number...\n"
               total))
  ...)
```

What should we remember?

Interactive Web Programs

```
(define (g)
  (do-g 0))

(define (web-read prompt total do-g)
  (local [(define key (remember (list do-g total)))]
    `(,prompt
      "To continue, call resume with" ,key "and value")))

(define (resume key val)
  (local [(define l (lookup key))]
    ((first l) ... (second l) ... val ...)))

(define (do-g total)
  (web-read
   (format "Total is ~a\nNext number...\n"
           total)
   total
   do-g))
```

How should `(second l)` and `val` be combined?

Interactive Web Programs

```
(define (g)
  (do-g 0))

(define (web-read/k prompt cont)
  (local [(define key (remember cont))]
    `(,prompt
      "To continue, call resume/k with" ,key "and value")))

(define (resume/k key val)
  (local [(define cont (lookup key))]
    (cont val)))

(define (do-g total)
  (web-read/k
   (format "Total is ~a\nNext number...\n"
           total)
   (lambda (val)
     (do-g (+ total val)))))
```

Interactive Web Programs

```
(define (h)
  (+ (num-read "first number")
     (num-read "second-number")))
```

⇒

```
(define (h)
  (web-read/k "First number"
              (lambda (v1)
                (web-read/k "Second number"
                            (lambda (v2)
                              (+ v1 v2))))))
```

But what if we want to use `h` twice (to add two pairs of numbers)?

Interactive Web Programs

```
(define (h)
  (+ (num-read "first number")
     (num-read "second-number")))
(define (i)
  ; works fine
  (begin (h) (h)))
```

```
(define (h)
  (web-read/k "First number"
              (lambda (v1)
                (web-read/k "Second number"
                            (lambda (v2)
                              (+ v1 v2))))))
(define (i)
  ; first call is useless
  (begin (h) (h)))
```

Continuation-Passing Style

If a function uses **`web-read/k`**, then to make it composable, it must always take a continuation

```
(define (h) (do-h identity))
(define (do-h cont)
  (web-read/k "First number"
              (lambda (v1)
                (web-read/k "Second number"
                            (lambda (v2)
                              (cont (+ v1 v2)))))))

(define (i) (do-i identity))
(define (do-i cont)
  (do-h (lambda (sum)
          ; web-pause/k is like web-read/k,
          ; but with no particular result
          (web-pause/k sum
                       (lambda ()
                         (do-h cont))))))
```

Continuation-Passing Style

```
(define (web-pause/k prompt cont)
  (local ((define key (remember cont)))
    `(,prompt
      "To continue, call p-resume/k with" ,key)))

(define (p-resume/k key)
  ((lookup key)))
```

Converting to Continuation-Passing Style

- Change every function that you define

```
; f : .... -> Y
```

  to add an argument **k**:

```
; f : .... (Y -> X) -> X
```

- Always call **k** instead of returning

- Never use a function's result directly

Direct Interactive Programs

Good:

```
(define (num-read prompt)
  (begin
    (printf "~a\n" prompt)
    (read)))
```

```
(define (h)
  (+ (num-read "First number")
     (num-read "Second number")))
```

Interactive Web Programs

Adequate:

```
(define (web-read/k prompt cont)
  (local [(define key (remember cont))]
    `(,prompt
      "To continue, call resume/k with" ,key "and value")))

(define (resume/k key val)
  (local [(define cont (lookup key))]
    (cont val)))
```

```
(define (do-h cont)
  (web-read/k "First"
              (lambda (v1)
                (web-read/k "Second"
                            (lambda (v2)
                              (cont (+ v1 v2)))))))

(define (h)
  (do-h identity))
```

Interactive Web Programs

Better:

```
(define (web-read prompt)
  ...
  (local [(define key (remember cont))]
    `(,prompt
      "To continue, call resume with" ,key "and value"))
  ...)

(define (resume key val)
  (local [(define cont (lookup key))]
    (cont val)))
```

```
(define (h)
  (+ (web-read "First")
     (web-read "Second")))
```

If we can implement this **`web-read`** somehow...

Implicit Continuations

With

```
(define (h)
  (+ (web-read "First")
     (web-read "Second")))
(h)
```

The implicit ***continuation*** of the first call to `web-read` is

```
(lambda (•)
  (+ •
     (web-read "Second")))
```

Implicit Continuations

With

```
(define (h)
  (+ (web-read "First")
     (web-read "Second")))
(h)
```

If the first `web-read` call produces `7`, then the continuation of the second `web-read` call is

```
(lambda (•)
  (+ 7
     •))
```

Implicit Continuations

With

```
(define (do-g total)
  (do-g (+ (web-read (format "Total: ~a" total))
           total)))
(do-g 0)
```

The continuation of the first call to **web-read** is

```
(lambda (•)
  (do-g (+ •
           0)))
```

Implicit Continuations

With

```
(define (do-g total)
  (do-g (+ (web-read (format "Total: ~a" total))
           total)))
(do-g 0)
```

If the first `web-read` call produces `7`, then the continuation of the second `web-read` call is

```
(lambda (•)
  (do-g (+ •
           7)))
```

Implicit Continuations

With

```
(define (do-g total)
  (do-g (+ (web-read (format "Total: ~a" total))
           total)))
(do-g 0)
```

If the second `web-read` call produces `8`, then the continuation of the second `web-read` call is

```
(lambda (•)
  (do-g (+ •
           15)))
```

etc.

Implementing web-read

We need an operation to convert the current *implicit* continuation into an *explicit* continuation:

```
(define (web-read prompt)
  ...
  (get-current-continuation)
  ...
  (local [(define key (remember cont))]
    `(,prompt
      "To continue, call resume with"
      ,key "and value"))
  ...)
```

This is not quite right, because the continuation of `(get-current-continuation)` is some context that wants a continuation, not the continuation of the `web-read` call...

Implementing web-read

`let/cc` locally binds a name to the "surrounding" continuation, and evaluates its body to produce a result:

```
(define (web-read prompt)
  (let/cc cont
    (local [(define key (remember cont))]
      `(,prompt
        "To continue, call resume with"
        ,key "and value"))))
```

Closer, but we need to escape instead of returning...

Implementing web-read

For now, use **`error`** to escape:

```
(define (web-read prompt)
  (let/cc cont
    (local [(define key (remember cont))]
      (error 'web-read
             "~a; to continue, call resume with ~a and value"
             prompt key))))
```

Reusing Direct-Style Web Pages

No more CPS, so re-using `h` for `i` is easy:

```
(define (web-pause prompt)
  (let/cc cont
    (local [(define key (remember cont))]
      (error 'web-pause
             "~a; to continue, call p-resume with ~a"
             prompt key))))

(define (p-resume key)
  (local [(define cont (lookup key))]
    (cont (void))))
```

```
(define (i)
  (web-pause (h))
  (h))
```

Reusing Direct-Style Web Pages

No CPS also means that we can use functions like **`map`**:

```
(define (web-read-each prompts)
  (map web-read prompts))

(define (m)
  (apply format "my ~a saw a ~a rock"
         (web-read-each '("noun" "adjective"))))
```

Continuations in web-read-all

```
(define (web-read-each prompts)
  (map web-read prompts))

(define (m)
  (apply format
         "my ~a saw a ~a rock"
         (web-read-each '("noun" "adjective"))))
```

```
(define (map f l)
  (cond
   [(empty? l) empty]
   [else (cons (f (first l))
               (map f
                    (rest l)))]))
```

Evaluation:

```
(m)
```

⇒

```
(apply format "my ~a saw a ~a rock"
       (web-read-each '("noun" "adjective")))
```

Continuations in web-read-all

```
(define (web-read-each prompts)
  (map web-read prompts))

(define (m)
  (apply format
         "my ~a saw a ~a rock"
         (web-read-each '("noun" "adjective"))))
```

```
(define (map f l)
  (cond
   [(empty? l) empty]
   [else (cons (f (first l))
               (map f
                    (rest l)))]))
```

Evaluation:

```
(apply format "my ~a saw a ~a rock"
       (web-read-each '("noun" "adjective")))
```

⇒

```
(apply format "my ~a saw a ~a rock"
       (map web-read '("noun" "adjective")))
```

Continuations in web-read-all

```
(define (web-read-each prompts)
  (map web-read prompts))

(define (m)
  (apply format
         "my ~a saw a ~a rock"
         (web-read-each '("noun" "adjective"))))
```

```
(define (map f l)
  (cond
   [(empty? l) empty]
   [else (cons (f (first l))
               (map f
                    (rest l)))]))
```

Evaluation:

```
(apply format "my ~a saw a ~a rock"
       (map web-read '("noun" "adjective")))

⇒  (apply format "my ~a saw a ~a rock"
          (cond
           [(empty? '("noun" "adjective")) empty]
           [else (cons (web-read (first '("noun" "adjective")))
                       (map web-read
                            (rest '("noun" "adjective"))))]))
```

Continuations in web-read-all

```
(define (web-read-each prompts)
  (map web-read prompts))

(define (m)
  (apply format
         "my ~a saw a ~a rock"
         (web-read-each '("noun" "adjective"))))
```

```
(define (map f l)
  (cond
   [(empty? l) empty]
   [else (cons (f (first l))
               (map f
                    (rest l)))]))
```

Evaluation:

```
(apply format "my ~a saw a ~a rock"
       (cond
        [(empty? '("noun" "adjective")) empty]
        [else (cons (web-read (first '("noun" "adjective")))
                    (map web-read
                         (rest '("noun" "adjective"))))]))
```

```
⇒  (apply format "my ~a saw a ~a rock"
          (cons (web-read (first '("noun" "adjective")))
                (map web-read
                     (rest '("noun" "adjective")))))
```

Continuations in web-read-all

```
(define (web-read-each prompts)
  (map web-read prompts))

(define (m)
  (apply format
         "my ~a saw a ~a rock"
         (web-read-each '("noun" "adjective"))))
```

```
(define (map f l)
  (cond
   [(empty? l) empty]
   [else (cons (f (first l))
               (map f
                    (rest l)))]))
```

Evaluation:

```
(apply format "my ~a saw a ~a rock"
       (cons (web-read (first '("noun" "adjective")))
             (map web-read
                  (rest '("noun" "adjective")))))
```

⇒

```
(apply format "my ~a saw a ~a rock"
       (cons (let/cc cont
               (local [(define key (remember cont))]
                 (error ...)))
             (map web-read
                  (rest '("noun" "adjective"))))))
```

Continuations in web-read-all

```
(define (web-read-each prompts)
  (map web-read prompts))

(define (m)
  (apply format
         "my ~a saw a ~a rock"
         (web-read-each '("noun" "adjective"))))
```

```
(define (map f l)
  (cond
   [(empty? l) empty]
   [else (cons (f (first l))
               (map f
                    (rest l)))]))
```

Evaluation:

```
(apply format "my ~a saw a ~a rock"
       (cons (let/cc cont
               (local [(define key (remember cont))]
                 (error ...)))
             (map web-read
                  (rest '("noun" "adjective")))))
```

⇒

```
(apply format "my ~a saw a ~a rock"
       (cons (local [(define key (remember
                                  (lambda (•)
                                    (apply format "my ~a saw a ~a rock"
                                           (cons •
                                                 (map web-read
                                                      (rest '("noun" "adjective"))))))))])
                 (error ...))
             (map web-read
                  (rest '("noun" "adjective")))))
```

Escaping

How **`error`** escapes (roughly):

```
(define top-level (let/cc k k))

(define (error ...)
  ; Write error message:
  ...
  ; Escape:
  (top-level top-level))
```

Applying a continuation throws away the current continuation!

So **`let/cc`** actually creates something like

```
(lambda↑ (•) ... • ...)
```

Direct-Style Interactive Web Pages

```
; mutated, for a kind of dynamic scope:
(define current-start-k #f)

; adjust `serve' for to set `current-start-k':
(define (serve)
  ...
  (return-page (let/cc k
                 (set! current-start-k k)
                 (dispatch (cadr m)))
               in out))

(define (web-read prompt)
  (let/cc k
    (current-start-k
     (web-read/k prompt (lambda (val)
                          (k val))))))
```

Continuations for Exceptions

```
; sum-items : list-of-num-and-sym -> num-or-false
; Returns the sum if all numbers, false otherwise
(define (sum-items l)
  (cond
   [(empty? l) 0]
   [else (if (symbol? (first l))
             false
             (if (number? (sum-items (rest l)))
                 (+ (first l) (sum-items (rest l)))
                 false))]))

; Better:
(define (sum-items l)
  (let/cc esc
    (local [(define (sum-items l)
              (cond
               [(empty? l) 0]
               [else (if (symbol? (first l))
                         (esc false)
                         (+ (first l) (sum-items (rest l))))]))]
      (sum-items l))))
```

Continuations for Coroutines

```
(define tasks empty)

(define (spawn! thunk)
  (set! tasks (append tasks (list thunk))))

(define (next!)
  (local [(define t (first tasks))]
    (set! tasks (rest tasks))
    (t)))

(define (swap)
  (let/cc k
    (begin (spawn! k) (next!))))

(define (loop label cnt)
  (begin (printf "~a ~a\n" label cnt)
         (swap)
         (loop label (add1 cnt))))

(spawn! (lambda () (loop "a" 0)))
(spawn! (lambda () (loop "b" 0)))
(next!)
```