Part I

First-Class Continuations
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:

```scheme
(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 \textit{continuation} of the first call to \texttt{web-read} is

(lambda (\_)
  (+ \_ \_
      (web-read "Second"))))
Implicit Continuations

With

\[
\text{(define } (h) \\
\quad (+ \text{ (web-read "First")} \\
\quad \text{ (web-read "Second")}) \text{))} \\
\text{(h)}
\]

If the first \texttt{web-read} call produces 7, then the continuation of the second \texttt{web-read} call is

\[
\text{(lambda (\cdot)} \\
\quad (+ 7 \\
\quad \cdot))
\]
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:

```scheme
(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 \texttt{h} for \texttt{i} is easy:

\begin{verbatim}
(define (web-pause prompt)
  (let/cc cont
    (local [(define key (remember cont))]
      (error 'web-pause
        "\textasciitilde 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))
\end{verbatim}
No CPS also means that we can use functions like `map`:

```scheme
(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"))))
```

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

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

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"))))))
(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")))))))))

Continuations in web-read-all
Escaping

How `error` escapes (roughly):

```scheme
(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

```scheme
(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!)
Part II
Implementing Continuations
KCFAE Grammar

\[\langle \text{KCFAE} \rangle \ := \ \langle \text{num} \rangle \]
\[\ | \ [+ \ \langle \text{KCFAE} \rangle \ \langle \text{KCFAE} \rangle \rangle \]
\[\ | \ [\ - \ \langle \text{KCFAE} \rangle \ \langle \text{KCFAE} \rangle \rangle \]
\[\ | \ \langle \text{id} \rangle \]
\[\ | \ \{\ \text{fun} \ \{\langle \text{id} \rangle \} \ \langle \text{KCFAE} \rangle \} \]
\[\ | \ \{\ \langle \text{KCFAE} \rangle \ \langle \text{KCFAE} \rangle \} \]
\[\ | \ \{\ \text{if0} \ \langle \text{KCFAE} \rangle \ \langle \text{KCFAE} \rangle \ \langle \text{KCFAE} \rangle \} \]
\[\ | \ \{\ \text{withcc} \ \langle \text{id} \rangle \ \langle \text{KCFAE} \rangle \} \]

\{\text{withcc} \ k \ [+ \ 1 \ \{k \ 2\}]\} \Rightarrow 2
\{\text{withcc} \ \text{done} \}
\{\{\text{withcc} \ \text{esc} \}
\{\text{done} \ [+ \ 1 \ \{\text{withcc} \ k \}
\{\text{esc} \ k\}]}\} \}
\} \Rightarrow 4
(define-type KCFAE-Value
    [numV (n number?)]
    [closureV (param symbol?)
        (body KCFAE?)
        (ds DefrdSub?)
    ]
    [contV (proc procedure?)])
Implementing withcc

; interp : KCFAE DefrdSub -> KCFAE-Value
(define (interp a-fae ds)
  (type-case KCFAE a-fae
    ...
    [withcc (id body-expr)
      (let/cc k
        (interp body-expr
          (aSub id
            (contV k)
            ds))))])

This will work, but it’s too meta-circular to tell us anything
Implementing Continuations

; interp : KCFAE DefrdSub (KCFAE-Value -> alpha) -> alpha
(define (interp a-fae ds k)
  ...
  ...)

39
Implementing Continuations

; interp : KCFAE DefrdSub (KCFAE-Value -> alpha) -> alpha
(define (interp a-fae ds k)
    ...
    [num (n) (k (numV n))]
    ...)

40-42
Implementing Continuations

; interp : KCFAE DefrdSub (KCFAE-Value -> alpha) -> alpha
(define (interp a-fae ds k)
  ...
  [add (l r)
   (interp l ds
    (lambda (v1)
      (interp r ds
       (lambda (v2)
        (k (num+ v1 v2))))))]
  ...
)
Implementing Continuations

; interp : KCFAE DefrdSub (KCFAE-Value -> alpha) -> alpha
(define (interp a-fae ds k)
  ...  
  [sub (l r)
   (interp l ds
    (lambda (v1)
     (interp r ds
      (lambda (v2)
       (k (num- v1 v2))))))]
  ...)

47
Implementing Continuations

; interp : KCFAE DefrdSub (KCFAE-Value -> alpha) -> alpha
(define (interp a-fae ds k)
  ...
  [id (name) (k (lookup name ds))]
  ...)
Implementing Continuations

; interp : KCFAE DefrdSub (KCFAE-Value -> alpha) -> alpha
(define (interp a-fae ds k)
  ...
  [fun (param body-expr)
     (k (closureV param body-expr ds))]
  ...)


Implementing Continuations

; interp : KCFAE DefrdSub (KCFAE-Value -> alpha) -> alpha
(define (interp a-fae ds k)
...
[app (fun-exp arg-exp)
  (interp fun-exp ds
    (lambda (fun-val)
      (interp arg-exp ds
        (lambda (arg-val)
          (type-case KCFAE fun-val
            [closureV (param body-exp ds)
              (interp body-exp
                (aSub param arg-val ds)
                k)]
            [contV (k)
              (k arg-val)]
            [else (error ...)]]))))])

...)

53-59
Implementing Continuations

; interp : KCFAE DefrdSub (KCFAE-Value -> alpha) -> alpha
(define (interp a-fae ds k)
  ...
  [if0 (test-expr then-expr else-expr)
   (interp test-expr ds
    (lambda (v)
      (if (numzero? v)
        (interp then-expr ds k)
        (interp else-expr ds k))))]
  ...
)
Implementing Continuations

; interp : KCFAE DefrdSub (KCFAE-Value -> alpha) -> alpha
(define (interp a-fae ds k)
  ...
  [withcc (id body-expr)
    (interp body-expr
      (aSub id
        (contV k)
        ds)
      k)]
  ...
)