Programming Language Engineering Master of Computer Science

Faculty of Science and Bio-Engineering Sciences
Vrije Universiteit Brussel

Section 2: Continuations

Theo D'Hondt
Software Languages Lab

"... continuation-passing style is a style of programming in which control is passed explicitly in the form of a continuation..."

Continuation Passing Style

```
(begin
  (define (factorial n)
    (if (> n 1)
      (* n (factorial (- n 1)))
      1))
  (factorial 10))
3628800
```

```
(begin
  (define (factorial n continue)
    (define (continuation p)
      (continue (* n p)))
    (if (> n 1)
      (factorial (- n 1) continuation)
      (continue 1)))
  (factorial 10 display))
3628800
```

Continuation Passing Style (cont'd)

```
(begin
  (define (fibonacci n)
    (if (> n 1)
      (+ (fibonacci (- n 1))
         (fibonacci (- n 2)))
      1))
  (fibonacci 15))
987
```

```
(begin
  (define (fibonacci n continue)
    (define (continuation p)
      (define (continuation q)
        (continue (+ p q)))
      (fibonacci (- n 2) continuation))
    (if (> n 1)
      (fibonacci (- n 1) continuation)
      (continue 1)))
  (fibonacci 15 display))
987
```

From Slip to cpSlip

```
(begin
  (define meta-level-eval eval)

  (define environment '())

;
; read-eval-print
;

  (define (loop output)
    (define rollback environment)

    (define (error message qualifier)
      (set! environment rollback)
      (display message)
      (loop qualifier))



    (display output)
    (newline)
    (display ">>>")
    (loop (evaluate (read))))

  (loop "Slip version 3"))
```

From Slip to cpSlip

```
(begin
  (define meta-level-eval eval)

  (define environment '())

;
; read-eval-print
;

  (define (loop output)
    (define rollback environment)

    (define (error message qualifier)
      (set! environment rollback)
      (display message)
      (loop qualifier))




    (display output)
    (newline)
    (display ">>>")
    (evaluate (read) loop))

  (loop "cpSlip version 0"))
```

cps

Evaluator:

```
(define (evaluate expression)
  (cond
    ((symbol? expression)
     (evaluate-variable expression))
    ((pair? expression)
     (let ((operator (car expression))
           (operands (cdr expression)))
       (apply
         (case operator
           ((begin)  evaluate-begin )
           ((define) evaluate-define)
           ((if)     evaluate-if    )
           ((lambda) evaluate-lambda)
           ((quote)  evaluate-quote )
           ((set!)   evaluate-set!  )
           ((while)  evaluate-while )
           (else     (evaluate-application operator))) operands)))
    (else
      expression)))
```

Evaluator:

```
(define (evaluate expression continue)
  (cond
    ((symbol? expression)
     (evaluate-variable expression continue))
    ((pair? expression)
     (let ((operator (car expression))
           (operands (cdr expression)))
       ((apply
          (case operator
            ((begin)  evaluate-begin )
            ((define) evaluate-define)
            ((if)     evaluate-if    )
            ((lambda) evaluate-lambda)
            ((quote)  evaluate-quote )
            ((set!)   evaluate-set!  )
            ((while)  evaluate-while )
            (else     (evaluate-application operator))) operands) continue)))
    (else
      (continue expression))))
```

cps

Evaluator:

```
(define (evaluate expression continue)
  (cond
    ((symbol? expression)
     (evaluate-variable expression continue))
    ((pair? expression)
     (let ((operator (car expression))
           (operands (cdr expression)))
       ((apply
          (case operator
            ((begin)  evaluate-begin )
            ((define) evaluate-define)
            ((if)     evaluate-if    )
            ((lambda) evaluate-lambda)
            ((quote)  evaluate-quote )
            ((set!)   evaluate-set!  )
            ((while)  evaluate-while )
            (else     (evaluate-application operator))) operands) continue)))
    (else
     (continue expression))))
```

cps

Currying

Evaluation functions

```
(define (evaluate-▲▼▲ ... operand ...)
  (lambda (continue)
    (define (continuation value)
       ... continue ...)
           ...
    (evaluate operand continuation)
           ... ))
```

```
(let ((operator (car expression))
      (operands (cdr expression)))
```

```
(define (evaluate expression continue)
     ...
  ((apply evaluate-▲▼▲ operands) continue)
     ... )
```

Evaluation functions

```
(define (evaluate-set! variable expression)
  (lambda (continue)
    (define (continuation value)
      (define binding (assoc variable environment))
      (set-cdr! binding value)
      (continue value))
    (evaluate expression continuation)))
```

```
(let ((operator (car expression))
      (operands (cdr expression)))
```

cps

```
(define (evaluate expression continue)
     ...
  ((apply evaluate-set! operands) continue)
     ... )
```

Evaluation: assignment

```
(define (evaluate-set! variable expression)
  (define binding (assoc variable environment))
  (if binding
    (let ((value (evaluate expression)))
      (set-cdr! binding value)
      value)
    (error "inaccessible variable: " variable)))
```

Evaluation: assignment

```scheme
(define (evaluate-set! variable expression)
  (lambda (continue)
    (define (continue-after-expression value)
      (define binding (assoc variable environment))
      (if binding
        (set-cdr! binding value)
        (error "inaccessible variable: " variable))
      (continue value))
    (evaluate expression continue-after-expression)))
```

cps

Evaluation: conditional

```
(define (evaluate-if predicate consequent . alternative)
  (define boolean (evaluate predicate))
  (if (eq? boolean #f)
    (if (null? alternative)
      '()
      (evaluate (car alternative)))
    (evaluate consequent)))
```

Evaluation: conditional

```
(define (evaluate-if predicate consequent . alternative)
  (lambda (continue)
    (define (continue-after-predicate boolean)
      (if (eq? boolean #f)
        (if (null? alternative)
          (continue '())
          (evaluate (car alternative) continue))
        (evaluate consequent continue)))
    (evaluate predicate continue-after-predicate)))
```

cps

Evaluation: begin

```
(define (evaluate-begin . expressions)
  (evaluate-sequence expressions))
```

```
(define (evaluate-sequence expressions)
  (define head (car expressions))
  (define tail (cdr expressions))
  (if (null? tail)
    (evaluate head)
    (begin
      (evaluate head)
      (evaluate-sequence tail))))
```

Evaluation: begin

```
(define (evaluate-begin . expressions)
  (lambda (continue)
    (evaluate-sequence expressions continue)))
```

```
(define (evaluate-sequence expressions continue)
  (define head (car expressions))
  (define tail (cdr expressions))
  (define (continue-with-sequence value)
    (evaluate-sequence tail continue))
  (if (null? tail)
    (evaluate head continue)
    (evaluate head continue-with-sequence)))
```

cps

Evaluation: lambda

```
(define (evaluate-lambda parameters . expressions)
  (make-procedure parameters expressions))
```

```
(define (make-procedure parameters expressions)
  (define lexical-environment environment)
  (lambda arguments
    (define dynamic-environment environment)
    (set! environment lexical-environment)
                     parameters arguments)
                     luate-sequence expressions)))
                     nt dynamic-environment)
```

```
(define (bind-variable variable value)
  (define binding (cons variable value))
  (set! environment (cons binding environment)))

(define (bind-parameters parameters arguments)
  (if (symbol? parameters)
    (bind-variable parameters arguments))
    (if (pair? parameters)
      (let
        ((variable (car parameters))
         (value    (car arguments )))
        (bind-variable variable value)
        (bind-parameters (cdr parameters) (cdr arguments)))))
```

Evaluation: lambda

```
(define (evaluate-lambda parameters . expressions)
  (lambda (continue)
    (continue (make-procedure parameters expressions))))
```

```
(define (make-procedure parameters expressions)
  (define lexical-environment environment)
  (lambda (arguments continue)
    (define dynamic-environment environment)
    (define (continue-after-sequence value)
      (set! environment dynamic-environment)
      (continue value))
    (set! environment lexical-environment)
    (bind-parameters parameters arguments)
    (evaluate-sequence expressions
                       continue-after-sequence)))
```

```
(define (bind-va
  (define binding
  (set! environme

(define (bind-pa
  (if (symbol? pa
    (bind-variab
    (if (pair? pa
      (let
        ((variab
         (value
        (bind-va
        (bind-pa
```

cps

Evaluation: define

```
(define (evaluate-define pattern . expressions)
  (if (symbol? pattern)
    (let* ((value (evaluate (car expressions)))
           (binding (cons pattern value)))
      (set! environment (cons binding environment))
      value)
    (let* ((binding (cons (car pattern) '())))
      (set! environment (cons binding environment))
      (let ((procedure
             (make-procedure (cdr pattern) expressions)))
        (set-cdr! binding procedure)
        procedure))))
```

Evaluation: define

```
(define (evaluate-define pattern . expressions)
  (lambda (continue)
    (define (continue-after-expression value)
      (define binding (cons pattern value))
      (set! environment (cons binding environment))
      (continue value))
    (if (symbol? pattern)
      (evaluate (car expressions)
                continue-after-expression))
      (let* ((binding (cons (car pattern) '())))
        (set! environment (cons binding environment))
        (let ((procedure
                (make-procedure (cdr pattern) expressions)))
          (set-cdr! binding procedure)
          (continue procedure))))))
```

cps

Evaluation: application

```
(define (evaluate-application operator)
  (lambda operands
    (apply (evaluate operator) (map evaluate operands))))
```

Evaluation: application

```
(define (evaluate-application operator)
  (lambda operands
    (lambda (continue)
      (define (continue-after-operator procedure)
        (define (evaluate-operands operands arguments)
          (define (continue-with-operands value)
            (evaluate-operands (cdr operands)
                               (cons value arguments)))
          (if (null? operands)
            (procedure (reverse arguments) continue)
            (evaluate (car operands)
                      continue-with-operands)))
        (evaluate-operands operands '()))
      (evaluate operator continue-after-operator))))
```

cps

Evaluation: application

```
(define (wrap-native-procedure native-procedure)
  (lambda (arguments continue)
    (define native-value
            (apply native-procedure arguments))
    (continue native-value)))
```

cps

cpSlip: Lisp in 120 lines

```
    (display output)
    (newline)
    (display ">>>")
    (evaluate (read) loop))

  (loop "cpSlip version 0"))
cpSlip version 0
>>>(begin
     (define (factorial n continue)
       (define (continuation p)
         (continue (* n p)))
       (if (> n 1)
         (factorial (- n 1) continuation)
         (continue 1)))
     (factorial 10 display))
3628800<unspecified>
>>>
```

Continuation Passing Style (cont'd)

```
(begin
  (define (fibonacci p q r)
    (if (> p 1)
      (fibonacci (- p 1) r (+ q r))
      r))
  (fibonacci 15 1 1))
987
```

```
(begin
  (define (fibonacci p continue)
    (define (continuation q r)
      (continue r (+ q r)))
    (if (> p 1)
      (fibonacci (- p 1) continuation)
      (continue 1 1)))
  (fibonacci 15 (lambda (q r)
                  (display r))))
987
```

Continuation Passing Style (cont'd)

```
(begin
  (define (fibonacci p q r)
    (if (> p 1)
      (fibonacci (- p 1) r (+ q r))
      r))
  (fibonacci 15 1 1))
987
```

```
(begin
  (define (fibonacci p continue)
    (define (continuation q r)
      (continue r (+ q r)))
    (if (> p 1)
      (fibonacci (- p 1) continuation)
      (continue 1 1)))
  (fibonacci 15 (lambda (q r)
                  (display r))))
987
```

Factoring in the environment

```
(begin
  (define meta-level-eval eval)

;
; read-eval-print
;

  (define (loop output environment)
    (define rollback environment)

    (define (error message qualifier)
      (display message)
      (loop qualifier rollback))






    (display output)
    (newline)
    (display ">>>")
    (evaluate (read) loop environment))

  (loop "cpSlip version 1" '()))
```

Factoring in the environment (cont'd)

```
(begin
  (define meta-level-eval eval)

;
; read-eval-print
;

  (define (loop output environment)
    (define rollback environment)

    (define (error message qualifier)
      (display message)
      (loop qualifier rollback))




    (display output)
    (newline)
    (display ">>>")
    (evaluate (read) loop environment))

  (loop "cpSlip version 1" '()))
```

Factoring in the environment (cont'd)

```
(begin
  (define meta-level-eval eval)

;
; read-eval-print
;

  (define (loop output environment)
    (define rollback environment)

    (define (error message qualifier)
      (display message)
      (loop qualifier rollback))



    (display output)
    (newline)
    (display ">>>")
    (evaluate (read) loop environment))

  (loop "cpSlip version 1" '()))
```

Evaluation functions

```
(define (evaluate-▲▼▲ ... operand ...)
  (lambda (continue environment)
    (define (continuation value environment)
       ... continue ...)
              ...
    (evaluate operand continuation environment)
              ... ))
```

```
(let ((operator (car expression))
      (operands (cdr expression)))
```

```
(define (evaluate expression continue environment)
     ...
  ((apply evaluate-▲▼▲ expression) continue environment)
     ... )
```

Evaluation functions

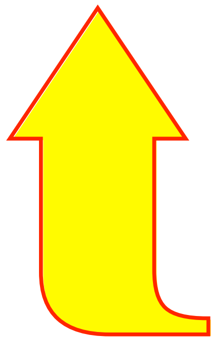
```
(define (evaluate-▲▼▲ ... operand ...)
  (lambda (continue environment)
    (define (continuation value environment)
       ... continue ...)
              ...
    (evaluate operand continuation environment)
              ... ))
```

```
(let ((operator (car expression))
      (operands (cdr expression)))
```

```
(define (evaluate expression continue environment)
     ...
  ((apply evaluate-▲▼▲ expression) continue environment)
     ... )
```

Evaluation functions

```
(define (evaluate-▲▼▲ ... operand ...)
  (lambda (continue environment)
    (define (continuation value environment)
      ... continue ...)
           ...
    (evaluate operand continuation environment)
           ... ))
```



```
(let ((operator (car expression))
      (operands (cdr expression)))
```

```
(define (evaluate expression continue environment)
     ...
  ((apply evaluate-▲▼▲ expression) continue environment)
     ... )
```

Evaluation functions

```
(define (evaluate-▲▼▲ ... operand ...)
  (lambda (continue environment)
    (define (continuation value environment)
       ... continue ...)
              ...
    (evaluate operand continuation environment)
              ... ))
```

```
(let ((operator (car expression))
      (operands (cdr expression)))
```

```
(define (evaluate expression continue environment)
     ...
  ((apply evaluate-▲▼▲ expression) continue environment)
     ... )
```

Evaluator:

```
(define (evaluate expression continue)
  (cond
    ((symbol? expression)
     (evaluate-variable expression continue))
    ((pair? expression)
     (let ((operator (car expression))
           (operands (cdr expression)))
       ((apply
          (case operator
            ((begin)  evaluate-begin )
            ((define) evaluate-define)
            ((if)     evaluate-if    )
            ((lambda) evaluate-lambda)
            ((quote)  evaluate-quote )
            ((set!)   evaluate-set!  )
            ((while)  evaluate-while )
            (else     (evaluate-application operator))) operands) continue)))
    (else
      (continue expression))))
```

cps

Evaluator:

```
(define (evaluate expression continue environment)
  (cond
    ((symbol? expression)
     (evaluate-variable expression continue environment))
    ((pair? expression)
     (let ((operator (car expression))
           (operands (cdr expression)))
       ((apply
          (case operator
            ((begin)  evaluate-begin )
            ((define) evaluate-define)
            ((if)     evaluate-if    )
            ((lambda) evaluate-lambda)
            ((quote)  evaluate-quote )
            ((set!)   evaluate-set!  )
            ((while)  evaluate-while )
            (else     (evaluate-application operator))) operands) continue
                                                                 environment)))
    (else
      (continue expression environment))))
```

cps*

Evaluator:

```
(define (evaluate expression continue environment)
  (cond
    ((symbol? expression)
     (evaluate-variable expression continue environment))
    ((pair? expression)
     (let ((operator (car expression))
           (operands (cdr expression)))
       ((apply
          (case operator
            ((begin)  evaluate-begin )
            ((define) evaluate-define)
            ((if)     evaluate-if    )
            ((lambda) evaluate-lambda)
            ((quote)  evaluate-quote )
            ((set!)   evaluate-set!  )
            ((while)  evaluate-while )
            (else     (evaluate-application operator))) operands) continue
                                                                   environment)))
    (else
      (continue expression environment))))
```

cps*

Evaluator:

```
(define (evaluate expression continue environment)
  (cond
    ((symbol? expression)
     (evaluate-variable expression continue environment))
    ((pair? expression)
     (let ((operator (car expression))
           (operands (cdr expression)))
       ((apply
          (case operator
            ((begin)  evaluate-begin )
            ((define) evaluate-define)
            ((if)     evaluate-if    )
            ((lambda) evaluate-lambda)
            ((quote)  evaluate-quote )
            ((set!)   evaluate-set!  )
            ((while)  evaluate-while )
            (else     (evaluate-application operator))) operands) continue
                                                                  environment)))
    (else
      (continue expression environment))))
```

cps*

Evaluation: assignment

```
(define (evaluate-set! variable expression)
  (lambda (continue)
    (define (continue-after-expression value)
      (define binding (assoc variable environment))
      (if binding
        (set-cdr! binding value)
        (error "inaccessible variable: " variable))
      (continue value))
    (evaluate expression continue-after-expression)))
```

cps

Evaluation: assignment

```
(define (evaluate-set! variable expression)
  (lambda (continue environment)
    (define (continue-after-expression value
                                       environment-after-expression)
      (define binding (assoc variable
                                       environment-after-expression))
      (if binding
        (set-cdr! binding value)
        (error "inaccessible variable: " variable))
      (continue value environment-after-expression))
    (evaluate expression continue-after-expression
                                                       environment)))
```

cps*

Evaluation: conditional

```
(define (evaluate-if predicate consequent . alternative)
  (lambda (continue)
    (define (continue-after-predicate boolean)
      (if (eq? boolean #f)
        (if (null? alternative)
          (continue '())
          (evaluate (car alternative) continue))
        (evaluate consequent continue)))
    (evaluate predicate continue-after-predicate)))
```

cps

Evaluation: conditional

```
(define (evaluate-if predicate consequent . alternative)
  (lambda (continue environment)
    (define (continue-after-predicate boolean
                                      environment-after-predicate)
      (if (eq? boolean #f)
        (if (null? alternative)
          (continue '() environment-after-predicate)
          (evaluate (car alternative) continue
                                     environment-after-predicate))
        (evaluate consequent continue
                                    environment-after-predicate)))
    (evaluate predicate continue-after-predicate
                                                environment)))
```

cps*

Evaluation: begin

```
(define (evaluate-begin . expressions)
  (lambda (continue)
    (evaluate-sequence expressions continue)))
```

```
(define (evaluate-sequence expressions continue)
  (define head (car expressions))
  (define tail (cdr expressions))
  (define (continue-with-sequence value)
    (evaluate-sequence tail continue))
  (if (null? tail)
    (evaluate head continue)
    (evaluate head continue-with-sequence)))
```

cps

Evaluation: begin

```
(define (evaluate-begin . expressions)
  (lambda (continue environment)
    (evaluate-sequence expressions
                                  continue environment)))
```

```
(define (evaluate-sequence expressions
                                  continue environment)
  (define head (car expressions))
  (define tail (cdr expressions))
  (define (continue-with-sequence value
                                           environment)
    (evaluate-sequence tail continue environment))
  (if (null? tail)
    (evaluate head continue environment)
    (evaluate head continue-with-sequence
                                         environment)))
```

cps*

Evaluation: lambda

```
(define (evaluate-lambda parameters . expressions)
  (lambda (continue)
    (continue (make-procedure parameters expressions))))
```

```
    (define (bind-va
      (define binding
      (set! environm

    (define (bind-pa
      (if (symbol? p
        (bind-variab
        (if (pair? p
          (let
            ((variab
             (value
            (bind-va
            (bind-pa
```

cps

```
(define (make-procedure parameters expressions)
  (define lexical-environment environment)
  (lambda (arguments continue)
    (define dynamic-environment environment)
    (define (continue-after-sequence value)
      (set! environment dynamic-environment)
      (continue value))
    (set! environment lexical-environment)
    (bind-parameters parameters arguments)
    (evaluate-sequence expressions
                       continue-after-sequence)))
```

Evaluation: lambda

```
(define (evaluate-lambda parameters . expressions)
  (lambda (continue environment)
    (continue (make-procedure parameters
                              expressions environment)
              environment)))
```

```
(define (make-procedure parameters
                        expressions environment)
  (lambda (arguments continue dynamic-environment)
    (define (continue-after-sequence value
                                     environment-after-sequence)
      (continue value dynamic-environment))
    (define lexical-environment
         (bind-parameters parameters arguments
                                     environment))
    (evaluate-sequence expressions
             continue-after-sequence lexical-environment)))
```

cps*

Evaluation: lambda

```
(define (bind-parameters parameters arguments environment)
  (if (symbol? parameters)
    (bind-variable parameters arguments environment)
    (if (pair? parameters)
      (let*
        ((variable (car parameters))
         (value (car arguments ))
         (environment (bind-variable
                                   variable value environment)))
        (bind-parameters
           (cdr parameters) (cdr arguments) environment))
      environment)))
```

```
(define (bind-variable variable value)
  (define binding (cons variable value))
  (set! environment
        (cons binding environment)))
```

```
(evaluate-sequence expressions
continue-after-sequence lexical-environment)))
```

cps*

Evaluation: define

```
(define (evaluate-define pattern . expressions)
  (lambda (continue)
    (define (continue-after-expression value)
      (define binding (cons pattern value))
      (set! environment (cons binding environment))
      (continue value))
    (if (symbol? pattern)
      (evaluate (car expressions)
                continue-after-expression))
      (let* ((binding (cons (car pattern) '())))
        (set! environment (cons binding environment))
        (let ((procedure
                (make-procedure (cdr pattern) expressions)))
          (set-cdr! binding procedure)
          (continue procedure)))))
```

cps

Evaluation: define

```
(define (evaluate-define pattern . expressions)
  (lambda (continue environment)
    (define (continue-after-expression
                       value environment-after-expression)
      (define binding (cons pattern value))
      (continue value
             (cons binding environment-after-expression)))
    (if (symbol? pattern)
      (evaluate (car expressions)
                   continue-after-expression environment))
      (let* ((binding (cons (car pattern) '()))
             (environment (cons binding environment))
             (procedure (make-procedure (cdr pattern)
                                   expressions environment)))
        (set-cdr! binding procedure)
        (continue procedure environment))))
```

cps*

Evaluation: application

```
(define (evaluate-application operator)
  (lambda operands
    (lambda (continue)
      (define (continue-after-operator procedure)
        (define (evaluate-operands operands arguments)
          (define (continue-with-operands value)
            (evaluate-operands (cdr operands)
                               (cons value arguments)))
          (if (null? operands)
            (procedure (reverse arguments) continue)
            (evaluate (car operands)
                      continue-with-operands)))
        (evaluate-operands operands '()))
      (evaluate operator continue-after-operator))))
```

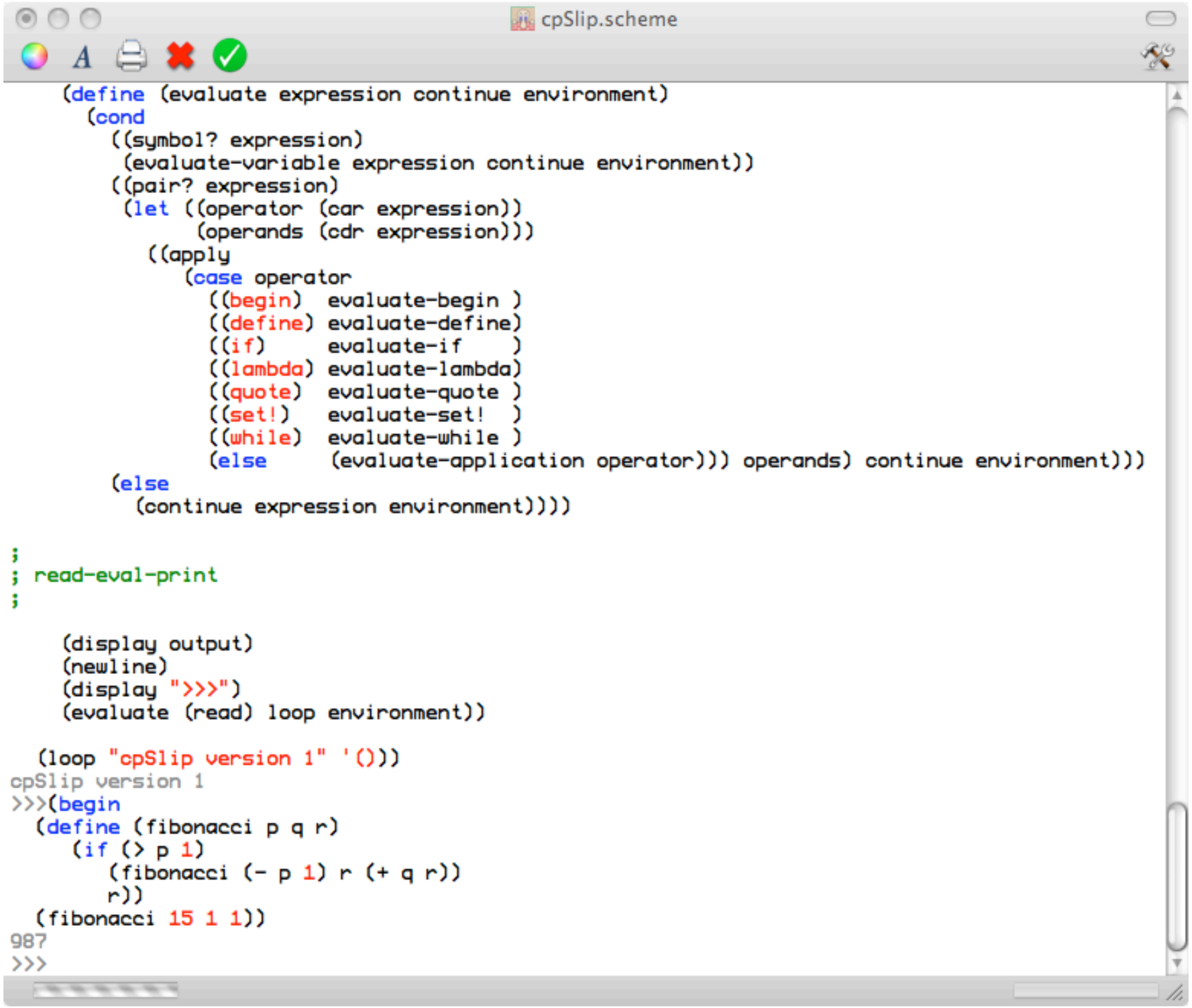
cps

Evaluation: application

```
(define (evaluate-application operator)
  (lambda operands
    (lambda (continue environment)
      (define (continue-after-operator procedure
                                       environment-after-operator)
        (define (evaluate-operands operands
                                        arguments environment)
          (define (continue-with-operands
                               value environment-with-operands)
            (evaluate-operands (cdr operands)
                               (cons value arguments)
                               environment-with-operands))
          (if (null? operands)
            (procedure (reverse arguments)
                                         continue environment)
            (evaluate (car operands)
                  continue-with-operands environment)))
        (evaluate-operands operands '()
                                                             )
      (evaluate operator c
                                                             )))
```

```
  (define (wrap-native-procedure native-procedure)
    (lambda (arguments continue)
      (define native-value
              (apply native-procedure arguments))
      (continue native-value)))
```

cpSlip: Lisp in 120(!) lines



```scheme
    (define (evaluate expression continue environment)
      (cond
        ((symbol? expression)
         (evaluate-variable expression continue environment))
        ((pair? expression)
         (let ((operator (car expression))
               (operands (cdr expression)))
           ((apply
              (case operator
                ((begin)  evaluate-begin )
                ((define) evaluate-define)
                ((if)     evaluate-if    )
                ((lambda) evaluate-lambda)
                ((quote)  evaluate-quote )
                ((set!)   evaluate-set!  )
                ((while)  evaluate-while )
                (else     (evaluate-application operator))) operands) continue environment)))
        (else
         (continue expression environment))))

;
; read-eval-print
;

    (display output)
    (newline)
    (display ">>>")
    (evaluate (read) loop environment))

  (loop "cpSlip version 1" '()))
cpSlip version 1
>>>(begin
  (define (fibonacci p q r)
     (if (> p 1)
        (fibonacci (- p 1) r (+ q r))
        r))
  (fibonacci 15 1 1))
987
>>>
```

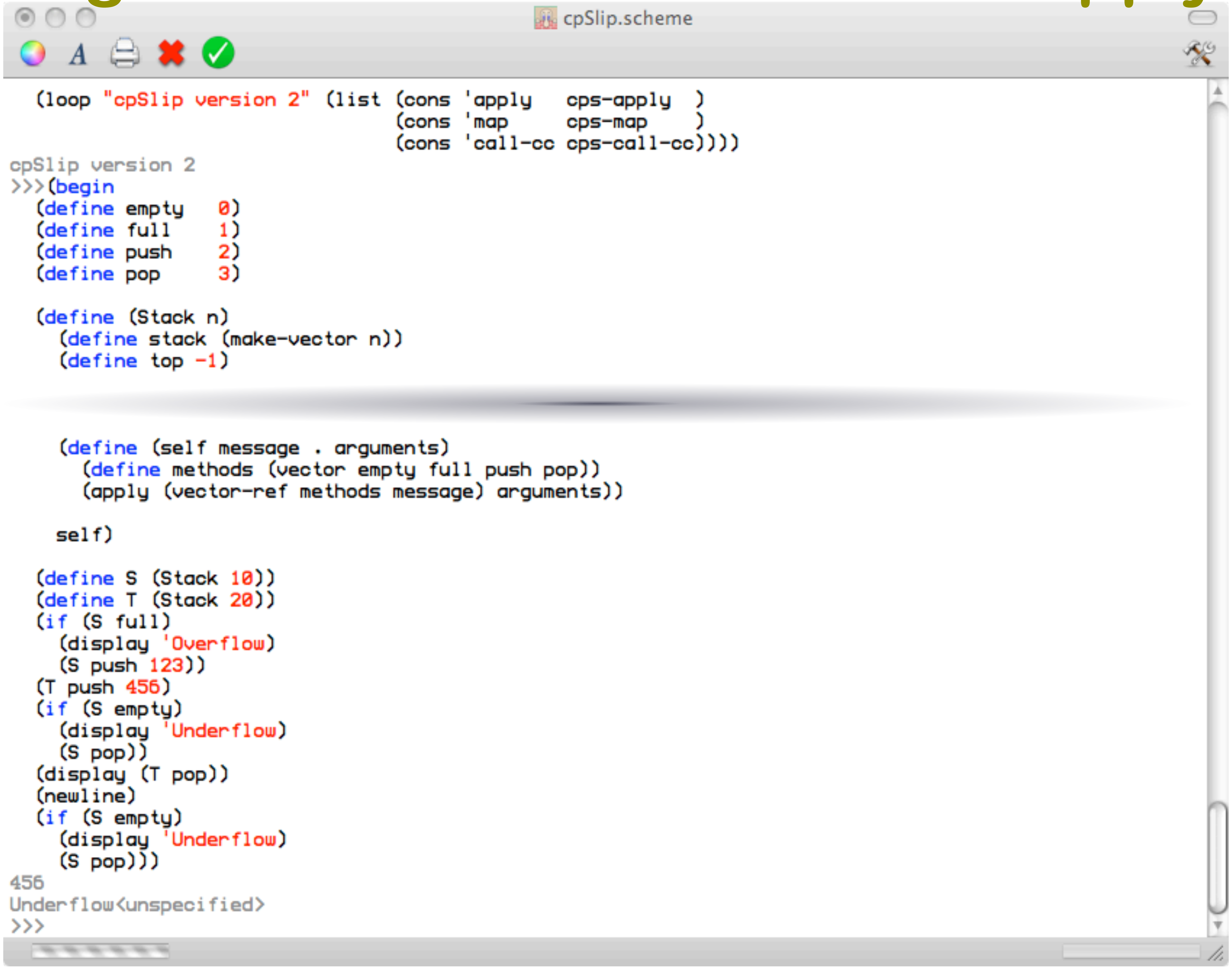
Higher order native functions

```
;
; read-eval-print
;
```

```
(define (cps-apply expression continue environment)
  (define procedure (car expression))
  (define arguments (cadr expression))
  (procedure arguments continue environment))
```

cps*

Higher order native functions: apply



cpSlip.scheme

```
  (loop "cpSlip version 2" (list (cons 'apply   cps-apply   )
                                 (cons 'map     cps-map     )
                                 (cons 'call-cc cps-call-cc))))

cpSlip version 2
>>>(begin
  (define empty   0)
  (define full    1)
  (define push    2)
  (define pop     3)

  (define (Stack n)
    (define stack (make-vector n))
    (define top -1)
```

```
    (define (self message . arguments)
      (define methods (vector empty full push pop))
      (apply (vector-ref methods message) arguments))

    self)

  (define S (Stack 10))
  (define T (Stack 20))
  (if (S full)
    (display 'Overflow)
    (S push 123))
  (T push 456)
  (if (S empty)
    (display 'Underflow)
    (S pop))
  (display (T pop))
  (newline)
  (if (S empty)
    (display 'Underflow)
    (S pop)))
456
Underflow<unspecified>
>>>
```

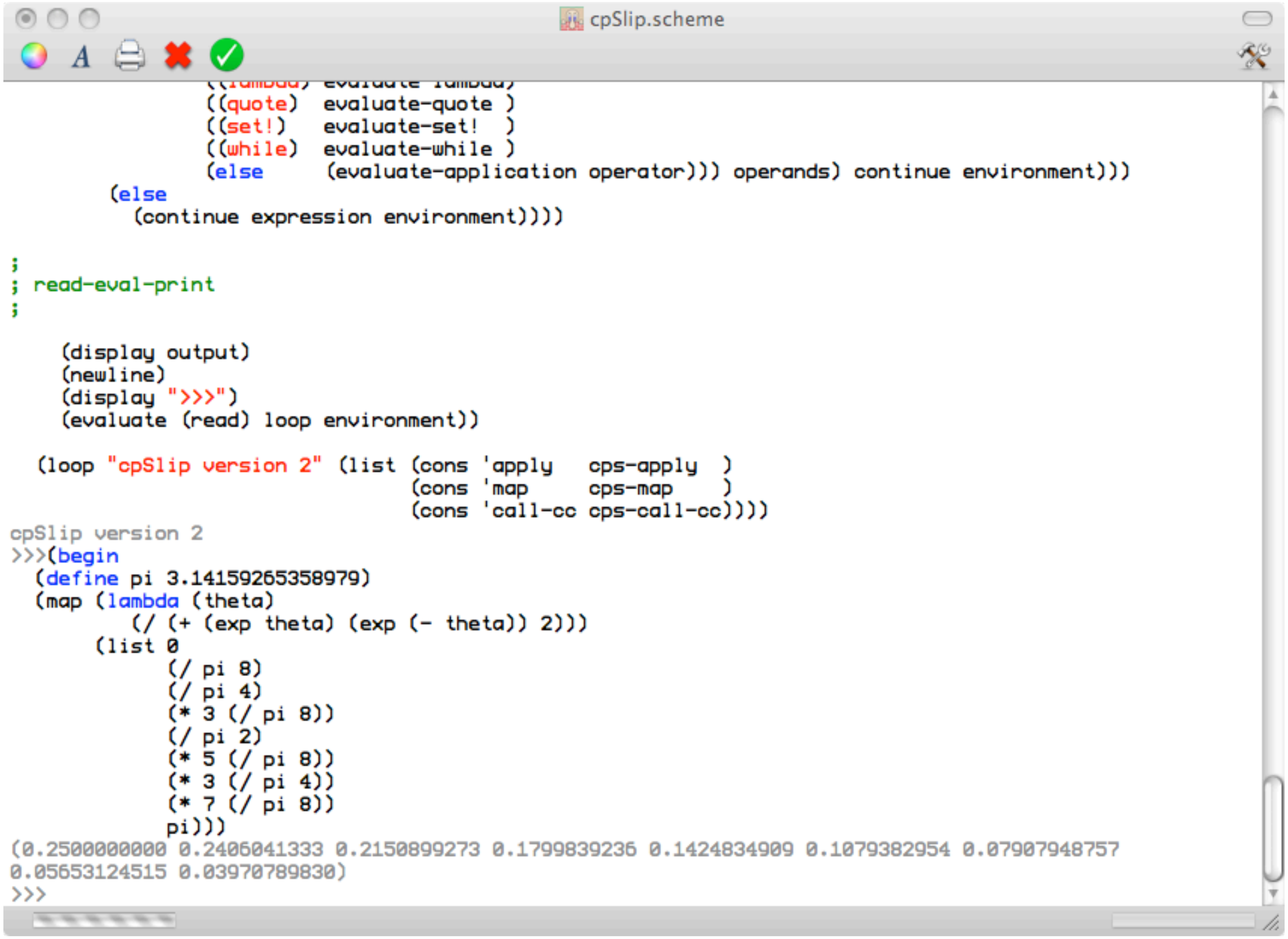
Higher order native functions

```
;
; read-eval-print
;
```

```
(define (cps-map expression continue environment)
  (define procedure (car expression))
  (define items (cadr expression))
  (define (iterate items values)
    (define (continue-after-item value environment)
      (iterate (cdr items) (cons value values)))
    (if (null? items)
      (continue (reverse values) environment)
      (procedure (list (car items)) continue-after-item
                                    environment)))
  (iterate items '()))
```

cps*

Higher order native functions: map



```
                ((lambda)  evaluate-lambda)
                ((quote)   evaluate-quote )
                ((set!)    evaluate-set!  )
                ((while)   evaluate-while )
                (else      (evaluate-application operator))) operands) continue environment)))
        (else
          (continue expression environment))))

;
; read-eval-print
;

     (display output)
     (newline)
     (display ">>>")
     (evaluate (read) loop environment))

  (loop "cpSlip version 2" (list (cons 'apply   cps-apply   )
                                 (cons 'map     cps-map     )
                                 (cons 'call-cc cps-call-cc))))
cpSlip version 2
>>>(begin
  (define pi 3.14159265358979)
  (map (lambda (theta)
          (/ (+ (exp theta) (exp (- theta)) 2)))
       (list 0
             (/ pi 8)
             (/ pi 4)
             (* 3 (/ pi 8))
             (/ pi 2)
             (* 5 (/ pi 8))
             (* 3 (/ pi 4))
             (* 7 (/ pi 8))
             pi)))
(0.2500000000 0.2406041333 0.2150899273 0.1799839236 0.1424834909 0.1079382954 0.07907948757
0.05653124515 0.03970789830)
>>>
```

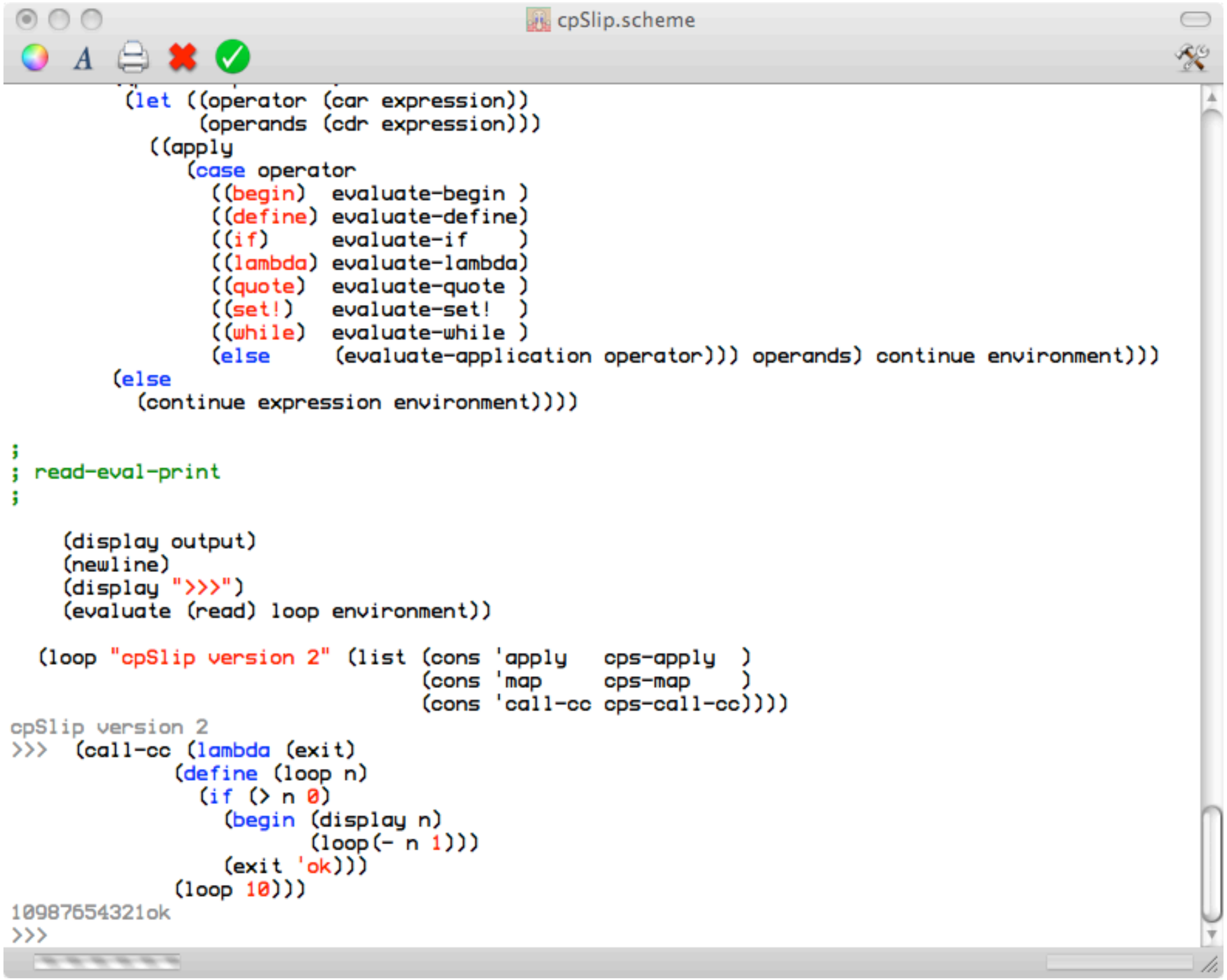
Higher order native functions

```
;
; read-eval-print
;
```

```
(define (cps-call-cc expression continue environment)
  (define procedure (car expression))
  (define value
    (call-with-current-continuation
      (lambda (continuation)
        (define wrapped-continuation
                (wrap-native-procedure continuation))
        (procedure (list wrapped-continuation) continue
                   environment))))
  (continue value environment))
```

cps*

Higher order native functions: call-cc


cpSlip.scheme

```
          (let ((operator (car expression))
                (operands (cdr expression)))
            ((apply
               (case operator
                 ((begin)  evaluate-begin )
                 ((define) evaluate-define)
                 ((if)     evaluate-if    )
                 ((lambda) evaluate-lambda)
                 ((quote)  evaluate-quote )
                 ((set!)   evaluate-set!  )
                 ((while)  evaluate-while )
                 (else     (evaluate-application operator))) operands) continue environment))
         (else
           (continue expression environment))))

;
; read-eval-print
;

    (display output)
    (newline)
    (display ">>>")
    (evaluate (read) loop environment))

  (loop "cpSlip version 2" (list (cons 'apply   cps-apply  )
                                 (cons 'map     cps-map    )
                                 (cons 'call-cc cps-call-cc))))
cpSlip version 2
>>> (call-cc (lambda (exit)
               (define (loop n)
                 (if (> n 0)
                   (begin (display n)
                          (loop(- n 1)))
                   (exit 'ok)))
               (loop 10)))
10987654321ok
>>>
```

First-class continuations: exit

```
(call-cc (lambda (exit)
           (define (loop n)
             (if (> n 0)
               (begin (display n)
                      (loop(- n 1)))
               (exit 'ok)))
           (loop 10)))
```

First-class continuations: return

```
(begin
  (define (list-length obj)
    (call-cc
      (lambda (return)
        (define (r obj)
          (if (null? obj)
            0
            (if (pair? obj)
              (+ (r (cdr obj)) 1)
              (return #f))))
        (r obj))))

  (display (list-length '(1 2 3 4)))
  (newline)
  (display (list-length '(a b . c)))
  (newline))
```

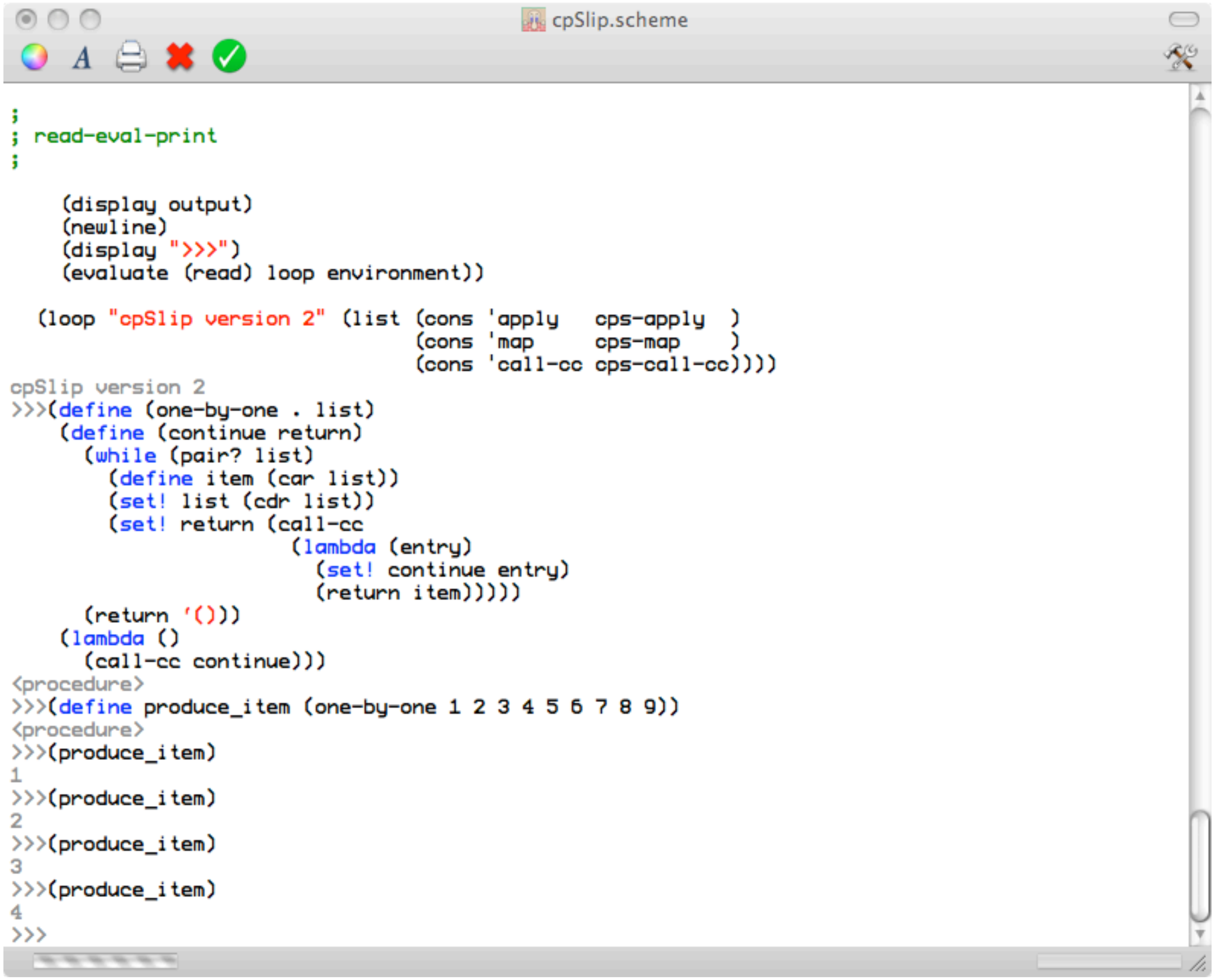
First-class continuations: select

```
(call-cc
  (lambda (exit)
    (define (iter list)
      (if (not (null? list))
        (if (negative? (car list))
          (exit (car list))
          (iter (cdr list)))
        '()))
    (iter '(54 0 37 -3 245 19))))
```

First-class continuations: one-by-one

```
(begin
  (define (one-by-one . list)
    (define (continue return)
      (while (pair? list)
        (define item (car list))
        (set! list (cdr list))
        (set! return (call-cc
                       (lambda (entry)
                         (set! continue entry)
                         (return item)))))
      (return '()))
    (lambda ()
      (call-cc continue)))
  (define produce_item (one-by-one 1 2 3 4 5 6 7 8 9))
  (display (produce_item))
  (display (produce_item))
  (display (produce_item))
  (display (produce_item)))
```

First-class continuations: one-by-one



```
;
; read-eval-print
;

    (display output)
    (newline)
    (display ">>>")
    (evaluate (read) loop environment))

  (loop "cpSlip version 2" (list (cons 'apply   cps-apply   )
                                 (cons 'map     cps-map     )
                                 (cons 'call-cc cps-call-cc))))
cpSlip version 2
>>>(define (one-by-one . list)
    (define (continue return)
      (while (pair? list)
        (define item (car list))
        (set! list (cdr list))
        (set! return (call-cc
                       (lambda (entry)
                         (set! continue entry)
                         (return item)))))
      (return '()))
    (lambda ()
      (call-cc continue)))
<procedure>
>>>(define produce_item (one-by-one 1 2 3 4 5 6 7 8 9))
<procedure>
>>>(produce_item)
1
>>>(produce_item)
2
>>>(produce_item)
3
>>>(produce_item)
4
>>>
```

Native call-cc

```
;
; read-eval-print
;
```

```
(define (cps-call-cc expression continue environment)
  (define procedure (car expression))
  (define (continuation arguments dynamic-continue
                                  dynamic-environment)
    (continue (car arguments) environment))
  (procedure (list continuation) continue environment))
```

cps*

Native call-cc

```
begin ParallelPrimes
start task A
A finds a prime: 2
start task B
B finds a prime: 2
start task C
C finds a prime: 2
A finds a prime: 3
B finds a prime: 3
C finds a prime: 3
A finds a prime: 5
B finds a prime: 5
C finds a prime: 5
A finds a prime: 7
B finds a prime: 7
C finds a prime: 7
A finds a prime: 11
B finds a prime: 11
C finds a prime: 11
stop task A
B finds a prime: 13
C finds a prime: 13
B finds a prime: 17
C finds a prime: 17
B finds a prime: 19
C finds a prime: 19
B finds a prime: 23
stop task C
B finds a prime: 29
stop task B
end ParallelPrimes
<unspecified>
```

A: primes < 12
B: primes < 30
C: primes < 20

Native call-cc

```
(begin
  (define *first-task* ())
  (define *last-task* ())
  (define *return* ())

  (define (enqueue cont)
    (if (null? *last-task*)
      (begin
        (set! *last-task* (list cont))
        (set! *first-task* *last-task*))
      (begin
        (set-cdr! *last-task* (list cont))
        (set! *last-task* (cdr *last-task*)))))

  (define (dequeue)
    (if (not (null? *first-task*))
      (begin
        (define cont (car *first-task*))
        (set! *first-task* (cdr *first-task*))
        (if (null? *first-task*)
          (set! *last-task* ()))
          (cont #f))
        (*return* #f)))

  (define (schedule task)
    (enqueue (lambda (ignore)
               (task)
               (dequeue))))

  (define (yield)
    (call-cc
      (lambda (cont)
        (enqueue cont)
        (dequeue))))

  (define (start)
    (call-cc
     (lambda (cont)
       (set! *return* cont)
       (dequeue))))
```

```
(define (print op id)
  (display op)
  (display " task ")
  (display id)
  (newline))

(define (report id prime)
  (display id)
  (display " finds a prime: ")
  (display prime)
  (newline)
  (yield))

(define (primes id n)
  (define mask (make-vector n #t))
  (define limit (sqrt n))
  (print "start" id)
  (define i 2)
  (while (<= i limit)
    (if (vector-ref mask i)
      (begin
        (report id i)
        (define j (+ i i))
        (while (< j n)
          (vector-set! mask j #f)
          (set! j (+ j i)))))
    (set! i (+ i 1)))
  (while (< i n)
    (if (vector-ref mask i )
      (report id i))
    (set! i (+ i 1)))
  (print "stop" id))

(display "begin ParallelPrimes")
(newline)
(schedule (lambda () (primes 'A 12)))
(schedule (lambda () (primes 'B 30)))
(schedule (lambda () (primes 'C 20)))
(start)
(display "end ParallelPrimes")
(newline))
```

Proper tailcalls

```
(begin
  (define meta-level-eval eval)

;
; natives
;









    (display output)
    (newline)
    (display ">>>")
    (evaluate (read) loop environment #f))

  (loop "cpSlip version 4" (list (cons 'apply    cps-apply   )
                                 (cons 'map      cps-map     )
                                 (cons 'for-each cps-for-each)
                                 (cons 'call-cc  cps-call-cc ))))
```

Proper tailcalls

```
(begin
  (define meta-level-eval eval)

;
; natives
;




    (display output)
    (newline)
    (display ">>>")
    (evaluate (read) loop environment #f))

  (loop "cpSlip version 4" (list (cons 'apply    cps-apply   )
                                 (cons 'map      cps-map     )
                                 (cons 'for-each cps-for-each)
                                 (cons 'call-cc  cps-call-cc ))))
```

Proper tailcalls

```
(define (evaluate expression continue environment tailcall)
   (cond
     ((symbol? expression)
      (evaluate-variable expression continue environment))
     ((pair? expression)
      (let ((operator (car expression))
            (operands (cdr expression)))
        ((apply
           (case operator
             ((begin)  evaluate-begin )
             ((define) evaluate-define)
             ((if)     evaluate-if    )
             ((lambda) evaluate-lambda)
             ((quote)  evaluate-quote )
             ((set!)   evaluate-set!  )
             ((while)  evaluate-while )
             (else     (evaluate-application operator))) operands)
                                        continue environment tailcall)))
     (else
       (continue expression environment))))
```

Proper tailcalls

```
(define (evaluate expression continue environment tailcall)
   (cond
     ((symbol? expression)
      (evaluate-variable expression continue environment))
     ((pair? expression)
      (let ((operator (car expression))
            (operands (cdr expression)))
        ((apply
           (case operator
             ((begin)  evaluate-begin )
             ((define) evaluate-define)
             ((if)     evaluate-if    )
             ((lambda) evaluate-lambda)
             ((quote)  evaluate-quote )
             ((set!)   evaluate-set!  )
             ((while)  evaluate-while )
             (else     (evaluate-application operator))) operands)
                                  continue environment tailcall)))
     (else
       (continue expression environment))))
```

Proper tailcalls

```
(define (evaluate-set! variable expression)
  (lambda (continue environment tailcall)
    (define (continue-after-expression value environment-after-expression)
      (define binding (assoc variable environment-after-expression))
      (if binding
        (set-cdr! binding value)
        (error "inaccessible variable: " variable))
      (continue value environment-after-expression))
    (evaluate expression continue-after-expression environment #f)))
```

```
(define (evaluate-if predicate consequent . alternative)
  (lambda (continue environment tailcall)
    (define (continue-after-predicate boolean environment-after-predicate)
      (if (eq? boolean #f)
        (if (null? alternative)
          (continue '() environment-after-predicate)
          (evaluate (car alternative) continue environment-after-predicate tailcall))
        (evaluate consequent continue environment-after-predicate tailcall)))
    (evaluate predicate continue-after-predicate environment #f)))
```

Proper tailcalls

```
(define (evaluate-set! variable expression)
  (lambda (continue environment tailcall)
    (define (continue-after-expression value environment-after-expression)
      (define binding (assoc variable environment-after-expression))
      (if binding
        (set-cdr! binding value)
        (error "inaccessible variable: " variable))
      (continue value environment-after-expression))
    (evaluate expression continue-after-expression environment #f)))
```

```
(define (evaluate-if predicate consequent . alternative)
  (lambda (continue environment tailcall)
    (define (continue-after-predicate boolean environment-after-predicate)
      (if (eq? boolean #f)
        (if (null? alternative)
          (continue '() environment-after-predicate)
          (evaluate (car alternative) continue environment-after-predicate tailcall))
        (evaluate consequent continue environment-after-predicate tailcall)))
    (evaluate predicate continue-after-predicate environment #f)))
```

Proper tailcalls

```
(define (make-procedure parameters expressions environment)
  (lambda (arguments continue dynamic-environment tailcall)
    (define (continue-after-sequence value environment-after-sequence)
      (continue value dynamic-environment))
    (define lexical-environment (bind-parameters parameters arguments environment))
    (if tailcall
      (evaluate-sequence expressions continue lexical-environment #t)
      (evaluate-sequence expressions continue-after-sequence lexical-environment #t))))
```

```
(define (evaluate-sequence expressions continue environment tailcall)
  (define head (car expressions))
  (define tail (cdr expressions))
  (define (continue-with-sequence value environment)
    (evaluate-sequence tail continue environment tailcall))
  (if (null? tail)
    (evaluate head continue environment tailcall)
    (evaluate head continue-with-sequence environment #f)))
```

Proper tailcalls

```
(define (make-procedure parameters expressions environment)
  (lambda (arguments continue dynamic-environment tailcall)
    (define (continue-after-sequence value environment-after-sequence)
      (continue value dynamic-environment))
    (define lexical-environment (bind-parameters parameters arguments environment))
    (if tailcall
      (evaluate-sequence expressions continue lexical-environment #t)
      (evaluate-sequence expressions continue-after-sequence lexical-environment #t))))
```

```
(define (evaluate-sequence expressions continue environment tailcall)
  (define head (car expressions))
  (define tail (cdr expressions))
  (define (continue-with-sequence value environment)
    (evaluate-sequence tail continue environment tailcall))
  (if (null? tail)
    (evaluate head continue environment tailcall)
    (evaluate head continue-with-sequence environment #f)))
```

Proper tailcalls

```
(define (evaluate-application operator)
   (lambda operands
     (lambda (continue environment tailcall)
       (define (continue-after-operator procedure environment-after-operator)
         (define (evaluate-operands operands arguments environment)
           (define (continue-with-operands value environment-with-operands)
             (evaluate-operands (cdr operands) (cons value arguments)
                                                         environment-with-operands))
           (if (null? operands)
             (procedure (reverse arguments) continue environment tailcall)
             (evaluate (car operands) continue-with-operands environment #f)))
         (evaluate-operands operands '() environment-after-operator))
       (evaluate operator continue-after-operator environment #f))))
```

Proper tailcalls

```
(define (evaluate-application operator)
   (lambda operands
     (lambda (continue environment tailcall)
       (define (continue-after-operator procedure environment-after-operator)
         (define (evaluate-operands operands arguments environment)
           (define (continue-with-operands value environment-with-operands)
             (evaluate-operands (cdr operands) (cons value arguments)
                                                               environment-with-operands))
           (if (null? operands)
             (procedure (reverse arguments) continue environment tailcall)
             (evaluate (car operands) continue-with-operands environment #f)))
         (evaluate-operands operands '() environment-after-operator))
       (evaluate operator continue-after-operator environment #f))))
```

Proper tailcalls

```scheme
(begin
  (define meta-level-eval eval)

  (define (wrap-native-procedure native-procedure)
    (lambda (arguments continue environment tailcall)
      (define native-value (apply native-procedure arguments))
      (continue native-value environment)))

  (define (cps-apply expression continue environment tailcall)
    (define procedure (car expression))
    (define arguments (cadr expression))
    (procedure arguments continue environment tailcall))

  (define (cps-map expression continue environment tailcall)
    (define procedure (car expression))
    (define items (cadr expression))
    (define (iterate items values)
      (if (null? items)
        (continue (reverse values) environment)
        (let ((head (car items))
              (tail (cdr items)))
          (define (continue-after-item value environment)
            (iterate tail (cons value values)))
          (procedure (list head) continue-after-item environment #f))))
    (iterate items '()))

 (define (cps-for-each expression continue environment tailcall)
    (define procedure (car expression))
    (define items (cadr expression))
    (define (iterate items)
      (if (null? items)
        (continue '() environment)
        (let ((head (car items))
              (tail (cdr items)))
          (define (continue-after-item value environment)
            (iterate tail))
          (procedure (list head) continue-after-item environment #f))))
    (iterate items))

  (define (cps-call-cc expression continue environment tailcall)
    (define procedure (car expression))
    (define (continuation arguments dynamic-continue dynamic-environment tailcall)
      (continue (car arguments) environment))
    (procedure (list continuation) continue environment tailcall))

  (define (loop output environment)
    (define rollback environment)

    (define (error message qualifier)
      (display message)
      (loop qualifier rollback))

    (define (bind-variable variable value environment)
      (define binding (cons variable value))
      (cons binding environment))

    (define (bind-parameters parameters arguments environment)
      (if (symbol? parameters)
        (bind-variable parameters arguments environment)
        (if (pair? parameters)
          (let*
            ((variable (car parameters))
             (value (car arguments ))
             (environment (bind-variable variable value environment)))
            (bind-parameters (cdr parameters) (cdr arguments) environment))
          environment)))

    (define (evaluate-sequence expressions continue environment tailcall)
      (define head (car expressions))
      (define tail (cdr expressions))
      (define (continue-with-sequence value environment)
        (evaluate-sequence tail continue environment tailcall))
      (if (null? tail)
        (evaluate head continue environment tailcall)
        (evaluate head continue-with-sequence environment #f)))

    (define (make-procedure parameters expressions environment)
      (lambda (arguments continue dynamic-environment tailcall)
        (define (continue-after-sequence value environment-after-sequence)
          (continue value dynamic-environment))
        (define lexical-environment (bind-parameters parameters arguments environment))
        (if tailcall
          (evaluate-sequence expressions continue lexical-environment #t)
          (evaluate-sequence expressions continue-after-sequence lexical-environment #t))))

    (define (evaluate-application operator)
      (lambda operands
        (lambda (continue environment tailcall)
          (define (continue-after-operator procedure environment-after-operator)
            (define (evaluate-operands operands arguments environment)
              (define (continue-with-operands value environment-with-operands)
                (evaluate-operands (cdr operands) (cons value arguments) environment-with-operands))
              (if (null? operands)
                (procedure (reverse arguments) continue environment tailcall)
                (evaluate (car operands) continue-with-operands environment #f)))
            (evaluate-operands operands '() environment-after-operator))
          (evaluate operator continue-after-operator environment #f))))

  (define (evaluate-begin . expressions)
    (lambda (continue environment tailcall)
      (evaluate-sequence expressions continue environment tailcall)))

  (define (evaluate-define pattern . expressions)
    (lambda (continue environment tailcall)
      (define (continue-after-expression value environment-after-expression)
        (define binding (cons pattern value))
        (continue value (cons binding environment-after-expression)))
      (if (symbol? pattern)
        (evaluate (car expressions) continue-after-expression environment #f))
        (let* ((binding (cons (car pattern) '()))
               (environment (cons binding environment))
               (procedure (make-procedure (cdr pattern) expressions environment)))
          (set-cdr! binding procedure)
          (continue procedure environment))))

  (define (evaluate-if predicate consequent . alternative)
    (lambda (continue environment tailcall)
      (define (continue-after-predicate boolean environment-after-predicate)
        (if (eq? boolean #f)
          (if (null? alternative)
            (continue '() environment-after-predicate)
            (evaluate (car alternative) continue environment-after-predicate tailcall))
          (evaluate consequent continue environment-after-predicate tailcall)))
      (evaluate predicate continue-after-predicate environment #f)))

  (define (evaluate-lambda parameters . expressions)
    (lambda (continue environment tailcall)
      (continue (make-procedure parameters expressions environment) environment)))

  (define (evaluate-quote expression)
    (lambda (continue environment tailcall)
      (continue expression environment)))

  (define (evaluate-set! variable expression)
    (lambda (continue environment tailcall)
      (define (continue-after-expression value environment-after-expression)
        (define binding (assoc variable environment-after-expression))
        (if binding
          (set-cdr! binding value)
          (error "inaccessible variable: " variable))
        (continue value environment-after-expression))
      (evaluate expression continue-after-expression environment #f)))

  (define (evaluate-variable variable continue environment)
    (define binding (assoc variable environment))
    (if binding
      (continue (cdr binding) environment)
      (let ((native-value (meta-level-eval variable (interaction-environment))))
        (if (procedure? native-value)
          (continue (wrap-native-procedure native-value) environment)
          (continue native-value environment)))))

  (define (evaluate-while predicate . expressions)
    (lambda (continue environment tailcall)
      (define (iterate value environment)
        (define (continue-after-predicate boolean environment-after-predicate)
          (define (continue-after-sequence value environment-after-sequence)
            (iterate value environment))
          (if (eq? boolean #f)
            (continue value environment)
            (evaluate-sequence expressions continue-after-sequence environment #f)))
        (evaluate predicate continue-after-predicate environment #f))
      (iterate '() environment)))

  (define (evaluate expression continue environment tailcall)
    (cond
      ((symbol? expression)
       (evaluate-variable expression continue environment))
      ((pair? expression)
       (let ((operator (car expression))
             (operands (cdr expression)))
         ((apply
            (case operator
              ((begin)  evaluate-begin )
              ((define) evaluate-define)
              ((if)     evaluate-if    )
              ((lambda) evaluate-lambda)
              ((quote)  evaluate-quote )
              ((set!)   evaluate-set!  )
              ((while)  evaluate-while )
              (else     (evaluate-application operator))) operands) continue environment tailcall)))
      (else
        (continue expression environment))))

  (display output)
  (newline)
  (display ">>>")
  (evaluate (read) loop environment #f))

(loop "cpSlip* version 3" (list (cons 'apply    cps-apply   )
                                (cons 'map      cps-map     )
                                (cons 'for-each cps-for-each)
                                (cons 'call-cc  cps-call-cc ))))
```

Proper tailcalls

```
(begin
  (define meta-level-eval eval)

  (define (wrap-native-procedure native-procedure)
    (lambda (arguments continue environment tailcall)
      (define native-value (apply native-procedure arguments))
      (continue native-value environment)))

  (define (cps-apply expression continue environment tailcall)
    (define procedure (car expression))
    (define arguments (cadr expression))
    (procedure arguments continue environment tailcall))

  (define (cps-map expression continue environment tailcall)
    (define procedure (car expression))
    (define items (cadr expression))
    (define (iterate items values)
      (if (null? items)
        (continue (reverse values) environment)
        (let ((head (car items))
              (tail (cdr items)))
          (define (continue-after-item value environment)
            (iterate tail (cons value values)))
          (procedure (list head) continue-after-item environment #f))))
    (iterate items '()))

 (define (cps-for-each expression continue environment tailcall)
    (define procedure (car expression))
    (define items (cadr expression))
    (define (iterate items)
      (if (null? items)
        (continue '() environment)
        (let ((head (car items))
              (tail (cdr items)))
          (define (continue-after-item value environment)
            (iterate tail))
          (procedure (list head) continue-after-item environment #f))))
    (iterate items))

  (define (cps-call-cc expression continue environment tailcall)
    (define procedure (car expression))
    (define (continuation arguments dynamic-continue dynamic-environment tailcall)
      (continue (car arguments) environment))
    (procedure (list continuation) continue environment tailcall))

  (define (loop output environment)
    (define rollback environment)

    (define (error message qualifier)
      (display message)
      (loop qualifier rollback))

    (define (bind-variable variable value environment)
      (define binding (cons variable value))
      (cons binding environment))

    (define (bind-parameters parameters arguments environment)
      (if (symbol? parameters)
        (bind-variable parameters arguments environment)
        (if (pair? parameters)
          (let*
            ((variable (car parameters))
             (value (car arguments ))
             (environment (bind-variable variable value environment)))
            (bind-parameters (cdr parameters) (cdr arguments) environment))
          environment)))

    (define (evaluate-sequence expressions continue environment tailcall)
      (define head (car expressions))
      (define tail (cdr expressions))
      (define (continue-with-sequence value environment)
        (evaluate-sequence tail continue environment tailcall))
      (if (null? tail)
        (evaluate head continue environment tailcall)
        (evaluate head continue-with-sequence environment #f)))

    (define (make-procedure parameters expressions environment)
      (lambda (arguments continue dynamic-environment tailcall)
        (define (continue-after-sequence value environment-after-sequence)
          (continue value dynamic-environment))
        (define lexical-environment (bind-parameters parameters arguments environment))
        (if tailcall
          (evaluate-sequence expressions continue lexical-environment #t)
          (evaluate-sequence expressions continue-after-sequence lexical-environment #t))))

    (define (evaluate-application operator)
      (lambda operands
        (lambda (continue environment tailcall)
          (define (continue-after-operator procedure environment-after-operator)
            (define (evaluate-operands operands arguments environment)
              (define (continue-with-operands value environment-with-operands)
                (evaluate-operands (cdr operands) (cons value arguments) environment-with-operands))
              (if (null? operands)
                (procedure (reverse arguments) continue environment tailcall)
                (evaluate (car operands) continue-with-operands environment #f)))
            (evaluate-operands operands '() environment-after-operator))
          (evaluate operator continue-after-operator environment #f))))

    (define (evaluate-begin . expressions)
      (lambda (continue environment tailcall)
        (evaluate-sequence expressions continue environment tailcall)))

    (define (evaluate-define pattern . expressions)
      (lambda (continue environment tailcall)
        (define (continue-after-expression value environment-after-expression)
          (define binding (cons pattern value))
          (continue value (cons binding environment-after-expression)))
        (if (symbol? pattern)
          (evaluate (car expressions) continue-after-expression environment #f))
          (let* ((binding (cons (car pattern) '()))
                 (environment (cons binding environment))
                 (procedure (make-procedure (cdr pattern) expressions environment)))
            (set-cdr! binding procedure)
            (continue procedure environment)))))

    (define (evaluate-if predicate consequent . alternative)
      (lambda (continue environment tailcall)
        (define (continue-after-predicate boolean environment-after-predicate)
          (if (eq? boolean #f)
            (if (null? alternative)
              (continue '() environment-after-predicate)
              (evaluate (car alternative) continue environment-after-predicate tailcall))
            (evaluate consequent continue environment-after-predicate tailcall)))
        (evaluate predicate continue-after-predicate environment #f)))

    (define (evaluate-lambda parameters . expressions)
      (lambda (continue environment tailcall)
        (continue (make-procedure parameters expressions environment) environment)))

    (define (evaluate-quote expression)
      (lambda (continue environment tailcall)
        (continue expression environment)))

    (define (evaluate-set! variable expression)
      (lambda (continue environment tailcall)
        (define (continue-after-expression value environment-after-expression)
          (define binding (assoc variable environment-after-expression))
          (if binding
            (set-cdr! binding value)
            (error "inaccessible variable: " variable))
          (continue value environment-after-expression))
        (evaluate expression continue-after-expression environment #f)))

    (define (evaluate-variable variable continue environment)
      (define binding (assoc variable environment))
      (if binding
        (continue (cdr binding) environment)
        (let ((native-value (meta-level-eval variable (interaction-environment))))
          (if (procedure? native-value)
            (continue (wrap-native-procedure native-value) environment)
            (continue native-value environment)))))

    (define (evaluate-while predicate . expressions)
      (lambda (continue environment tailcall)
        (define (iterate value environment)
          (define (continue-after-predicate boolean environment-after-predicate)
            (define (continue-after-sequence value environment-after-sequence)
              (iterate value environment))
            (if (eq? boolean #f)
              (continue value environment)
              (evaluate-sequence expressions continue-after-sequence environment #f)))
          (evaluate predicate continue-after-predicate environment #f))
        (iterate '() environment)))

    (define (evaluate expression continue environment tailcall)
      (cond
        ((symbol? expression)
         (evaluate-variable expression continue environment))
        ((pair? expression)
         (let ((operator (car expression))
               (operands (cdr expression)))
           ((apply
              (case operator
                ((begin)  evaluate-begin )
                ((define) evaluate-define)
                ((if)     evaluate-if    )
                ((lambda) evaluate-lambda)
                ((quote)  evaluate-quote )
                ((set!)   evaluate-set!  )
                ((while)  evaluate-while )
                (else     (evaluate-application operator))) operands) continue environment tailcall)))
        (else
          (continue expression environment))))

    (display output)
    (newline)
    (display ">>>")
    (evaluate (read) loop environment #f))

 (loop "cpSlip* version 3" (list (cons 'apply    cps-apply   )
                                 (cons 'map      cps-map     )
                                 (cons 'for-each cps-for-each)
                                 (cons 'call-cc  cps-call-cc ))))
```

aspects