CS 565 August 25, 1996
Jens Palsberg Readings: EOPL, Secs 8.1–8.5, 9.1

Lecture Note 5:

Continuations

Converting to Continuation-Passing Style

Finding programs with iterative behavior

Want to go back to *recursive* vs. *iterative* behavior:

```
(define fact
  (lambda (n)
    (if (zero? n) 1 (* n (fact (sub1 n))))))

(define fact-iter
  (lambda (n)
    (fact-iter1 n 1)))

(define fact-iter1
  (lambda (n a)
    (if (zero? n) a (fact-iter1 (sub1 n) (* n a)))))
```



```
  (fact 4)
= (* 4 (fact 3))
= (* 4 (* 3 (fact 2)))
= (* 4 (* 3 (* 2 (fact 1))))
= (* 4 (* 3 (* 2 (* 1 (fact 0)))))
= (* 4 (* 3 (* 2 (* 1 1))))
= 24
```

```
  (fact-iter 4)
= (fact-iter1 4 1)
= (fact-iter1 3 4)
= (fact-iter1 2 12)
= (fact-iter1 1 24)
= (fact-iter1 0 24)
= 24
```


A program with iterative behavior can be written in flowchart-like pseudocode:

```
fact:       a := 1
fact-iter1: if (zero? n) then return a
                         else begin a := n*a
                                    n := n-1
                                    goto fact-iter1
                              end
```



```
(define even-length?
  (lambda (l)
    (if (null? l) #t (odd-length? (cdr l)))))

(define odd-length?
  (lambda (l)
    (if (null? l) #f (even-length? (cdr l)))))
```

```
even-length?: if (null? l) then return #t
                           else begin l := (cdr l);
                                      goto odd-length?
                           end
odd-length?:  if (null? l) then return #f
                           else begin l := (cdr l);
                                      goto even-length?
                           end
```

Goal: write a grammar to describe programs that look like this. Will need two non-terminals: ⟨simple-exp⟩, for expressions that can go on the right-hand sides of assignments, and ⟨tail-form-exp⟩, for the programs themselves.

```
⟨tail-form-exp⟩ ::= ⟨simple-exp⟩
                ::= ( ⟨simple-exp⟩ {⟨simple-exp⟩}* )
                ::= ( if ⟨simple-exp⟩ ⟨tail-form-exp⟩ ⟨tail-form-exp⟩ )
⟨simple-exp⟩    ::= ⟨identifier⟩
                ::= ⟨constant⟩
                ::= ( ⟨simple operation⟩ {⟨simple-exp⟩}* )
                ::= ( lambda ( {⟨identifier⟩}* ) ⟨tail-form-exp⟩ )
```

Tail form positions are those where the subexpression, when evaluated, gives the value of the whole expression (no promises or wrapping). All other positions must be simple.

Book is much more formal about this.

```
(car x)                         tail form (also simple)

(car (cdr x))                   tail form (also simple)

(car (f x))                     not tail form

(f (car x))                     tail form
```



```
(lambda (v) (k (+ v 1)))        tail form

(lambda (v) (+ (k v) 1))        not tail form

(if (zero? x) (f (+ x 1)) (g (- x 1)))   tail form

(if (zero? x) (+ (f x) 1) (g (- x 1)))   not tail form

(if (p x) (f (+ x 1)) (g (- x 1)))       not tail form

(if (p (car x)) (f (+ x 1)) (g (- x 1))) not tail form

(lambda (x)                      ; not tail form
  (if (zero? x) 0 (+ (f (sub1 x)) 1)))

(lambda (x)                          ; tail form
  (if (zero? x) 0 (f (sub1 x) (lambda (v) (k (+ v 1))))))

(lambda (n a)                          ; tail form
  (if (zero? n) a (fact-iter1 (sub1 n) (* n a))))
```

A more complete grammar:

```
⟨tail-form-exp⟩ ::= ( cond { ( ⟨simple-exp⟩ ⟨tail-form-exp⟩ ) }* )
                ::= ( let ( { ( ⟨identifier⟩ ⟨simple-exp⟩ ) }* ) ⟨tail-form-exp⟩ )
                ::= ( letrec ( { ( ⟨identifier⟩ ⟨simple-exp⟩ ) }* ) ⟨tail-form-exp⟩ )
                ::= ( begin {⟨simple-exp⟩}* ⟨tail-form-exp⟩ )
                ::= ( set! ⟨identifier⟩ ⟨simple-exp⟩ )
```


Note: our ⟨simple⟩ corresponds to what the book calls "simple *and* tail-form" (sorry).

Converting to Tail Form

```
(define fact
  (lambda (n)
    (if (zero? n) 1 (* n (fact (sub1 n))))))
```

Typical state:

```
(* 6 (* 5 (* 4 (fact 3))))
```

Let's write another version of `fact`, called `(fact-cps n g)`, which computes `(g (fact n))`.

```
(define fact-cps
  (lambda (n g)
    (if (zero? n)
        (g 1)
        (fact-cps (sub1 n)
          (lambda (v) (g (* n v)))))))
```

```
(define foo
  (lambda (x y) -----))

==>

(define foo-cps
  (lambda (x y k) (k --------)))
```

```
(k (foo a (sub1 n)))

==>

(foo-cps a (sub1 n) k)
```

```
(k (----- (foo a (sub1 n)) -----))

==>

(foo-cps a (sub1 n) (lambda (v) (k (----- v ----)))
```



```
(define remove-all
  (lambda (a lsym)
    (cond
      ((null? lsym) '())
      ((eq? a (car lsym))
       (remove-all a (cdr lsym)))
      (else
        (cons
          (car lsym)
          (remove-all a (cdr lsym)))))))
```

```
(define remove-all-cps
  (lambda (a lsym k)
    (k (cond
         ((null? lsym) '())
         ((eq? a (car lsym))
          (remove-all a (cdr lsym)))
         (else
           (cons
             (car lsym)
             (remove-all a (cdr lsym))))))))
```



```
(define remove-all-cps
  (lambda (a lsym k)
    (cond
      ((null? lsym) (k '()))
      ((eq? a (car lsym))
       (k (remove-all a (cdr lsym))))
      (else
        (k
          (cons
            (car lsym)
            (remove-all a (cdr lsym))))))))
```

```
(k (cons
     (car lsym)
     (remove-all a (cdr lsym))))

==>

(remove-all-cps a (cdr lsym)
  (lambda (v)
    (k (cons
         (car lsym)
         v))))
```



```
(define remove-all-cps
  (lambda (a lsym k)
    (cond ((null? lsym) (k '()))
          ((eq? a (car lsym))
           (remove-all-cps a (cdr lsym) k))
          (else
            (remove-all-cps a (cdr lsym)
              (lambda (v)
                (k (cons
                     (car lsym)
                     v))))))))
```

```
(define subst
  (lambda (old new s)
    (if (pair? s)
        (cons (subst old new (car s))
              (subst old new (cdr s)))
        (if (eq? s old) new s))))
```

```
(define subst-cps
  (lambda (old new s k)
    (if (pair? s)
      (k (cons (subst old new (car s))
               (subst old new (cdr s))))
      (if (eq? s old)
        (k new)
        (k s)))))
```



```
(define subst-cps
  (lambda (old new s k)
    (if (pair? s)
      (subst-cps old new (cdr s)
        (lambda (cdr-val)
          (k (cons (subst old new (car s))
                   cdr-val))))
      (if (eq? s old)
        (k new)
        (k s)))))
```

```
(define subst-cps
  (lambda (old new s k)
    (if (pair? s)
        (subst-cps old new (cdr s)
          (lambda (cdr-val)
            (subst-cps old new (car s)
              (lambda (car-val)
                (k (cons car-val
                         cdr-val))))))
        (if (eq? s old)
          (k new)
          (k s)))))
```



```
(define subst-cps
  (lambda (old new s k)
    (if (pair? s)
        (subst-cps old new (car s)
          (lambda (car-val)
            (subst-cps old new (cdr s)
              (lambda (cdr-val)
                (k (cons car-val
                         cdr-val))))))
        (if (eq? s old)
          (k new)
          (k s)))))
```



```
(define subst-with-letrec
  (lambda (old new s)
    (letrec
      ((loop
         ;; (loop s) = (subst old new s)
         (lambda (s)
           (if (pair? s)
             (cons (loop (car s))
                   (loop (cdr s)))
             (if (eq? s old) new s)))))
      (loop s))))

(define subst-with-letrec-cps
  (lambda (old new s k)
    (letrec
      ((cps-loop (lambda (s k)
                   (if (pair? s)
                     (cps-loop (car s)
                           (lambda (car-val)
                             (cps-loop (cdr s)
                                   (lambda (cdr-val)
                                     (k (cons
                                          car-val
                                          cdr-val))))))
                     (if (eq? s old)
                       (k new)
                       (k s))))))
      (cps-loop s k))))
```

```
(define subst-with-letrec
  (lambda (old new s)
    (letrec
      ((cps-loop (lambda (s k)
               (if (pair? s)
                   (cps-loop (car s)
                         (lambda (car-val)
                           (cps-loop (cdr s)
                                 (lambda (cdr-val)
                                   (k (cons
                                        car-val
                                        cdr-val))))))
                   (if (eq? s old)
                     (k new)
                     (k s))))))
      (cps-loop s (lambda (x) x)))))
```


CPSing `if` and `let`

These follow our rules, too!

```
(k (if (foo x) ...  ...))  = (foo-cps x
                               (lambda (v)
                                 (k (if v ... ...))))
                           = (foo-cps x
                               (lambda (v)
                                 (if v (k ...) (k ...))))
```

```
(k (let ((y (foo x))) ...)) = (foo-cps x
                                (lambda (v)
                                  (k (let ((y v)) ...))))
                            = (foo-cps x
                                (lambda (v)
                                  (let ((y v)) (k ...))))
                            = (foo-cps x
                                (lambda (y)
                                  (k ...)))
```


From Tail Form to First-Order Form: Data Structure Representations of Continuations

We've converted to tail form by using continuation-passing style. Now we want to get rid of all those higher-order functions.

```
(define remove-all-cps
  (lambda (a lsym k)
    (cond ((null? lsym)
           (k '()))
          ((eq? a (car lsym))
           (remove-all-cps a (cdr lsym) k))
          (else
            (remove-all-cps a (cdr lsym)
              (lambda (v) (k (cons (car lsym) v))))))))

(define remove-all
  (lambda (a lsym)
    (remove-all-cps a lsym (lambda (v) v))))
```


Continuations as a datatype

Grammar for continuations

```
⟨continuation⟩  ::= (lambda (v) v)
                ::= (lambda (v) (⟨continuation⟩ (cons (car lsym) v)))
```

Interface Specification. (Remember $\lceil k \rceil$ means a representation of k).

```
(make-identity) = ⌈(lambda (v) v)⌉
(make-rem1 lsym ⌈k⌉) = ⌈(lambda (v) (k (cons (car lsym) v)))⌉
(apply-continuation ⌈k⌉ v) = (k v)
```

Representation-Independent Code

```
(define remove-all-cps
  (lambda (a lsym k)
    (cond ((null? lsym)
           (apply-continuation k '()))
          ((eq? a (car lsym))
           (remove-all-cps a (cdr lsym) k))
          (else
            (remove-all-cps a (cdr lsym)
              (make-rem1 lsym k))))))

(define remove-all
  (lambda (a lsym)
    (remove-all-cps a lsym (make-identity))))
```

Procedural representation of continuations

```
(define make-identity
  (lambda ()
    (lambda (v) v)))

(define make-rem1
  (lambda (lsym k)
    (lambda (v)
      (apply-continuation k (cons (car lsym) v)))))

(define apply-continuation
  (lambda (k v)
    (k v)))
```


Abstract Syntax Tree Representation

Representation:

(make-identity) = '(identity-record)
(make-rem1 v k) = '(rem1-record v k)

Here the right-hand sides are patterns for list structures; the value of (make-rem1 v k) is a list whose first element is the symbol rem1, whose second element is the value of v, and whose third element is the value of k.

```
(define-record identity-record ())    ; def make-identity-record
(define-record rem1-record (lsym k))  ; def make-rem1-record
```

```
(define apply-continuation
  (lambda (k v)
    (record-case k
      (identity-record () v)     ; this was (make-identity)
      (rem1-record (lsym k)      ; this was (make-rem1 lsym k)
        (apply-continuation k (cons (car lsym) v)))
      (else (report-bad-continuation k)))))

(define report-bad-continuation
  (lambda (k)
    (error 'apply-continuation
      "bad continuation ~s" k)))
```


Another Representation

Since there's only one possible tag (rem1), we don't need to include it.

(make-identity) = '()
(make-rem1 $lsym$ k) = ($lsym$. k)

```
(define make-identity
  (lambda ()
    '()))

(define make-rem1
  (lambda (lsym k)
    (cons lsym k)))

(define apply-continuation
  (lambda (k v)
    (if (null? k)
      v
      (let ((lsym (car k))
            (k1 (cdr k)))
        (apply-continuation k1
          (cons (car lsym) v))))))
```


Another alternate representation

Like the previous, but store only (car lsym), not lsym.

```
(make-identity) = '()
(make-rem1 lsym k) = `(,(car lsym) . ,k)
```

Here ` introduces a pattern; , marks a pattern expression, so `(,(car lsym) . ,k) denotes a cons-cell whose car is the value of the car of lsym and whose cdr is the value of k. This is backquote notation; see Dybvig.

```
(define make-identity (lambda () '()))

(define make-rem1
  (lambda (lsym k)
    (cons (car lsym) k)))

(define apply-continuation
  (lambda (k v)
    (if (null? k)
      v
      (let ((car-of-lsym (car k))
            (k1 (cdr k)))
        (apply-continuation k1
          (cons car-of-lsym v))))))
```

Another case study: subst

```
(define subst-cps
  (lambda (old new s k)
    (if (pair? s)
      (subst-cps old new (car s)
        (lambda (v1)
          (subst-cps old new (cdr s)
            (lambda (v2) (k (cons v1 v2))))))
      (if (eq? s old) (k new) (k s)))))

(define subst
  (lambda (old new s)
    (subst-cps old new s (lambda (x) x))))
```

```
⟨cont⟩ ::= (lambda (x) x)
       ::= (lambda (v1)
             (subst-cps old new (cdr s)
               (lambda (v2) (⟨cont⟩ (cons v1 v2)))))
       ::= (lambda (v2) (⟨cont⟩ (cons v1 v2)))
```


Datatype of subst-continuations

```
(make-identity) = ⌈(lambda (x) x)⌉

(make-subst1 old new s ⌈k⌉) = ⌈ (lambda (v1)
               (subst-cps old new (cdr s)
                 (lambda (v2) (k (cons v1 v2)))))⌉

(make-subst2 v1 ⌈k⌉) = ⌈(lambda (v2) (k (cons v1 v2)))⌉

(apply-continuation ⌈k⌉ v) = (k v)
```


Representation-Independent Code

```
(define subst-cps
  (lambda (old new s k)            ; k is a representation
    (if (pair? s)
      (subst-cps old new (car s)
        (make-subst1 old new s k))
      (if (eq? s old)
        (apply-continuation k new)
        (apply-continuation k s)))))

(define subst
  (lambda (old new s)
    (subst-cps old new s (make-identity))))
```


Procedural Representation

```
(define make-identity
  (lambda ()
    (lambda (x) x)))

(define make-subst1
  (lambda (old new s k)
    (lambda (v1)
      (subst-cps old new (cdr s) (make-subst2 v1 k)))))

(define make-subst2
  (lambda (v1 k)
    (lambda (v2) (apply-continuation k (cons v1 v2)))))

(define apply-continuation
  (lambda (k v)
    (k v)))
```

```
(define-record identity-record ())
(define-record subst1-record (old new s k))
(define-record subst2-record (v1 k))

(define apply-continuation
  (lambda (k x)
    (record-case k
      (identity-record () x)
      (subst1-record (old new s k)
        (let ((v1 x))
          (subst-cps old new (cdr s)
            (make-subst2 v1 k))))
      (subst2-record (v1 k)
        (let ((v2 x))
          (apply-continuation k (cons v1 v2))))
      (else (report-bad-continuation k)))))
```