CS 565 September 21, 1997
Jens Palsberg Readings: EOPL Ch. 8.1-8.5, 9.1-9.2, 10.1-10.2

Lecture Note 6:

A translation of Scheme programs to simple Java programs via CPS-conversion

Contents

1 Overview **2**

2 From Scheme to Tail Form **3**

3 From Tail Form to First-Order Form **17**

3.1 Continuations as a Datatype 18
3.2 Representation-Independent Code 18
3.3 Procedural Representation of Continuations . . . 19
3.4 Abstract Syntax Tree Representation 20
3.5 Nifty Tree Representation 21
3.6 Even Niftier Tree Representation 22
3.7 List Representation of Continuations 27
3.8 Stack Representation of Continuations 32
3.9 Clever Representation of Continuations 40

4 From First-Order Form to Imperative Form **45**

1 Overview

This note shows how to translate a Scheme program into efficient low-level code. The translation is in a number of steps.

1. From a Scheme program to a Scheme program in *tail form*. The idea is to transform the program such that functions never return. This is done by introducing *continuations*.

2. From a Scheme program in tail-form to a Scheme program in *first-order form*. The idea is to transform the program such that all functions are defined at the top level. This is done by representing the continuations as first-order data structures.

3. From a Scheme program in first-order form to a Scheme program in *imperative form*. The idea is to transform the program such that functions take no arguments. This is done by passing the arguments in a fixed number of global variables. It is possible to avoid using a stack because the program is in tail form.

4. From a Scheme program in imperative form to Java, C, machine code, etc. A Scheme program in imperative form is so close to machine code that the translation is simple.

This note covers the first three steps. The fourth step was exemplified in Homework 4.

2 From Scheme to Tail Form

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: *SimpleExp*, for expressions that can go on the right-hand sides of assignments, and *TailFormExp*, for the programs themselves.

TailFormExp	::=	*SimpleExp*
	|	(*SimpleExp* $SimpleExp_1 \ldots SimpleExp_n$)
	|	(`if` *SimpleExp TailFormExp TailFormExp*)
	|	(`set!` *Identifier SimpleExp*)
	|	(`begin` $SimpleExp_1 \ldots SimpleExp_n$ *TailFormExp*)
SimpleExp	::=	*Identifier*
	|	*Constant*
	|	(*PrimitiveOperation* $SimpleExp_1 \ldots SimpleExp_n$)
	|	(`lambda` ($Identifier_1 \ldots Identifier_n$) *TailFormExp*)

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.

A more complete grammar:

TailFormExp	::=	(`cond` (*SimpleExp TailFormExp*) ...)
	|	(`let` (*SimpleDeclList*) *TailFormExp*)
	|	(`letrec` (*SimpleDeclList*) *TailFormExp*)
SimpleDeclList	::=	$SimpleDecl_1 \ldots SimpleDecl_n$
SimpleDecl	::=	(*Identifier SimpleExp*)

Note: our *SimpleExp* corresponds to what the book calls "simple *and* tail-form".

Examples

```
(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))))
```

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)))))))
```

The Transformation Rules

```
(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 ...)))
```

3 From Tail Form to First-Order Form

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))))
```

3.1 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)` $= \lceil$`(lambda (v) v)`$\rceil$
`(make-rem1 lsym` $\lceil k \rceil$`)` $= \lceil$`(lambda (v) (`k `(cons (car lsym) v)))`$\rceil$
`(apply-continuation` $\lceil k \rceil$ `v)` $=$ `(`k `v)`

3.2 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))))
```

3.3 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)))
```

3.4 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 ())
(define-record rem1-record (lsym k))


(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)))
```

3.5 Nifty Tree Representation

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

$$\texttt{(make-identity)} = \texttt{'()}$$
$$\texttt{(make-rem1 } lsym\ k\texttt{)} = (lsym\ .\ k)$$

```
(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))))))
```

3.6 Even Niftier Tree 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))))
```

Continuation ::= `(lambda (x) x)`

::= `(lambda (v1)`
`(subst-cps old new (cdr s)`
`(lambda (v2) (`*Continuation* `(cons v1 v2)))))`

::= `(lambda (v2) (`*Continuation* `(cons v1 v2)))`

Datatype of `subst`-Continuations

$$\texttt{(make-identity)} = \lceil \texttt{(lambda (x) x)} \rceil$$

$$\texttt{(make-subst1 old new s } \lceil k \rceil \texttt{)} = \lceil \texttt{ (lambda (v1)}$$
$$\texttt{(subst-cps old new (cdr s)}$$
$$\texttt{(lambda (v2) (}k\texttt{ (cons v1 v2)))))} \rceil$$

$$\texttt{(make-subst2 v1 } \lceil k \rceil \texttt{)} = \lceil \texttt{(lambda (v2) (}k\texttt{ (cons v1 v2)))} \rceil$$

$$\texttt{(apply-continuation } \lceil k \rceil \texttt{ v)} = \texttt{(}k\texttt{ v)}$$

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)))
```

Abstract Syntax Tree Representation

```
(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)))))
```

3.7 List Representation of Continuations

List of Frames Representation

```
(make-identity) = '((lastframe))

(make-subst1 old new s k) = `((frame1 ,old ,new ,s) . ,k)

(make-subst2 v1 k) = `((frame2 ,v1) . ,k)
```

Typical continuation looks like a list of frames:

```
((frame1 <o> <n> <s_1>)
 (frame2 <v_1>)
 (frame2 <v_2>)
 ...
 (lastframe))
```

```
(define-record lastframe ())
(define-record frame1 (old new s))
(define-record frame2 (v1))

(define make-identity
  (lambda ()
    (cons (make-lastframe) '())))

(define make-subst1
  (lambda (old new s k)
    (cons (make-frame1 old new s) k)))

(define make-subst2
  (lambda (v1 k)
    (cons (make-frame2 v1) k)))

(define apply-continuation
  (lambda (k x)
    (let ((frame (car k))(k (cdr k)))
      (record-case frame
        (frame1 (old new s)
          (let ((v1 x))
            (subst-cps old new (cdr s) (make-subst2 v1 k))))
        (frame2 (v1)
          (let ((v2 x))
            (apply-continuation k (cons v1 v2))))
        (lastframe () x)
        (else (report-bad-continuation k))))))
```

List of Frames without Endmarker

Same as list-of-frames, except

```
(make-identity) = '()

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

(define apply-continuation
  (lambda (k x)
    (if (null? k)  ; annoying to have to test this first!
      x
      (let ((frame (car k))(k (cdr k)))
        (record-case frame
          (frame1 (old new s)
            (let ((v1 x))
              (subst-cps old new (cdr s)
                (make-subst2 v1 k))))
          (frame2 (v1)
            (let ((v2 x))
              (apply-continuation k
                (cons v1 v2))))
          (else (report-bad-continuation k)))))))
```

Unframed List

Like list-of-frames, but without interior sublists

```
(make-identity) = '(lastframe)

(make-subst1 old new s k) = `(frame1 ,old ,new ,s . ,k)

(make-subst2 v1 k) = `(frame2 ,v1 . ,k)
```

Typical continuation looks like a list of items

```
(frame1 <o> <n> <s_1> frame2 <v_1> frame2 <v_2> ... lastframe)
```

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

(define make-subst1
  (lambda (old new s k)
    (cons 'frame1 (cons old (cons new (cons s k))))))

(define make-subst2
  (lambda (v1 k)
    (cons 'frame2 (cons v1 k))))

(define apply-continuation
  (lambda (k x)
    (case (car k)
      ((frame1) (let ((old   (cadr k))
                      (new  (caddr k))
                      (s   (cadddr k))
                      (k   (cddddr k)))
                  (let ((v1 x))
                    (subst-cps old new (cdr s)
                      (make-subst2 v1 k)))))
      ((frame2) (let ((v1 (cadr k))
                      (k  (cddr k)))
                  (let ((v2 x))
                    (apply-continuation k (cons v1 v2)))))
      ((lastframe) x)
      (else (report-bad-continuation k))))))
```

3.8 Stack Representation of Continuations

Will start with unframed-stack representation, and represent the continuation as a real stack: an array and a pointer.

Will represent the stack as a Scheme vector (equivalent to an array: a randomly-accessible mutable data structure).

```
(define make-stack
  (lambda (size)
    (list '*stack -1 (make-vector size))))

(define stack->top cadr)
(define stack->data caddr)

(define stack-ref
  (lambda (stack ptr)
    (vector-ref (stack->data stack) ptr)))

;; works by side-effect
(define stack-set-val!
  (lambda (stack ptr val)
    (vector-set! (stack->data stack) ptr val)))

(define stack-set-top!
  (lambda (stack ptr)
    (set-car! (cdr stack) ptr)))
```

These work by side-effect, but return a pointer to the altered stack. This is convenient for composing stack operations (just like `cons`!)

```
(define stack-push
  (lambda (val stack)
    (let ((ptr (stack->top stack)))
      (stack-set-top! stack (+ ptr 1))
      (stack-set-val! stack (+ ptr 1) val)
      stack)))

(define stack-pop
  (lambda (stack n)
    (stack-set-top! stack (- (stack->top stack) n))
    stack))
```

Now that we are doing pointer arithmetic, some decent debugging tools are a must:

```
(define debug-print
  (lambda (stack)
    (if (eq? (car stack) '*stack)
      (begin
        (printf "sp = ~s~%" (stack->top stack))
        (let loop ((sp (stack->top stack)) (n 10))
          (if (or (< sp 0) (zero? n))
            (printf "~%")
            (begin
              (printf "stack[~s] = ~s~%" sp
                      (stack-ref stack sp))
              (loop (- sp 1) (- n 1))))))))
      (printf "Warning: bad stack!~%stack = ~s~%" stack))))
```

Now we can alter the unframed-stack representation to use this array representation. Note that we are always using the *same* array, but we are modifying it destructively. This works just so long as we have only one continuation active at any time. If not, this will fail dramatically.

```
(define make-identity
  (lambda ()
    (let ((stack (make-stack 200)))    ; use some suitable size
      (stack-push 'lastframe stack))))

(define make-subst1
  (lambda (old new s k)
    (stack-push 'frame1        ; just change cons to stack-push
      (stack-push old
        (stack-push new
          (stack-push s k))))))

(define make-subst2
  (lambda (v1 k)
    (stack-push 'frame2
      (stack-push v1 k))))
```

```
(define apply-continuation
  (lambda (k x)                          ; k is a stack
    (debug-print k)
    (let ((sp (stack->top k)))
      (case (stack-ref k sp)
        ((frame1) (let ((old   (stack-ref k (- sp 1)))
                        (new   (stack-ref k (- sp 2)))
                        (s     (stack-ref k (- sp 3)))
                        (k     (stack-pop k 4)))
                    (let ((v1 x))
                      (subst-cps old new (cdr s)
                        (make-subst2 v1 k)))))
        ((frame2) (let ((v1 (stack-ref k (- sp 1)))
                        (k  (stack-pop k 2)))
                    (let ((v2 x))
                      (apply-continuation k
                        (cons v1 v2)))))
        ((lastframe) x)
        (else (report-bad-continuation k))))))
```

Testing

Here is a run, with a snapshot at every invocation of `apply-continuation`.

```
> (subst 'foo 'bar '((a foo) foo))
sp = 8
stack[8] = frame1
stack[7] = foo
stack[6] = bar
stack[5] = (a foo)
stack[4] = frame1
stack[3] = foo
stack[2] = bar
stack[1] = ((a foo) foo)
stack[0] = lastframe

sp = 10
stack[10] = frame1
stack[9] = foo
stack[8] = bar
stack[7] = (foo)
stack[6] = frame2
stack[5] = a
stack[4] = frame1
stack[3] = foo
stack[2] = bar
stack[1] = ((a foo) foo)
```

```
sp = 8
stack[8] = frame2
stack[7] = bar
stack[6] = frame2
stack[5] = a
stack[4] = frame1
stack[3] = foo
stack[2] = bar
stack[1] = ((a foo) foo)
stack[0] = lastframe

sp = 6
stack[6] = frame2
stack[5] = a
stack[4] = frame1
stack[3] = foo
stack[2] = bar
stack[1] = ((a foo) foo)
stack[0] = lastframe

sp = 4
stack[4] = frame1
stack[3] = foo
stack[2] = bar
stack[1] = ((a foo) foo)
stack[0] = lastframe
```

```
sp = 6
stack[6] = frame1
stack[5] = foo
stack[4] = bar
stack[3] = (foo)
stack[2] = frame2
stack[1] = (a bar)
stack[0] = lastframe

sp = 4
stack[4] = frame2
stack[3] = bar
stack[2] = frame2
stack[1] = (a bar)
stack[0] = lastframe

sp = 2
stack[2] = frame2
stack[1] = (a bar)
stack[0] = lastframe

sp = 0
stack[0] = lastframe

((a bar) bar)
```

3.9 Clever Representation of Continuations

Original:

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

CPS version:

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

(define fact
  (lambda (n)
    (fact-cps (lambda (v) v))))
```

Continuations for `fact`:

$$\begin{aligned} \textit{Continuation} &::= \texttt{(lambda (v) v)} \\ &::= \texttt{(lambda (v) (}\textit{Continuation}\ \texttt{(* n v)))} \end{aligned}$$

$$\texttt{(make-identity)} = \lceil \texttt{(lambda (v) v)} \rceil$$
$$\texttt{(make-fact1 n} \lceil k \rceil \texttt{)} = \lceil \texttt{(lambda (v) (}k\ \texttt{(* n v)))} \rceil$$

```
(define fact-cps
  (lambda (n k)
    (if (zero? n)
      (apply-cont k 1)
      (fact-cps (- n 1) (make-fact1 n k)))))

(define fact
  (lambda (n)
    (fact-cps n (make-identity))))
```

Claim: Every `fact` continuation is of the form

```
(lambda (v) (* p v))
```

for some `p`.

```
(lambda (v) v) = (lambda (v) (* 1 v))

If k = (lambda (v) (* p v))
then
(make-fact1 n k) = (lambda (v) (k (* n v)))
                 = (lambda (v) ((lambda (w) (* p w))
                                (* n v)))
                 = (lambda (v) (* p (* n v)))
                 = (lambda (v) (* (* p n) v))
```

So we can represent `(lambda (v) (* p v))` by just the number `p`. (Neat!)

```
(define make-identity
  (lambda () 1))

(define make-fact1
  (lambda (n p)      ; p       repr (lambda (v) (* p v))
    (* n p)))        ; (* n p) repr (lambda (v) (* (* n p) v))

(define apply-cont
  (lambda (p v)
    (* p v)))
```

```
(define fact-cps
  (lambda (n k)
    (if (zero? n)
      (apply-cont k 1)
      (fact-cps (- n 1) (make-fact1 n k)))))

(define fact
  (lambda (n)
    (fact-cps n (make-identity))))
```

Inlining these definitions in `fact-cps` we get:

```
(define fact-cps
  (lambda (n k)
    (if (zero? n)
        k     ; = (* k 1) = (apply-cont k 1)
        (fact-cps (- n 1) (* n k)))))

(define fact
  (lambda (n)
    (fact-cps n 1)
```

Wow! It's `fact-iter`! You won't be able to do things this dramatic too often, but you can make sense of many mysterious things as clever representations of continuations.

4 From First-Order Form to Imperative Form

Now we have tail-form, first-order (no first-class functions, just data structures) program. Want to convert it to flowchart. Will use idea of lambda-variables as registers: Replace

```
(define foo
  (lambda (x y) ..body..))
...
(foo E1 E2)
```

by

```
foo: ..body..   % take input from registers x and y


x := E1;        % be careful about overwriting registers here!
y := E2;
goto foo;
```

Will do this using Scheme instead of pseudocode.

```
;;; cps version, with representation-independent driver
;;; uses abstract syntax tree representation

(define-record identity ())
(define-record rem1 (lsym k))

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

(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 apply-continuation
  (lambda (k v)
    (record-case k
      (identity () v)
      (rem1 (lsym k1)
        (apply-continuation k1 (cons (car lsym) v))))))
```

```
;;; register machine version

(define remove-all
  (lambda (a lsym)
    (let ((a a) (lsym lsym)
          (k (make-identity))
          (v '*unbound*))
      (letrec
        ((remove-all-cps
           (lambda ()
             (cond ((null? lsym)
                    (set! v '())
                    (apply-continuation))
                   ((eq? a (car lsym))
                    (set! lsym (cdr lsym))
                    (remove-all-cps))
                   (else
                    (set! k (make-rem1 lsym k))
                    (set! lsym (cdr lsym))
                    (remove-all-cps)))))
         (apply-continuation
           (lambda ()
             (record-case k
               (identity () v)
               (rem1 (lsym k1)
                     (set! k k1)
                     (set! v (cons (car lsym) v))
                     (apply-continuation))))))
        (remove-all-cps)))))
```

Now let's do it for `subst`. Let's take the array representation:

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

(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 make-identity
  (lambda ()
    (let ((stack (make-stack 200)))  ; use some suitable size
      (stack-push 'lastframe stack))))

(define make-subst1
  (lambda (old new s k)
    (stack-push 'frame1      ; just change cons to stack-push
      (stack-push old
        (stack-push new
          (stack-push s k))))))

(define make-subst2
  (lambda (v1 k)
    (stack-push 'frame2 (stack-push v1 k))))
```

```
(define apply-continuation
  (lambda (k x)                          ; k is a stack
    (let ((sp (stack->top k))
          (ins (stack-ref k sp)))
      (case ins
        ((frame1) (let ((old   (stack-ref k (- sp 1)))
                        (new   (stack-ref k (- sp 2)))
                        (s     (stack-ref k (- sp 3)))
                        (k     (stack-pop! k 4)))
                    (let ((v1 x))
                      (subst-cps old new (cdr s)
                        (make-subst2 v1 k)))))
        ((frame2) (let ((v1 (stack-ref k (- sp 1)))
                        (k  (stack-pop! k 2)))
                    (let ((v2 x))
                      (apply-continuation k
                        (cons v1 v2)))))
        ((lastframe) x)
        (else (report-bad-continuation k))))))
```

```
(define subst
  (lambda (old-arg new-arg s-arg)
    ;; (subst-cps old new (make-identity))
    (set! old old-arg)
    (set! new new-arg)
    (set! k (make-identity))
    (subst-cps)))

(define subst-cps
  (lambda ()                  ; uses old, new, s, k
    (if (pair? s)
      ;; (subst-cps old new (car s)
      ;;   (make-subst1 old new s k))
      (begin
        (set! k (make-subst1 old new s k))
        (set! s (car s))
        (subst-cps))
      (if (eq? s old)
        (begin
          ;; (apply-continuation k new)
          (set! x new)
          (apply-continuation))
        (begin
          ;; (apply-continuation k s)
          (set! x s)
          (apply-continuation))))))
```

```
(define apply-continuation
  (lambda ()                                ; uses k, x
    (let ((sp (stack->top k))
          (ins (stack-ref k sp)))
      (case ins
        ((frame1) (let
                    ;; need different local names
                    ((old1   (stack-ref k (- sp 1)))
                     (new1   (stack-ref k (- sp 2)))
                     (s1     (stack-ref k (- sp 3)))
                     (k1     (stack-pop! k 4)))
                    (let ((v1 x))
                      ;;(subst-cps old new (cdr s)
                      ;; (make-subst2 v1 k))
                      (set! old old1)
                      (set! new new1)
                      (set! s (cdr s1))
                      (set! k (make-subst2 v1 k1))
                      (subst-cps))
        ((frame2) (let
                    ((v1 (stack-ref k (- sp 1)))
                     (k1  (stack-pop! k 2)))
                    (let ((v2 x))
                      ;; (apply-continuation k
                      ;;  (cons v1 v2))
                      (set! x (cons v1 v2))
                      (set! k k1)
                      (apply-continuation)))
        ((lastframe) x)
        (else (report-bad-continuation k))))))
```