Lecture Note 6:

More Continuations

From Tail Form to Imperative Form

We've gone from a general program to tail form (by the CPS transformation), and from tail form to first-order (iterative) form (by converting continuations to data structures). Our goal is to get to imperative (flowchart) form. But first, let's explore some niftier representations of continuations.

```
(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))))
```

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)))))
```


Stack representation

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`. Probably you'd want to do other things as well, like print the other variables, put in a similar trace at the top of `subst-cps`, and provide a convenient way of switching these traces on and off.

```
> (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)
```


Using clever representations

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`:

$\langle\text{cont}\rangle$ ::= `(lambda (v) v)`
::= `(lambda (v) (`$\langle\text{cont}\rangle$` (* n v)))`

`(make-identity)` = $\lceil$`(lambda (v) v)`$\rceil$
`(make-fact1 n `$\lceil k \rceil$`)` = $\lceil$`(lambda (v) (`k` (* 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. We'll do more of this next time when we deal with stack languages.

Converting 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))))))
```

```scheme
;;; register machine version

(define remove-all
  (lambda (a lsym)
    (let ((a a) (lsym lsym)
          (k (make-identity))
          (v '*unbound*))
      (letrec
        ((remove-all-cps
           (lambda ()
             ;; (printf "remove-all: a = ~s l = ~s k = ~s~%" a l k)
             (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 ()
             ;; (printf "apply-cont: k = ~s v = ~s~%" k v)
             (record-case k
               (identity () v)
               (rem1 (lsym k1)
                     (set! k k1)
                     (set! v (cons (car lsym) v))
                     (apply-continuation))))))
        (remove-all-cps)))))
```


Doing it in C

Now let's do it in C. Will allocate all data structures (including continuations) in the heap.

```c
#include <stdlib.h>

/* data structures for linear lists */

typedef int DATA;

struct list_node {
  DATA car;
  struct list_node *cdr;
};

typedef struct list_node *list;

#define NULL 0

list cons(DATA a, list l) {        /* ANSI C */
  list new;
  new = (list) malloc(sizeof(struct list_node)); /* cast! */
  new->car = a;
  new->cdr = l;
  return(new);
}
```



```c
/* data structures for continuations */

struct cont_node {
  list             lsym_field;
  struct cont_node *k_field; /* null repr the identity */
};

typedef struct cont_node *cont;

#define IDENTITY 0

cont make_rem1(list lsym, cont k) {
  cont k1;
  k1 = (cont) malloc(sizeof(struct cont_node));
  k1->lsym_field = lsym;
  k1->k_field = k;
  return(k1);
}
```



```c
/* finally, the working part */

list remove_all (DATA a, list lsym) {

  list v;                /* register for returning values */
  cont k = {0};                /* initially NULL, I think */

remove_all_cps:
  if (lsym == NULL)        /* if (null? lsym) */
    {v = NULL;
     goto apply_continuation;}
  else if (a == lsym->car)
    {lsym = lsym->cdr;
     goto remove_all_cps;}
  else {k = make_rem1(lsym, k);
        lsym = lsym->cdr;
        goto remove_all_cps;};

apply_continuation:
  if (k == NULL)        /* is k the identity continuation? */
    {return(v);}
  else
    {v = cons(k->lsym_field->car, v);
     k = k->k_field;
     goto apply_continuation;};

};
```

```
/* test scaffold */

void print_list(list l) {
  printf("(");
  while (l != NULL) {
    printf("%d", l->car);
    l = l->cdr;
    if (l != NULL) printf(" ");
  };
  printf(")");
}

list array_to_list(int data[]) {
  list l = NULL;
  int n = 0;
  int i;
  while (data[i] != -1) {i++;};
  i--;
  while (i >= 0) {l = cons(data[i],l); i--;};
  return(l);
}

int a[] = {1, 2, 3, 4, 2, 5, 7, -1};
list l;
```



```
main(){
  printf("starting up...\n");
  l = array_to_list(a);
  printf("input is  ");
  print_list(l);
  printf("\n");
  l = remove_all(2, l);
  printf("output is ");
  print_list(l);
  printf("\n");
};

/* and the output is....

[112] gcc -o remove-all remove-all.c
[113] remove-all
starting up...
input is  (1 2 3 4 2 5 7)
output is (1 3 4 5 7)

*/
```

Are we having fun yet?

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))))))
```


Exercises: (1) `old` and `new` never change, so they don't need to pushed and popped off the stack. Change the program to incorporate this observation. (2) Rewrite the program so that instead of having a single register `k`, we have two registers `vec` and `sp`. Then unwind the calls to `make-subst1` and `make-subst2`, so that all the `vector-set!`'s are explicit. This will make explicit the patterns of sharing and side-effecting used in this program.