

## CS6848 - Principles of Programming Languages

## Principles of Programming Languages

V. Krishna Nandivada

IIT Madras

- Type systems
  - Simply typed lambda calculus
  - Recursive types
  - polymorphic types
  - Semantics
  - Type inference
  - Type soundness
  - Typed Assembly Language



## Continuation Passing Style

**Goal: translate a scheme program into efficient code.** Four steps.

- 1 From a Scheme program to a Scheme program in tail form.
  - transform the program such that functions never return.
  - done by introducing continuations.
  - Program in tail form needs no stack (except for passing arguments).
- 2 From a Scheme program in tail-form to a Scheme program in first-order form.
  - transform the program such that all functions are defined at the top level.
  - Represent the continuations as first-order data structures.
- 3 From a Scheme program in first-order form to a Scheme program in imperative form.
  - transform the program such that functions take no arguments.
  - pass the arguments in a fixed number of global variables.
  - need no stack at all as - no arguments are passed.
- 4 From a Scheme program in imperative form to C, machine code, etc.
  - A Scheme program in imperative form is close to machine code.
  - key task is to replace each function call with a jump.



## recursive procedures

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

(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))))
         = (* 4 (* 3 (* 2 1)))
         = (* 4 (* 3 2))
         = (* 4 6)
         = 24
```

- each call of `fact` is made with a promise that the value returned will be multiplied by the value of `n` at the time of the call; and
- thus `fact` is invoked in larger and larger control contexts as the calculation proceeds.
- the contexts are passed in the stack.



## java

```
int fact(int n) {
    int a=1;
    while (n!=0) {
        a=n*a;
        n=n-1;
    }
    return a;
}
```

## scheme

```
(define fact-iter
  (lambda (n)
    (fact-iter-acc n 1)))

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



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

- fact-iter-acc is always invoked in the same context (in this case, no context at all).
- when fact-iter-acc calls itself, it does so at the "tail end" of a call to fact-iter-acc. that is, no promise is made to do anything with the returned value other than return it as the result of the call to fact-iter-acc.



## example of tail form

```
(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
```



## grammar for scheme in cps

```
simpleExp ::= identifier
          | constant
          | ( primitiveOperation simpleExp_1 ... simpleExp_n )
          | (set! identifier simpleExp )
          | (lambda ( identifier_1 ... identifier_n ) tailFormExp )

tailFormExp ::= simpleExp
             | ( simpleExp simpleExp_1 ... simpleExp_n ) ; application
             | (if simpleExp tailFormExp tailFormExp ) ; conditional
             | (begin simpleExp_1 ... simpleExp_n tailFormExp ) ; block
```

## Two position in the program: simple and tail-form

- 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.



```
tailFormExp ::= (cond ( simpleExp tailFormexp ) ...)
              | (letrec ( simpleDeclList ) tailFormExp )
              | (let ( simpleDeclList ) tailFormExp )
simpleDeclList ::= simpleDecl_1 ...simpleDecl_n
simpleDecl ::= ( identifier simpleExp )
```

simpleexp corresponds to th “simple and tail form” in the text book.



## tail form or not

```
(car x) ; also simple
(car (cdr x)) ; also simple
(car (f x))
(f (car x))
(lambda (v) (k (+ v 1)))
(lambda (v) (+ (k v) 1))
(if (zero? x) (f (+ x 1)) (g (- x 1)))
(if (zero? x) (+ (f x) 1) (g (- x 1)))
(if (p x) (f (+ x 1)) (g (- x 1)))
(if (p (car x)) (f (+ x 1)) (g (- x 1)))
(lambda (x)
  (if (zero? x) 0 (+ (f (- x 1)) 1)))
(lambda (x)
  (if (zero? x) 0 (f (- x 1) (lambda (v) (k (+ v 1))))))
(lambda (n a) ; tail form
  (if (zero? n) a (fact-iter-acc (- n 1) (* n a))))
```



# Sample tail form conversion

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

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

(fact-cps n k) computes (k (fact n)) for any k!

Note: k is a one argument lambda expression that represent the context.



# fact-cps correctness

By induction on n that,  $(k (fact n)) = (fact-cps n k)$   
 For  $n = 0$ ,  $(k (fact 0)) = (k 1) = (fact-cps 0 k)$ . If  $n > 0$

```
(k (fact n))
= (k (* n (fact (- n 1)))) ; definition of fact

= ((lmabda (v) (k (*n v)))
  (fact (-n 1))) ; beta conversion

= (fact-cps (-n 1)
  (lambda (v) (k (* n v)))) ; induction hypothesis

= (fact-cps n k)
```



```

(fact-cps 4 k)
= (fact-cps 3 (lambda (v) (k (* 4 v))))
...
= ((lambda (v) (k (* 4 (* 3 (* 2 (* 1 v)))))) 1)
= (k (* 4 (* 3 (* 2 (* 1 1)))))
= (k 24)

```

hint: set `k = lambda (x) x` to get the original value



```

(define foo
  (lambda (x y) -----))
==>
(define foo-cps
  (lambda (x y k) (k -----)))

(k (foo a (- n 1)))
==>
(foo-cps a (- n 1) k)

(k (----- (foo a (- n 1)) -----))
==>
(foo-cps a (- n 1) (lambda (v) (k (----- v -----))))

(k (if (simpExp) e1 e2))
==>
(if (simpExp) (k e1) (k e2))

```



## Example transformation

```

(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)))))))

```



## Example transformation (cont.)

```

(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))))))

```



## Example transformation (cont.)

```
(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)))))))
==>
(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)))))))
```



## Example transformation 2

```
(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)) ; choice 1
                (subst old new (cdr s)))) ; choice 2.
        (if (eq? s old) (k new) (k s)))))
```



## Example transformation 2, by choice 2 (cdr)

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



## Example transformation 2, by choice 1 (car)

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



## Convert to tail form

```
(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))))
```



# Recap

- Idea of CPS
- Step by step approach to convert scheme to cps.
- Algorithm to convert Scheme programs to Tail form.

## What you should be able to answer (necessary not sufficient)

- Given a scheme program convert it to tail form.



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



# Getting the original function behavior from the cps version

```
(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))))
```



## Getting rid of the 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))))
```

**Goal:** Remove the higher order functions and instead replace them with procedure calls.



## Getting rid of the higher order functions

```
(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))))

(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)))
```

**Goal:** Remove the higher order functions and instead replace them with procedure calls.



## Representation for continuations

- For each continuation and the application of the continuation create a new procedure.
- Grammar for continuations for `remove-all-cps`:
 

```
Cont ::= (lambda (v) v)
      ::= (lambda (v) (Cont (cons (car lsym) v)))
```
- Thus we need to specify three functions<sup>1</sup>:
  - `(make-identity) = [ lambda (v) v ]`
  - `(make-rem1 lsym [k]) = [lambda (v) (k (cons (car lsym) v))]`
  - `(apply-continuation [k] v) = (k v)`



## Representing continuations as records

- Represent each of the continuations as a list (influenced by the underlying AST).

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

- 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))
```

- The `apply` function becomes more involved.

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



## Slim representation

- Only one possible tag `rem1`. Drop it!

```
(make-identity) = '()
(make-rem1 v k) = (v . k)

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

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

(define apply-continuation
  (lambda (k v)
    (if (null? k)
        v ; this was record case (identity-record)
        (let ((lsym (car k)) ; case rem1-record
              (k1 (cdr k) )
              (apply-continuation k1 (cons (car lsym) v))))))
```



## Example 2

```
(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))))
```



## subst With first-order functions (representation independent)

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



## subst With first-order functions (contd.) - 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)))
```





## subst With first-order functions (contd.) - AST 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 (error "bad continuation")))))
```



## Example 3

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))))
```



## First order continuations

```
(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))))

(define make-fact1 (lambda (n k) (lambda (v) (k (* n v)))))

(define make-identity ...)
```



## fact-cps is same as fact-iter!

- **Observe:** Every `fact` continuation is of the form `(lambda (v) (* p v))`, for some integer `p`.
- `(lambda (v) v) = (lambda (v) (* 1 v))`
- `(make-fact1 n k) = ?`
- **Represent** `(lambda (v) (* p v))` by `p`.
- **Definition for** `make-identity`, `make-fact1` and `apply-cont`.
- **Substitute the new definitions (aka inline!). Voila!**



## From first-order form to imperative

- We have tail form, first order (no first-class functions) program.
- Each lambda variable (formal and actual parameters) can go via globals.
- `(define foo (lambda (x y) ... body ... ))`

`(foo e1 e2)`

- Imperative form:

`foo: ... body ...`

`...`

`x = e1; y = e2; //Careful:don't overwrite globals.`

`goto foo;`



## Example transformation

Original  $\Rightarrow$

```
(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 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 ...
            (remove-all-cps))))))
```



## apply-continuation for remove-all-cps

Original  $\Rightarrow$

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

```
(define remove-all
  ...
  (apply-continuation
    (lambda ()
      (record-case k
        (identity () v)
        (rem1 (lsym k1)
          (set! k k1)
          (set! v (cons (car lsym) v))
          (apply-continuation))))))
```



## Useless assignment

Transform the `subst-cps` code from first-order to imperative form.



- Algorithm to convert programs in tail form to first-order form.
- Algorithm to convert programs in first-order form to imperative form.

**What you should be able to answer (necessary not sufficient)**

- Given a scheme program convert it to imperative form.

