

Computer Science I

Professor Tom Ellman
Lecture 18

Procedures as Return Values

- A theoretical curiosity!
- A mechanism for avoiding redundant computation.
- An elegant tool for drawing recursive pictures

General Form of Definition for a Procedure Returning a Procedure

```
(define <Name>
  (lambda <Args1>
    (lambda <Args2> <body> ) ) )
```

↓
Expression defining the return value of the function <Name>.

↓
Expression defining the function <Name>.

Sum and Product of Functions

```
(define function-sum
  (lambda (f1 f2) (lambda (x) (+ (f1 x) (f2 x)))))

(define function-product
  (lambda (f1 f2) (lambda (x) (* (f1 x) (f2 x)))))
```

Sum of Functions

```
Welcome to Dr. Scheme.
> (define square (lambda (x) (* x x)))
> (define cube (lambda (x) (* x x x)))
> (define square-plus-cube (function-sum square cube))
> (square-plus-cube 2)
12
> (+ (* 2 2) (* 2 2 2))
12
```

Predicate Conjunction, Disjunction and Negation

```
(define predicate-conjunction
  (lambda (p1 p2) (lambda (x) (and (p1 x) (p2 x)))))

(define predicate-disjunction
  (lambda (p1 p2) (lambda (x) (or (p1 x) (p2 x)))))

(define predicate-negation
  (lambda (p) (lambda (x) (not (p x)))))
```

Disjunction of Predicates

```
Welcome to Dr. Scheme.  
> (define non-negative? (predicate-disjunction zero? positive?))  
> (non-negative? 0)  
#t  
> (non-negative? 7)  
#t  
> (non-negative? -7)  
#f
```

MAP-FUNCTION

Converts a function $\langle\text{Type-A}\rangle \rightarrow \langle\text{Type-B}\rangle$ into a function $\langle\text{List-of}(\text{Type-A})\rangle \rightarrow \langle\text{List-of}(\text{Type-B})\rangle$:

```
(define map-function (lambda (f) (lambda (lst) (map f lst))))
```

Double a List of Lumbers

```
Welcome to Dr. Scheme.  
> (define double (lambda (x) (* x 2)))  
> (define double-all (map-function double))  
> (double-all '(1 2 3 4 5))  
(2 4 6 8 10)  
> (map double '(1 2 3 4 5))  
(2 4 6 8 10)
```

FLAT-REDUCE-FUNCTION

Converts a function $\langle\text{Type}\rangle \times \langle\text{Type}\rangle \rightarrow \langle\text{Type}\rangle$ into a function $\langle\text{List-of}(\text{Type})\rangle \rightarrow \langle\text{Type}\rangle$:

```
(define flat-reduce-function  
  (lambda (f id)  
    (lambda (lst) (reduce-flat f lst id))))
```

DEEP-REDUCE-FUNCTION

Converts function $\langle\text{Type}\rangle \times \langle\text{Type}\rangle \rightarrow \langle\text{Type}\rangle$ into a function $\langle\text{Tree-of}(\text{Type})\rangle \rightarrow \langle\text{Type}\rangle$:

```
(define deep-reduce-function  
  (lambda (f base-op)  
    (lambda (lst) (reduce-deep f lst base-op))))
```

Currying a Function

- Suppose a function F takes two arguments.
- Then F is equivalent to another function CF :
 $((CF \langle\text{arg1}\rangle) \langle\text{arg2}\rangle) = (F \langle\text{arg1}\rangle \langle\text{arg2}\rangle)$.
- CF is a function of one argument that returns a function of one argument.
- The function CF is called the “Curried” version of F .
- Named for Haskell Curry.
- (He was really into this stuff!)

Currying the AVERAGE Procedure

```
(define curry-binary-function (lambda (f) ...?...))  
(define average (lambda (x y) (/ (+ x y) 2)))
```

Welcome to Dr. Scheme.

```
> (define curried-average (curry-binary-function average))  
> (define average-with-five (curried-average 5))  
> (average-with-five 15)  
10  
> ((curried-average 5) 15)  
10
```

CURRY-BINARY-FUNCTION

```
(define curry-binary-function  
  (lambda (f)  
    (lambda (x)  
      (lambda (y) (f x y)))))
```

UNCURRY-BINARY-FUNCTION

```
(define uncurry-binary-function  
  (lambda (f)  
    (lambda (x y) ((f x) y))))
```

CURRY-NARY-FUNCTION

```
(define curry-nary-function  
  (lambda (f n)  
    (if (= n 1)  
        f  
        (lambda (x)  
          (curry-nary-function (lambda lst (apply f (cons x lst)))  
                                (- n 1))))))
```

UNCURRY-NARY-FUNCTION

```
(define uncurry-nary-function  
  (lambda (f)  
    (lambda lst (apply-curried-function f lst))))  
  
(define apply-curried-function  
  (lambda (f args)  
    (if (null? args)  
        f  
        (apply-curried-function (f (car args)) (cdr args)))))
```

DOUBLY-WEIGHTED-SUMSTER

```
(define doubly-weighted-sumster (lambda (w1st1 w1st2) ...?...))
```

Welcome to Dr. Scheme.

```
> (define w1 (list 1 2 3))  
> (define w2 (list 4 5 6))  
> (define wsum (doubly-weighted-sumster w1 w2))  
> (wsum (list 7 8 9))  
270  
> (+ (* 1 4 7) (* 2 5 8) (* 3 6 9))  
270
```

DOUBLY-WEIGHTED-SUMER

```
(define doubly-weighted-sumer
  (lambda (w1 w2 lst) (apply + (map * lst w1 w2))))
```

First we define a related function that takes all three arguments (W1 W2 LST) at once.

DOUBLY-WEIGHTED-SUMSTER

```
(define doubly-weighted-sumster
  (lambda (w1 w2)
    (lambda (lst) (apply + (map * lst w1 w2)))))
```

Next we modify the definition by moving the third argument LST into the argument list of a nested lambda expression.

DOUBLY-WEIGHTED-SUMSTER

```
(define doubly-weighted-sumster
  (lambda (w1 w2)
    (lambda (lst) (apply + (map * lst w1 w2)))))
```

Isn't this a bit wasteful? After all, each time we use WSUM, we multiply corresponding members of W1 and W2, even though the values of W1 and W2 do not change.

DOUBLY-WEIGHTED-SUMSTER

```
(define doubly-weighted-sumster
  (lambda (w1 w2)
    (let ((temp (map * w1 w2)))
      (lambda (lst) (apply + (map * lst temp))))))
```

In this version, we multiply corresponding members of W1 and W2 once and for all, at the time we create WSUM.

COMPOSE

Combines two unary functions f and g into a new unary function that first applies g to its argument and then applies f to its argument.

```
(define compose
  (lambda (f g) (lambda (arg) (f (g arg)))))
```

COMPOSE-N-TIMES

Takes a unary function f and an integer n and returns a new function that applies f to its argument n times.

```
(define compose-n-times
  (lambda (f n)
    (if (= n 0)
        identity
        (compose f (compose-n-times f (- n 1)))))
```

The APPLY-N-TIMES Procedure

```
(define apply-n-times (lambda (f arg n) ...?...))  
(define increment (lambda (x) (+ x 1)))  
(define double (lambda (x) (* 2 x)))
```

Welcome to Dr. Scheme.

```
> (apply-n-times increment 3 5)  
8  
> (apply-n-times double 5 4)  
80  
> (apply-n-times double 7 0)  
7
```

Definition of APPLY-N-TIMES

```
(define apply-n-times  
  (lambda (f arg n)  
    ((compose-n-times f n) arg)))
```

Definition of APPLY-N-TIMES

```
(define apply-n-times  
  (lambda (f arg n)  
    (if (= n 0)  
        arg  
        (f (apply-n-times f arg (- n 1))))))
```

Multiplication is Repeated Addition

```
(define multiply  
  (lambda (x y)  
    (apply-n-times (lambda (z) (+ x z))  
                   0  
                   y)))
```

Exponentiation is Repeated Multiplication

```
(define exponentiate  
  (lambda (x y)  
    (apply-n-times (lambda (z) (* x z))  
                   1  
                   y)))
```

Generalization

- Multiplication is repeated addition.
- We use APPLY-N-TIMES convert addition into multiplication.
- Exponentiation is repeated multiplication.
- We use APPLY-N-TIMES convert multiplication into exponentiation.
- Can we generalize this conversion process?

EMPOWER

```
(define empower
  (lambda (proc id)
    (lambda (x y)
      (apply-n-times (lambda (z) (proc x z))
                      id
                      y))))
```

Using EMPOWER

```
(define multiply (empower + 0))
(define exponentiate (empower * 1))
(define mystery (empower cons '()))
(define enigma (empower append '()))
```

Using EMPOWER

```
Welcome to Dr. Scheme.
> (mystery 'raven 4)
(raven raven raven raven)
> (mystery 'raven 0)
()
> (enigma '(ying yang) 3)
(ying yang ying yang ying yang)
> (enigma '(ying yang) 0)
()
```