

Higher-Order Functions

cs480 (Prasad)

L156HOF

1

Equivalent Notations

```
(define (f x y) (...body...))  
=  
(define f  
  (lambda (x y)  
    (...body...)  
  )  
)
```

cs480 (Prasad)

L156HOF

2

Function Values

```
(define tag  
  (lambda (t l) (cons t l))  
)
```

```
(tag 'int '(1)) -> (int 1)
```

What characterizes a function?

- Expression in the body of the definition.
- The sequence of formal parameters.

cs480 (Prasad)

L156HOF

3

Anonymous Functions

```
( (lambda (t l) (cons t l))  
  'int '(1) )
```

- Name of the function is “irrelevant”.
- Simplification:

```
(define tag cons)
```

- Assigning function values is similar to assigning primitive values.

(*first-class values*).

cs480 (Prasad)

L156HOF

4

Higher-order Functions

- In Scheme, *function values* can be
 - a) passed to other functions.
 - b) returned from functions.
 - c) stored in data structures.
 - FORTRAN, Ada, etc prohibit a), b), c).
 - Pascal allows only a).
 - LISP/C++/Java support approximations in a round about way.
 - LISP : Meta-programming, C++: Function pointers.
 - Java : via Objects (Closures coming in JDK 7)
 - Scala, Python and C# (delegates) are multi-paradigm languages with support for higher-order functions.
 - ML/Scheme are functional languages that treat function values as first-class citizens.

Implementing subprograms using a stack breaks down here. Heap and garbage collection required.

cs480 (Prasad)

L156HOF

5

Applications

- Abstracting commonly occurring patterns of control. (*factoring*)
- Defining generic functions.
- Instantiating generics.

(*ORTHOGONALITY*)

- Eventually contribute to readability, reliability, and reuse.
- *Library*: collection of higher-order functions.

cs480 (Prasad)

L156HOF

6

Factoring Commonality

<pre>(1 2 ... n) -> (1 4 ... n^2)</pre>	<pre>(a b c) -> ((a) (b) (c))</pre>
<pre>(define (sq1 L) (if (null? L) () (cons (* (car L) (car L)) (sq1 (cdr L)))))</pre>	<pre>(define (brac L) (if (null? L) () (cons (list (car L) (car L)) (brac (cdr L)))))</pre>

cs480 (Prasad)

L156HOF

7

The map function

```
(define (map fn lis)
  (if (null? lis) ()
      (cons (fn (car lis))
            (map fn (cdr lis))
          )
  )
)
(map (lambda(x) (* x x)) '(1 2 3))
(map (lambda(x) (list x)) '(a b c))
```

Built-in map:

```
(map + '(1 2 3) '(4 5 6)) = (5 7 9)
```

cs480 (Prasad)

L156HOF

8

foldr (reduce) and foldl (accumulate)

(Racket :> Intermediate Level)

```
(foldr + 100 (list 2 4 8))  
= 2 + 4 + 8 + 100  
= 114
```

```
(foldr cons '(c) '(a b))  
= (a b c)
```

```
(foldl * 100 (list 2 4 8))  
= 8 * 4 * 2 * 100  
= 6400
```

```
(foldl cons '(c) '(a b))  
= (b a c)
```

cs480 (Prasad)

L156HOF

9

Templates/Generics

```
(define (imax x y)  
  (if (< x y) y x)  
)
```

```
(define (cmax x y)  
  (if (char-<? x y)  
      y x )  
)
```

- Parameterize wrt comparison

- *Generalization*

```
(define (max lt? x y)  
  (if (lt? x y) y x)  
)
```

- *Specialization*

```
(max string-<?  
      "a" "ab")
```

- Passing function values

cs480 (Prasad)

L156HOF

10

Instantiation

```
(define (maxg lt?)  
  (lambda (x y)  
    (max lt? x y) )  
)  
( (maxg >) 4 5 )  
  (* customization at run-time *)
```

- Function generating function.
 - *Closure* construction.
- Higher-order functions in a library may be tailored this way for different use.

cs480 (Prasad)

L156HOF

11

Recursion vs Higher-Order Functions

```
(define-struct cost ($))  
; generates code for make-cost & cost-$  
(define costList  
  (list (make-cost 5) (make-cost 10)  
        (make-cost 25)))  
  
(check-expect (totalCost costList) 40)  
(check-expect (totalCost costList)  
               (totalCostHF costList))
```

cs480 (Prasad)

L156HOF

12

Simple Recursion vs Map-Reduce version

```
(define (totalCost cl)
  (if (null? cl) 0
      [+ (cost-$ (car cl))
          (totalCost (cdr cl))] ) )
(define (totalCostHF cl)
  (foldr + 0 (map cost-$ cl)))
```

- Both foldl and foldr admissible.
- Requires DrRacket : > HtDP Intermediate Level

cs480 (Prasad)

L156HOF

13

Scoping

- Rules governing the association of *use* of a variable with the corresponding *declaration*.

```
proc p (x: int) ;
  proc q;
  var y: int;
  begin [x]:= y end;
begin q end;
```

- Imperative Language : *Local / non-local* variables.
- Functional Language : *Bound / free* variables.

cs480 (Prasad)

L156HOF

14

- **Scoping rules** enable determination of declaration corresponding to a **non-local / free** variable.

• *Alternatives*

- Fixed at function definition time
 - Static / lexical scoping
 - Scheme, Ada, C++, Java, C#, etc
- Fixed at function call time
 - Dynamic scoping
 - Franz LISP, APL, etc.

cs480 (Prasad)

L156HOF

15

Pascal Example

```
proc p;
var z:int;

  proc q;
  begin z := 5 end;

  proc r;
  var z:int;
  begin q end;

  proc s;
  var z:int;
  begin q end;

begin ... end;
```

cs480 (Prasad)

L156HOF

16

Scoping : *z??*

Static

```
r:
  z := 5;
s:
  z := 5;
```

Calls to *q* in *r* and *s*
update the variable *z*
declared in *p*.

Dynamic

```
r:
  z := 5;
s:
  z := 5;
```

Calls to *q* in *r* and *s*
update the variables *z*
and *z* declared in *r*
and *s* respectively.

cs480 (Prasad)

L156HOF

17

Scoping : *functional style*

```
(define y 5)
( (lambda (x) (+ x y)) 3 )
```

- Point of definition = Point of call.

```
(define (f x) (+ x y))
(define (g y) (f y))
(g 16)
```

- Naming context of definition \neq Naming context of call.

cs480 (Prasad)

L156HOF

18

Scoping Rules

• Static Scoping

```
(g y) y <- 16
(f y) y <- 16
(+ x y) x <- 16
      y <- 5
```

21

• Dynamic Scoping

```
(g y) y <- 16
(f y) y <- 16
(+ x y) x <- 16
      y <- 16
      y <- 5
```

32

cs480 (Prasad)

L156HOF

19

Closure

```
(define (addn n)
  (lambda (x) (+ x n) )
)
```

(addn 5) =

(lambda (x) (+ x n))	n <- 5
-----------------------------	--------

Closure = Function Definition +
Creation-time environment
(to enforce lexical scoping for free variables)

cs480 (Prasad)

L156HOF

20

Application

- Instantiating generic functions.
- *Object-oriented Programming*
Using (1) `let`-construct (to introduce local variables) and (2) assignments, *objects* and *classes* can be simulated.
- *Streams*
Creation of “infinite” data structures.

cs480 (Prasad)

L156HOF

21

Lambda Expressions

- `(define (id x) x)`
- `(define id (lambda (x) x))`
- `(lambda (x) x)`
- `((lambda (x) x) 5)`
- `(lambda () 5)`

cs480 (Prasad)

L156HOF

22

Lambda Expressions

- `(lambda () 1)`
- `((lambda () 1))`
- `((lambda (x) x)`
 `(lambda (y) y))`
- `((lambda (x) (x x))`
 `(lambda (y) y))`

cs480 (Prasad)

L156HOF

23