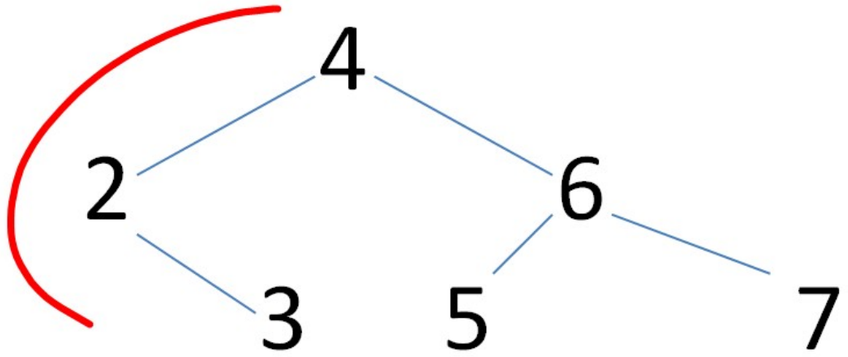
Programming Languages

Wael Mustafa

Faculty of Engineering and Information Technology

Printing a Binary Tree




```
> (print-tree '(4 (2 () (3 () ())) (6 (5 () ()) (7 () ()))))
4
|-2
| |- -3
|-6
| |- -5
| |- -7
```

Printing a Binary Tree

```
(define (print-tree tree)
  (print-tree-rec tree 0))
(define (print-tree-rec tree D)
  (cond ((null? tree))
        (else (print-spaces D)
              (display (key tree) (newline))
              (print-tree-rec (left tree) (+ D 1))
              (print-tree-rec (right tree) (+ D 1)))))
```

Printing a Binary Tree

```
(define  (print-spaces  N)
  (cond  ((= N  0))
         (else (write " ")
               (print-spaces  (-  N  1)))))
```

1 space

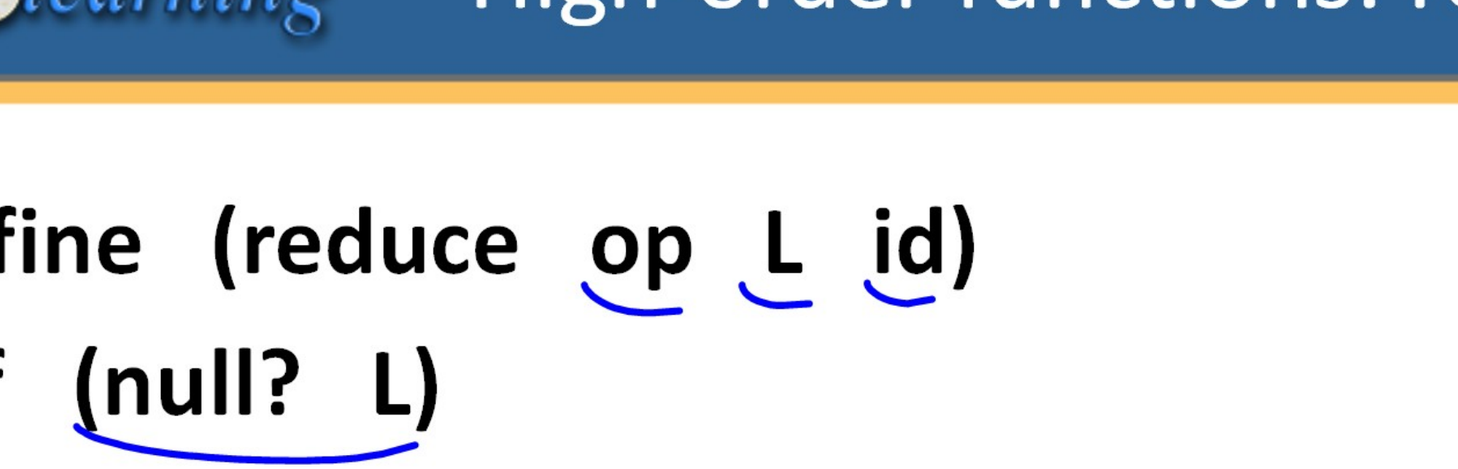
High-order functions: reduce

```
> (reduce  +  '(1 2 3)  0)
= (+ 1 (+ 2 (+ 3 0)))
=6
> (reduce  /  '(64  8  4)  2)
= (/  64  (/  8  (/  4  2)))
= 16
> (reduce  union '((1  3)  (2  3) (4  5))  '())
(1  2  3  4  5)
```

```
> (mapcar '(1 2) '(2 3))
(1 2 3)
map
apply
eval
```

High-order functions: reduce

```
(define  (reduce  op  L  id)
  (if  (null?  L)
       id
       (op  (car  L)  (reduce  op  (cdr  L)  id))))
```



Let and let*

- Used to define local variables in a new environment
- Syntax:

  (let ((v1 e1) (v2 e2) … (vn en)) expr)

  (let* ((v1 e1) (v2 e2) … (vn en)) expr)

- Both bind v1, .., vn to the values of e1, …, en in the *expr*
- Let does the binding in parallel
- Let* does the binding sequentially
- Both return the value of the *expr*

Let and let*

```
> (let ((x 2)) (* x x))
4
> (let ((x 4) (y (+ x 2))) (+ x y))
Runt-time error: unbound variable x
> (let* ((x 4) (y (+ x 2))) (+ x y))
10
> (let ((x 4)) (let ((y (+ x 2))) (* x y)))
24
> (let ((x 0) (y 1)) (let ((x y) (y x)) (list x y)))
(1 0)
> (let ((x 0) (y 1)) (let* ((x y) (y x)) (list x y)))
(1 1)
```

Let and let* are not primitive

- **(let ((v1 e1) (v2 e2)) expr)**

  Is equivalent to

((lambda (v1 v2) expr) e1 e2)

- **(let* ((v1 e1) (v2 e2)) expr)**

  Is equivalent to

((lambda (v1) ((lambda (v2) expr) e2)) e1)

closures

;; one closure is created and referenced through my-counter

```
(define my-counter
        (let ((count 0))
          (lambda () (set! count (+ count 1))
                     count)))

> (my-counter)
1
> (my-counter)
2
> (my-counter)
2
```

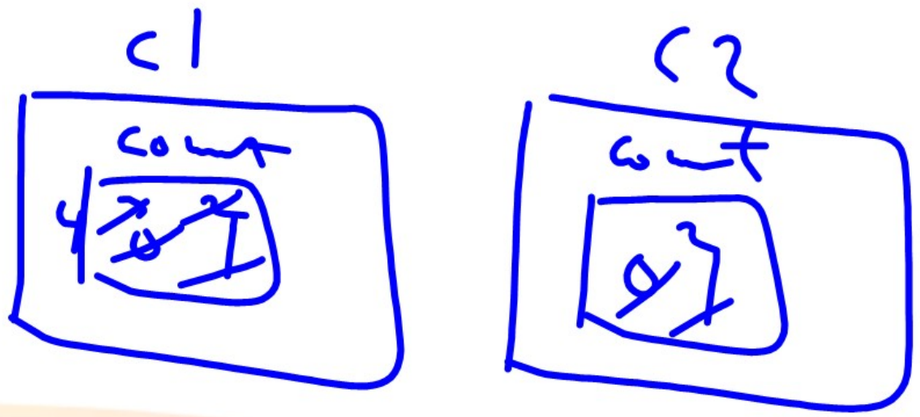
closures

;; every time make-counter executes a new closure is created with an independent copy of environment (count)

```
(define  (make-counter)
  (let  ((count  0))
        (lambda  ()  (set!  count  (+  count  1))  count)))
;;Or
(define  make-counter
  (lambda ()
    (let ((count  0))
         (lambda  ()  (set!  count  (+  count  1))  count))))
```

closures

```
> (define   c1   (make-counter))
c1
> (define   c2   (make-counter))
c2
> (define   c3   (make-counter))
c3
> (c1)
1
> (c1)
2
> (c2)
1
> (c2)
2
> (c1)
3
> (c1)
4
> (c3)
1
```



Delayed evaluation and thunks

- Any expression **e** as a function argument, for example **(+ x 2)**, is always evaluated and the result is obtained (eager evaluation)
- Sometimes it is more efficient to delay the evaluation
- A **thunk** of an expression **e** is zero argument function when executed evaluates **e** and returns the result
- To Thunk an expression e

  (define th (lambda () e))
- To evaluate e use **(th)**

Eager evaluation can be a problem

```
(define   (my-if-1   x    y   z)
    (if   x   y   z))
(define   (fact-wrong   n)
    (my-if-1     (= n 0)      1      (* n  (fact-wrong (- n 1))))
```

- Infinite recursion in fact-wrong. It keeps calling it self at n, n-1, n-2, …, 1, 0, -1, -2, …

(fact-wrong 0) =

(my-if-1 true 1 (* 0 (fact-wrong -1))) =

(my-if-1 true 1 (* 0 (my-if-1 false 1 (* -1 (fact-wrong -2))))) =

…

Solving the problem using thunks

```
(define   (my-if-2   x    y    z) ;;y and z are assumed to be thunks
    (if   x   (y)   (z)))
(define   (fact   n)
    (my-if-2   (= n 0)
               (lambda () 1)
               (lambda () (* n  (fact (- n 1))))))
```

How good is thunking?

(define (g …) …);; g involves large computation

(define (h th)
(if some-cond 1 (th)))

(h (lambda () (g …)))

- Great savings when **some-cond** is true
- No loss and no win if **some-cond** is false. The function g is executed one time

How good is thunking?

```
(define (g ...) ...);; g involves large computation

(define (h    th)
  (let  ((v1    (if   c1    1   (th)))
         (v2    (if   c2   2   (th)))
                  ...
         (vn    (if   cn   n   (th))))
    ...  ))

(h     (lambda  ()  (g ...)))
```

- The **g** function will be evaluated for every true condition. There will be loss when two (or more) conditions evaluate to true.
- Next we use delay and force to avoid multiple evaluation of thunks

Proper lists and pairs

- a proper list is a list in which cdr is a list.
- **(cons *elem list*)** always produces a proper list in which car is *elem* and cdr is *list*.
- **(cons atom1 atom2)** produces a pair (ANA improper list) written as ‘(atom1.atom2)
- car of the pair returns atom1 and cdr returns atom2
- (cons 5 ‘a) returns the pair ‘(5.a)
- (car ‘(5.a)) returns 5
- (cdr ‘(5.a)) returns a
- list? determins if argument is a proper list
- (list? (cons 7 ‘(a b c))) returns true
- (list? (cons 7 ‘a)) returns false

Mutable pairs

- **mcons** is similar to cons but makes a mutable pair
- **mcar** returns the first component of a mutable pair
- **mcdr** returns the first component of a mutable pair
- **mpair?** determine if argument is a mutable pair

```
> (set!  x  (mcons  3    4))
‘(3.4)
> (mcar  x)
3
> (mcdr  x)
4
> (mpair?  x)
true
```

Mutable pairs

- **set-mcar!** takes a mutable pair and an expression, evaluates expression, and changes the first component in the pair to the expression value
- **set-mcdr!** takes a mutable pair and an expression, evaluates expression, and changes the second component in the pair to the expression value

```
> (set!  x  (mcons  a    10))
(a.10)
> (set-mcdr!  x   (*  (mcdr  x)  5))
50
> (set-mcar!  x   'b)
b
> (display x)
(b.50)
```