Programming Languages

Wael Mustafa

Faculty of Engineering and Information Technology

Functions as returned values

```
(define   (plus-list  x)
   (cond      ((number?   x)
               (lambda   (y)  (+  (sum  x)  y)))
              ((list?   x)
               (lambda  (y)  (+  (sum-list   x)  y)))
              (else   (lambda (z) z))))
> ((plus-list   3))   4) ; execute closure on 4
10
> ((plus-list   '(1  3  5))   5); execute closure on 5
14
```

Functions as returned values: currying

```
(define pow
  (lambda  (x)
    (lambda  (y)
       (if   (=  y  0)
             1
             (*   x   ((pow   x)  (-   y  1)))))))
(define three-to-the   (pow  3))
(define  eightyone  (three-to-the 4))
(define sixteen   ((pow  2)   4))
```

3^x

2^x

Built-in higher-order functions: map

- map takes two arguments: a function and a list
- map builds a new list whose elements are the results of applying the function to each element of the input list

```
> (map  abs  '(-1  2  -3  4))
(1  2  3  4)
> (map  (lambda  (x)  (+  1  x))  '(-1  3  0))
(0  4  1)
```

Map function

```
(define (map  f   L)
   (cond   ((null?   L)   '( ))
               (else   (cons   (f   (car   L))
                                    (map   f   (cdr   L)))))))
```

Eval function

Evaluates an expression and returns the result

\> (cons ‘* ‘(3 2 5))

(* 3 2 5)

\> (eval (cons ‘* ‘(3 2 5)))

30

\> (eval (list ‘+ 3 3 5))

11

’(+ 3 3 5)

\> (eval (append ‘(* 2 3) ‘(4)))

~~16~~ 24

’(* 2 3 4)

\> (eval (cons ‘append ‘(‘(1 2) ‘(3))))

(1 2 3)

’(append (1 2) (3))

\> (append x y)

eval

What is wrong?

```
(define (atomcount s)
  (cond  ((null?  s)  0)
         ((atom? s)  1)
         (else  (+  (map  atomcount  s)))))
```

'((2) a ((b)))

3

(1 1 1)

```
> (atomcount  '(a  b))
```

Run-time error: + expects <number> but given (1 1)

```
> (+ 7 10 -30)
-13
> (+ '(1 1 1))
```

Correct atomcount using eval

```
(define   (atomcount   s)
  (cond   ((null?   s)   0)
          ((atom?   s)  1)
          (else   (eval   (cons   '+
                            (map  atomcount   s))))))
```

(1 1)

(+ 1 1)

```
> (atomcount  '(a  b))
2
> (  atomcount  '((5) ((2 1)) (((7))))   )
4
```

1

High-order functions: apply

- Takes two arguments: a function and a list containing arguments
- Executes the function on the arguments inside the list and returns the result

> (apply + '(3 3 5)) = (+ 3 3 5)

11

> (apply append '((7 3) (5 4))) =

(7 3 5 4)

(append '(7 3) '(5 4))

Correct atomcount using apply

```
(define (atomcount s)
  (cond ((null? s) 0)
        ((atom? s) 1)
        (else (apply + (map atomcount s)))))
```

'(1 1)

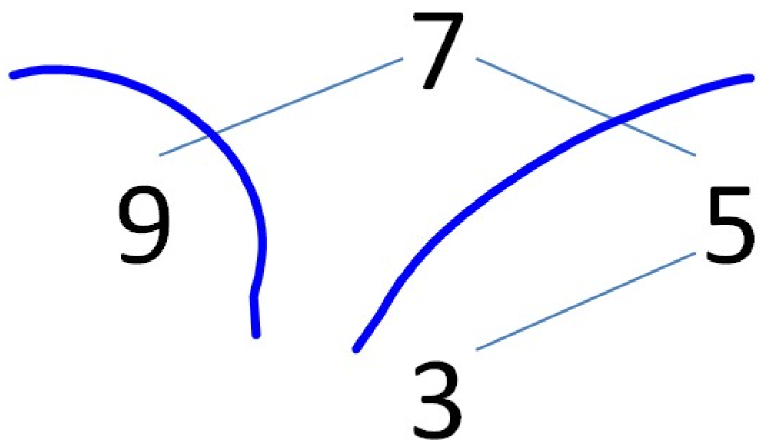
> (atomcount '(a b))

2

Binary search trees in Racket

- Represent a tree as a list
- (root-element left-subtree right-subtree)
- Left-subtree and right-subtree are lists
- An empty tree is represented by an empty list ( )

Example

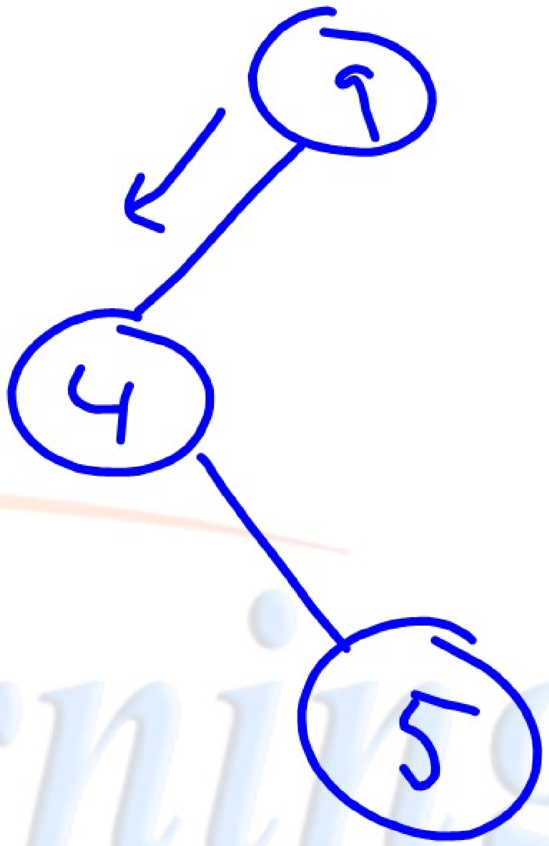


(7 (9 ( ) ( )) (5 (3 ( ) ( )) ( )))

Binary search tree insertion

> (insert 5 ‘())

(5 () ())



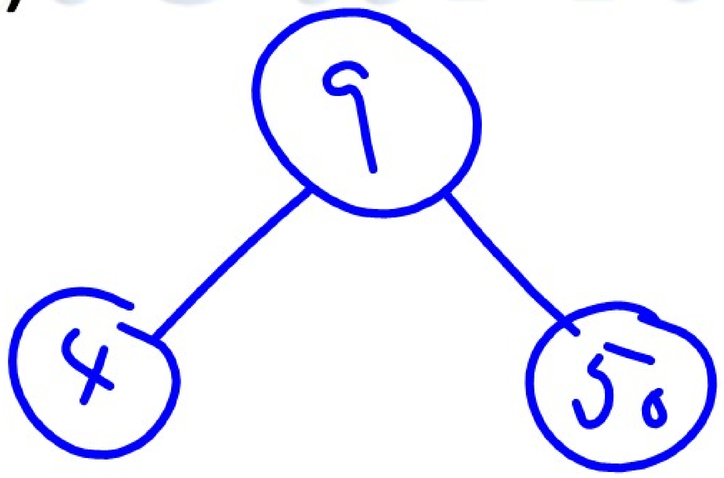
> (insert 5 ‘(9 (4 () ()) ()))

(9 (4 () (5 () ())) ())

> (insert 50 ‘(9 (4 () ()) ()))

(9 (4 () ()) (50 () ()))

> (insert

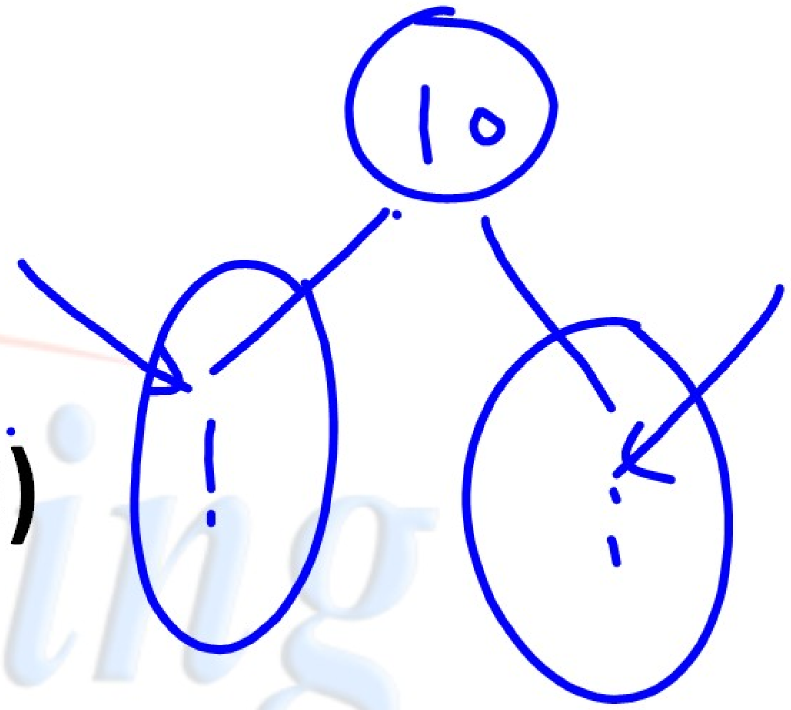


Binary search tree insertion

```
(define (key tree) (car tree))
(define (left tree) (cadr tree))
(define (right tree) (caddr tree))
```

Binary search tree insertion

```
(define (insert  el  tree)
   (cond ((null? tree) (list el () ()))
         ((= el (key tree)) tree)
         ((< el (key tree))
            (list (key tree)
                  (insert el (left tree))
                  (right tree)))
         (else  (list (key tree)
                      (left tree)
                      (insert el (right tree))))))
```



Building a binary search tree from a list

- Given a list of numbers L, start from an empty BST and go through elements in L inserting each into the BST

```
(define (list-to-tree  L) (list-to-tree-helper   L     '()   ))

(define (list-to-tree-helper   L    tree)
    (cond  ((null?  L)  tree)
              (else  (list-to-tree-helper
                                  (cdr  L)
                                  (insert   (car L)    tree)))))
```

f

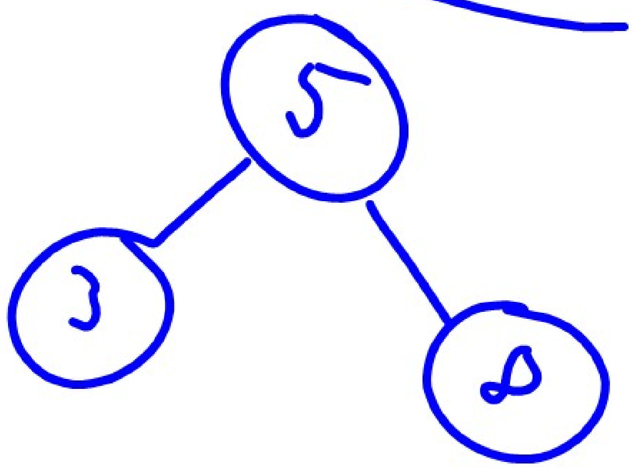
'(5)

'((5 () ()))

= (f '() '(5 () ()))

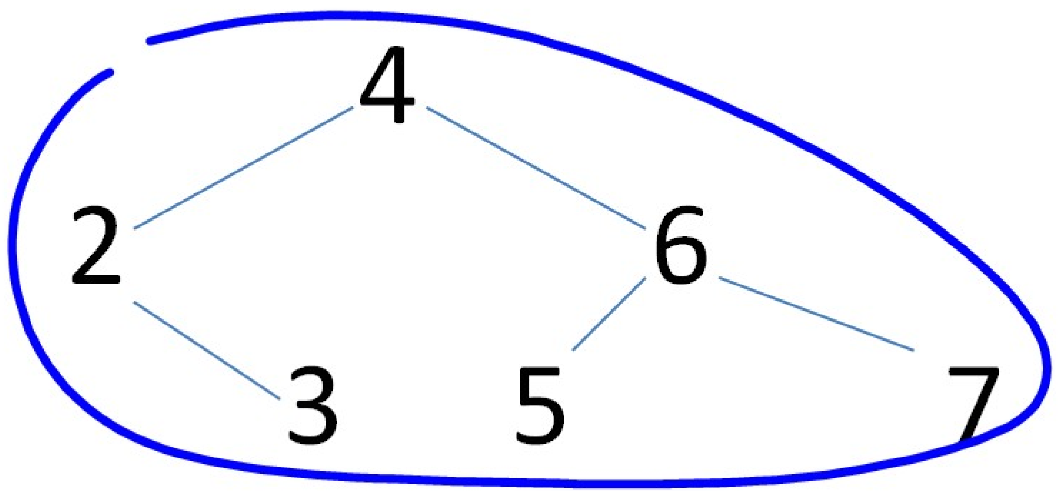
5

> (list-to-tree '(5 ~~3 8~~))
(5 (3 () ()) (8 () ()))

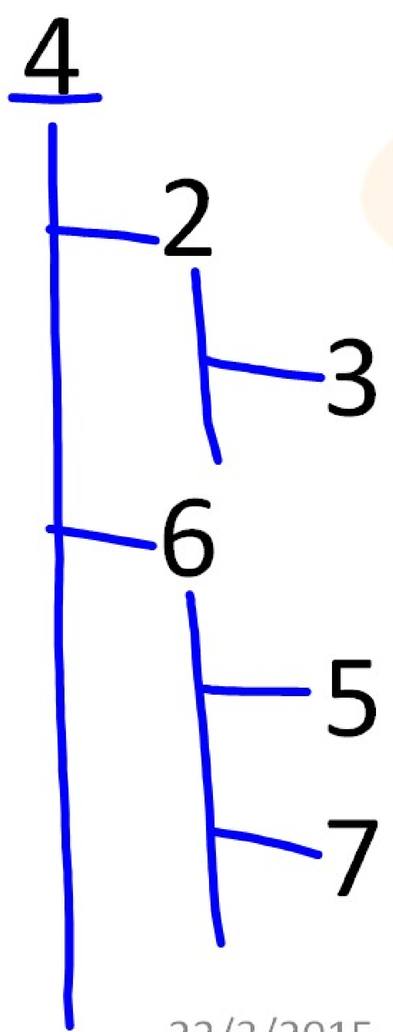


=

Printing a Binary Tree



> (print-tree ‘(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)))))
```

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)
```

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)