UNIVERSITY OF WATERLOO

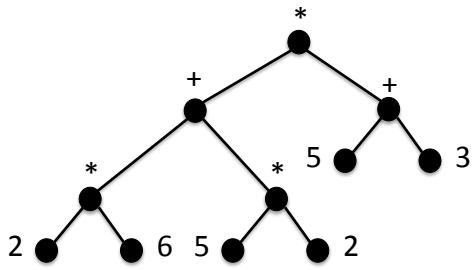
David R. Cheriton School of Computer Science

CS 115: Introduction to Computer Science

Presented By Ahmed Ibrahim

Fall 2015

Recall: General Arithmetic Expressions



((2 * 6) + (5 * 2)) * (5 + 3)

```
(make-binode '*
    (make-binode '+
            (make-binode '* 2 6)
            (make-binode '* 5 2))
    (make-binode '+ 5 3))
```

```
(list '*
    (list '+
            (list '* 2 6)
            (list '* 5 2))
    (list '+ 5 3))
```

Nested Lists

- In this course, we have discussed flat lists (no nesting),
  - for example: (list 1 'a "hello" 'x);
- And lists of lists (**one level of nesting**):
  - (list (list 1 "a") (list 2 "b"))
- We now consider **nested lists** (arbitrary nesting):

```
(list
    (list 1 (list 2 3))
    4
    (list 5 (list 6 7 8) 9) )
```

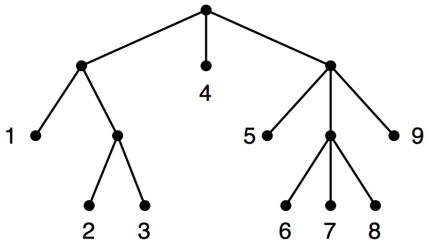

Recall: Variations on trees

- Number of children of internal nodes:
  - ✓ at most two
  - ✓ exactly two
  - ✓ any number
- Labels:
  - ✓ on all nodes
  - – just on leaves
- ✓ Order of children (matters or not)
- ✓ Tree structure (from data or for convenience)

Nested lists as **leaf-labeled** trees

- It is often helpful to visualize a nested list as a leaf-labeled tree, in which the leaves correspond to the elements of the list, and the internal nodes indicate the nesting.

```
(list
    (list 1 (list 2 3))
    4
    (list 5 (list 6 7 8) 9) )
```



- Why? it is common practice to use leaf-labeled trees to represent the evolution of species, populations, organisms and more.

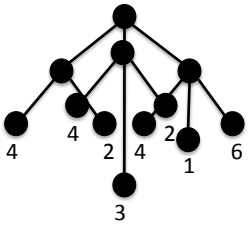
Examples of **leaf-labeled** trees:

Empty

(list 4 2)

(list (list 4 2) 3 (list 4 1 6))

(list (list 3) 2 (list 5) (list 4 (list 3 6)))



Each non-empty tree is a list of sub-trees.

The first sub-tree in the list is either

- a single leaf (not a list) or
- a sub-tree rooted at an internal node (a list).

Data definition for **leaf-labeled** trees

```
;; A leaf-labelled tree (LLT) is one of the following
;; * empty
;; * (cons Num LLT)
;; * (cons LLT LLT) where first LLT is nonempty.
```

The labels could be **ANY** non-list Racket value.

Template for **leaf-labeled** trees

The template follows from the data definition.

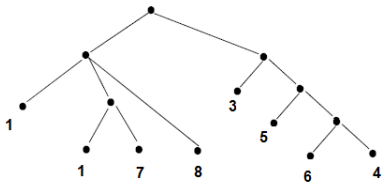
```
;; (define (my-llt-fun l)
;;   (cond
;;   [(empty? l) . . . ]
;;   [(cons? (first l))
;;          . . . (my-llt-fun (first l)) . . .
;;          . . . (my-llt-fun (rest l)) . . . ]
;;   [else . . .  (first l) . . .  (my-llt-fun (rest l)) . . . ]))
```

```
;; A leaf-labelled tree (LLT) is one
of the following
;; * empty
;; * (cons Num LLT)
;; * (cons LLT LLT) where first LLT is
;; nonempty.
```

The function count-leaves

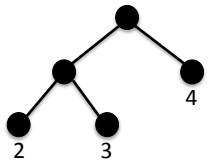
```
(define (count-leaves l)
(cond
    [(empty? l) 0]
    [(cons? (first l))
        (+
          (count-leaves (first l))
          (count-leaves (rest l)))]
    [else
        (+ 1
           (count-leaves (rest l)))]))
```

```
(define (my-llt-fun l)
    (cond
    [(empty? l) . . . ]
    [(cons? (first l))
            . . . (my-llt-fun (first l)) . . .
            . . . (my-llt-fun (rest l)) . . . ]
    [else . . .  (first l) . . .  (my-llt-fun (rest
l)) . . . ]))
```



Condensed trace of count-leaves

(count-leaves (list (list 2 3) 4))
⇒ (+ (count-leaves (list 2 3)) (count-leaves (list 4)))
⇒ (+ (+ 1 (count-leaves (list 3))) (count-leaves (list 4)))
⇒ (+ (+ 1 (+ 1 (count-leaves (list)))) (count-leaves (list 4)))
⇒ (+ (+ 1 (+ 1 0)) (count-leaves (list 4)))
⇒ (+ (+ 1 1) (count-leaves (list 4)))
⇒ (+ 2 (count-leaves (list 4)))
⇒ (+ 2 (+ 1 (count-leaves (list))))
⇒ (+ 2 (+ 1 0))
⇒ (+ 2 1)
⇒ 3




```
(define (count-leaves l)
(cond
    [(empty? l) 0]
    [(cons? (first l))
        (+
          (count-leaves (first l))
          (count-leaves (rest l)))]
    [else
        (+ 1
           (count-leaves (rest l)))]))
```

Flattening a nested list

- flatten produces a flat list from a nested list.

```
;; flatten: LLT → (listof Num)
;; (define (flatten l) . . . )
```

- We make use of the built-in Racket function, append, which we examined in Module 7.
- (append (list 1 2) (list 3 4)) ⇒ (list 1 2 3 4)
- You should continue to use cons when constructing a list from a single element and another list.

REMEMBER

Example: flatten function

```
(define l (list (list 1 2) (list 3 4)))
;; (flatten l) produces a flat list from l.
;; flatten: LLT → (listof Num)
(define (flatten l)
    (cond
        [(empty? l) empty]
        [(cons? (first l)) (append (flatten (first l))
                                   (flatten (rest l)))]
        [else (cons (first l) (flatten (rest l)))]))
```

Condensed trace of flatten

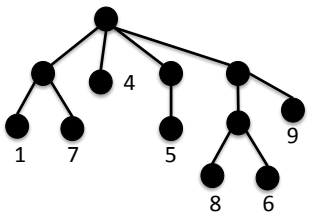
(flatten (list (list 2 3) 4))
⇒ (append (flatten (list 2 3)) (flatten (list 4)))
⇒ (append (cons 2 (flatten (list 3))) (flatten (list 4)))
⇒ (append (cons 2 (cons 3 (flatten (list)))) (flatten (list 4)))
⇒ (append (cons 2 (cons 3 empty)) (flatten (list 4)))
⇒ (append (cons 2 (cons 3 empty)) (cons 4 (flatten (list))))
⇒ (append (cons 2 (cons 3 empty)) (cons 4 empty))
⇒ (cons 2 (cons 3 (cons 4 empty)))

```
(define (flatten l)
    (cond
        [(empty? l) empty]
        [(cons? (first l))
(append (flatten (first l))
(flatten (rest l)))]
        [else (cons (first l)
(flatten (rest l)))]))
```


Practical Exercise

;; A leaf-labelled tree (**LLT**) is one of the following
;; * empty
;; * (cons Num LLT)
;; * (cons LLT LLT) where first LLT is nonempty.

Draw the tree representation for the following LLT:
(list (list 1 7) 4 (list 5) (list (list 8 6) 9)).



Additional Practical Exercise

Write the function ***llt-replace*** which consumes a leaf-labeled tree (tree) and produces a new leaf-labeled tree by replacing all leaves in tree with the symbol 'leaf.

For example, (llt-replace (list (list 1 2) 3 (list 4 5))) should produce (list (list 'leaf 'leaf) 'leaf (list 'leaf 'leaf)).

```
(define (llt-replace tree)
(cond
 [(empty? tree) empty]
 [(cons? (first tree))
  (cons (llt-replace (first tree))
        (llt-replace (rest tree)))]
 [else
  (cons 'leaf (llt-replace (rest tree)))]))
```

```
(define (my-llt-fun l)
        (cond
        [(empty? l) . . . ]
        [(cons? (first l))
                . . . (my-llt-fun (first l)) . . .
                . . . (my-llt-fun (rest l)) . . . ]
        [else . . .  (first l) . . .  (my-llt-fun (rest
l)) . . . ]))
```


cons, list & append

THANK YOU!