# General trees

CS135 Lecture 15

# L15.0 Lists of lists of lists of lists of…

# Arbitrarily nested lists

First, we had lists:

```
(list 'apple 'eggs 'bread 'apple 'milk 'bread)
```

Then, we had lists of lists:

```
(list (list 1 1) (list -1 1) (list -1 -1) (list -1 1))
```

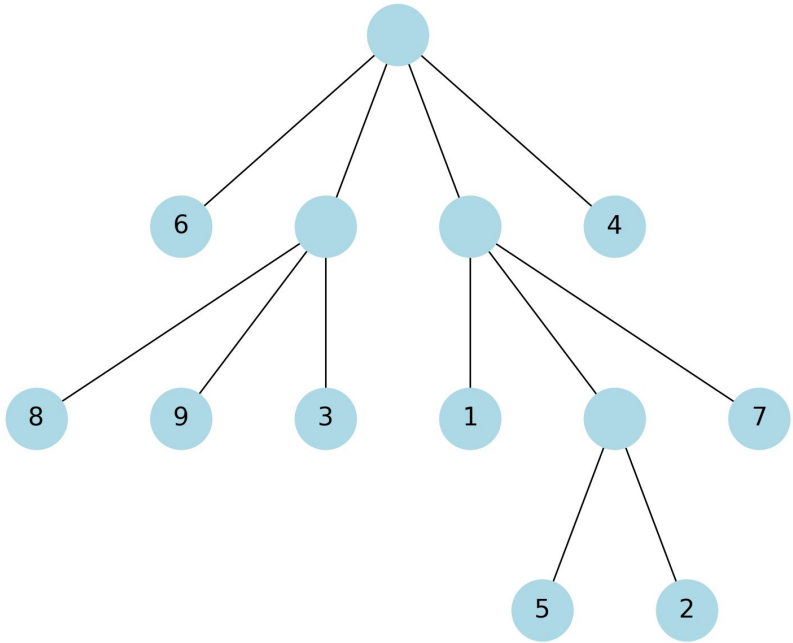
Today we will consider **all** of the remaining cases: lists of lists of lists, lists of lists of lists of lists, lists of lists of lists of lists of lists, ….

Or more generally, *arbitrarily nested lists*:

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

# Representing trees with arbitrarily nested lists

```
(define nt-eg
  (list 6
        (list 8 9 3)
        (list 1
              (list 5 2)
              7)
        4))
```




In this example, the list can be viewed as representing a tree where nodes can have any number of children and only the leaves are labeled.

# Data definition

For simplicity, we use trees of numbers in our example, but we could also have trees of strings, points, symbols, or anything we like.

```
;; A tree of numbers (NTree) is one of
;; * Num
;; * (cons NTree (listof NTree))
```

More simply an **`NTree`** is a number or a non-empty list of **`NTree`**.

Note that **`empty`** is not an **`NTree`**. There is no empty **`NTree`**.

# NTree?

```
;; Predicate to check that something is an NTree
;; NTree?: Any -> Bool
;;
(define (NTree? x)
  (cond [(number? x) true]
        [(not (cons? x)) false]
        [(empty? (rest x)) (NTree? (first x))]
        [else (and (NTree? (first x)) (NTree? (rest x)))]))
(check-expect (NTree? nt-eg) true)
(check-expect (NTree? pi) true)
(check-expect (NTree? empty) false)
```

# Template for `NTree`

Trees of numbers are complicated enough that we want to explicitly define a template for manipulating them. It follows directly from the data definition.

```
(define (ntree-template nt)
  (cond [(number? nt)...]
        [(empty? (rest nt))
           (... (ntree-template (first nt)))]
        [else (... (ntree-template (first nt))
                   (ntree-template (rest nt)))]))
```

The cases are: 1) a number, 2) a tree with one subtree, and 3) a tree with multiple subtrees.

# **Example**: Adding the leaves of an **`NTree`**

```
;; Add the numbers in a NTree
;; nt-add: NTree -> Num
(define (nt-add nt)
  (cond [(number? nt) nt]
        [(empty? (rest nt)) (nt-add (first nt))]
        [else (+ (nt-add (first nt)) (nt-add (rest nt)))]))

(check-expect (nt-add nt-eg) 45)
(check-expect (nt-add (list 100 101 102)) 303)
(check-within (nt-add pi) pi 0.000000001)
```

# Lots of potential exam questions here…

1. Does an `NTree` contain a particular value?
2. Are all the values in an `NTree` positive?
3. How many times does a particular number appear?
4. How deep is an `NTree`?
5. How long is the longest list in an `NTree`?
6. Produce the equivalent `NTree` with all the values squared.
7. Are two trees “the same”?
8. …

We won’t ignore such a productive source of exam questions that are easy for us to state. If you happen to be studying for an exam, try ~~some~~ all of these.

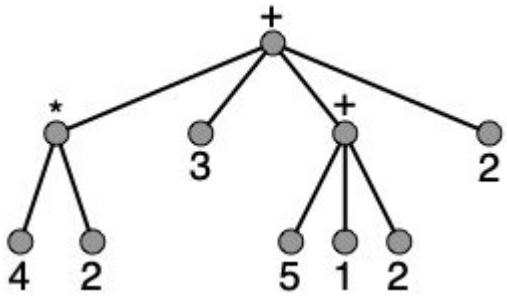
# L15.1 Expression trees

# Expression trees

The internal nodes of an **`NTree`** are unlabeled.

If we add operators to the nodes, we can represent arithmetic expressions.

```
(+ (* 4 2)
   3
   (+ 5 1 2)
   2)
```



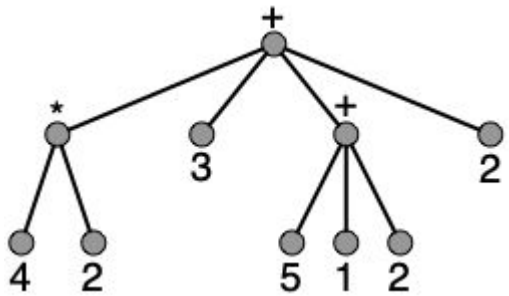

For simplicity, we will restrict the operations to `+` and `*`.

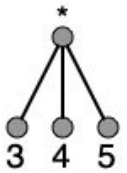
# Data definition

```
;; An arithmetic expression (AExp) is one of:
;; * Num
;; * (cons (anyof '+ '*) (listof AExp))

(define ae-eg1
  (list '+
        (list '* 4 2)
        3
        (list '+ 5 1 2)
        2))

(define ae-eg2 (list '* 3 4 5))
```



# `'+` and `'*` are symbols representing operators

We are representing an expression as a tree. One of the advantages of Racket and related languages, such as Scheme and Lisp,  is the structural resemblance between a function definition (“code”) and a list (“data”). Treating code as data means you can write functions that modify, analyze, or even generate other functions. This “meta-programming” ability can makes it easier to build complex and flexible systems.

Code: `(+ (* 4 2) 3 (+ 5 1 2) 2)`

Data: `(list '+ (list '* 4 2) 3 (list '+ 5 1 2) 2)`

# Evaluating an expression tree

```
;; evaluate an arithmetic expression
;; eval: AExp -> Num
(define (eval exp)
  (...))


(check-expect (eval ae-eg1) 21)
(check-expect (eval ae-eg2) 60)
```

We have two cases: 1) a number, and 2) an operator followed by a list of **`AExp`**. Unlike an **`NTree`**, we have only two cases because we are not recurring on the operator, we are applying the operator to the list of **`AExp`**.

# Evaluating an expression tree

```
;; evaluate an arithmetic expression
;; eval: AExp -> Num
(define (eval exp)
  (cond [(number? exp) exp]
        [else (... (first exp) (rest exp))]))

(check-expect (eval ae-eg1) 21)
(check-expect (eval ae-eg2) 60)
```

How do we recursively call **`eval`**?

# Evaluating an expression tree

```
;; evaluate an arithmetic expression
;; eval: AExp -> Num
(define (eval exp)
  (cond [(number? exp) exp]
        [else (apply (first exp) (rest exp))]))

(check-expect (eval ae-eg1) 21)
(check-expect (eval ae-eg2) 60)
```

A key observation is that **`eval`** can't be recursively applied to an operator followed by a list of **`AExp`**. We need a separate function (**`apply`**) to do that.

# Applying an operator to a list of expression trees

```
;; apply an operator to a list of arithmetic expressions
;; apply: (anyof '+ '*) (listof AExp) -> Num
(define (apply op el)
  (cond [(empty? el) (cond [(symbol=? '+ op) 0]
                           [(symbol=? '* op) 1])]
        [(symbol=? '+ op) (+ (eval (first el))
                             (apply op (rest el)))]
        [(symbol=? '* op) (* (eval (first el))
                             (apply op (rest el)))]))

(check-expect (apply '+ (list 1 4 (list '* 3 3) 16)) 30)
```

# Mutual recursion between **`eval`** and **`apply`** (1/3)

```
(eval (list '+ 7 (list '* 2 3)))⇒...

(apply (first (list '+ 7 (list '* 2 3))
       (rest (list '+ 7 (list '* 2 3))))⇒...

(apply '+ (list 7 (list '* 2 3)))⇒...

(+ (eval (first (list 7 (list '* 2 3))))
   (apply '+ (rest (list 7 (list '* 2 3)))))⇒...

(+ (eval 7)
   (apply '+ (rest (list 7 (list '* 2 3)))))⇒...

(+ 7 (+ (eval (list '* 2 3))
        (apply '+ (rest (list (list '* 2 3))))))⇒...
```

# Mutual recursion between **`eval`** and **`apply`** (2/3)

```
(+ 7 (+ (eval (list '* 2 3))
        (apply '+ (rest (list (list '* 2 3))))))⇒...

(+ 7 (+ (apply '* (list 2 3))
        (apply '+ (rest (list (list '* 2 3))))))⇒...

(+ 7 (+ (* (eval 2) (apply '* (rest (list 2 3))))
        (apply '+ (rest (list (list '* 2 3))))))⇒...

(+ 7 (+ (* 2 (apply '* (rest (list 2 3))))
        (apply '+ (rest (list (list '* 2 3))))))⇒...

(+ 7 (+ (* 2 (apply '* (list 3)))
        (apply '+ (rest (list (list '* 2 3))))))⇒...
```

# Mutual recursion between **`eval`** and **`apply`** (3/3)

```
(+ 7 (+ (* 2 (apply '* (list 3)))
        (apply '+ (rest (list (list '* 2 3))))))⇒...

(+ 7 (+ (* 2 (* 3 (apply '* (rest (list 3)))))
        (apply '+ (rest (list (list '* 2 3))))))⇒...

(+ 7 (+ (* 2 (* 3 1))
        (apply '+ (rest (list (list '* 2 3))))))⇒...

(+ 7 (+ 6 (apply '+ empty)))⇒...

(+ 7 (+ 6 0))⇒...

13
```

# L15.3 `(listof Any)`

# `(listof Any)`

The most general type for a list is `(listof Any)`, where `Any` includes `(listof Any)` itself, along with `Num`, `Bool`, `Sym`, `Char`, and `Str`.

Some examples:

```
empty
(list 'apple 'eggs 'bread 'apple 'milk 'bread)
(list (list 1 1) (list -1 1) (list -1 -1) (list -1 1))
(list 6 (list 8 9 3) (list 1 (list 5 2) 7) 4)
(list empty (list empty empty (list empty)) empty)
(list true (list false (list true (list false))))
```

Even the empty list is an acceptable element for a `(listof Any)`.

# Atoms

In Racket, and in related languages like Scheme and Lisp, any data that is not a list is sometimes called an “atom”.

Atoms include **`Num`**, **`Bool`**, **`Sym`**, and **`Char`**. A **`Str`** is considered to be an atom, although in CS135 we also consider it to be a composite data type.

```
(define (atom? x)
  (not (list? x)))
(check-expect (atom? 100) true)
(check-expect (atom? "hello") true)
(check-expect (atom? empty) false)
```

We will use **`Atom`** in contracts to mean “anything not a list”.

# Data definition

This data definition is an extension of the data definition for lists by allowing `first` to be an `Atom` or another `(listof Any)`.

```
;; A (listof Any) is one of:
;; * empty
;; * (cons Atom (listof Any))
;; * (cons (listof Any) (listof Any))
```

From this data definition, you should now be able to visualize both the template and the associated “rules”, if we were to require them.

# Flattening a `(listof Any)` to a `(listof Atom)`

```
;; flatten an arbitrarily nested list to a list of atoms
;; flatten: (listof Any) -> (listof Atom)
(define (flatten lst)
  (cond [(empty? lst) lst]
        [(atom? (first lst))
         (cons (first lst) (flatten (rest lst)))]
        [else
         (append (flatten (first lst)) (flatten (rest lst)))]))
(check-expect (flatten empty) empty)
(check-expect
 (flatten (list 'apple 'eggs 'bread 'apple 'milk 'bread))
 (list 'apple 'eggs 'bread 'apple 'milk 'bread))
```

# Test cases for flattening an arbitrarily nested list

```
(check-expect
 (flatten
  (list (list 1 1) (list -1 1) (list -1 -1) (list -1 1)))
 (list 1 1 -1 1 -1 -1 -1 1))
(check-expect
 (flatten (list 6 (list 8 9 3) (list 1 (list 5 2) 7) 4))
 (list 6 8 9 3 1 5 2 7 4))
(check-expect
 (flatten (list empty (list empty empty (list empty)) empty))
 ???)
(check-expect
 (flatten (list true (list false (list true (list false)))))
 (list true false true false))
```

# L15.4 Quote notation

# Writing lists more concisely

Racket supports a concise notation for writing lists called “quote notation”, or just “quoting”, which makes it easier to write arbitrarily nested lists.

To use quote notation in Racket, simply prefix the list with an apostrophe—for example, write `'(a b c)` for `(list 'a 'b 'c)`. Notice there is just one apostrophe at the front of the list. Each symbol doesn’t need an apostrophe.

```
'()
'(apple eggs bread apple milk bread)
'((1 1) (-1 1) (-1 -1) (-1 1))
'(6 (8 9 3) (1 (5 2) 7) 4)
'(#t (#f (#t (#f))))
'(+ (* 4 2) 3 (+ 5 1 2) 2)
```

# Quoting

Quote notation provides a super-compact notation for lists.

**`cons`** notation emphasizes a fundamental characteristic of a list: It has a first element and the rest of the elements. Elements of the list can be computed as the list is constructed. But writing out a list with **`cons`** notation is unwieldy.

**`list`** notation makes our lists more compact but loses the reminder about the first element and the rest. Like **`cons`**, elements of the list can be computed as the list is constructed.

Quote notation is even more compact but loses the ability to compute elements during construction.

We will sometimes use quote notation in for convenience in lectures and assignments, but it will not be tested on exams.

# L15 Summary

# L15: You should know

- The relationship between arbitrarily nested lists and trees.
- Producing and consuming trees of numbers (**`NTree`**).
- Representing expressions as trees.
- Evaluating expression trees.
- What an “atom” is in Racket and related languages
- Producing and consuming **`(listof Any)`** and **`(listof Atom)`**.
- How to read quote notation.

# L15: Allowed constructs

Newly allowed constructs:

```
Atom '
```

Previously allowed constructs:

```
( ) [ ] + - * / = < > <= >= ;
abs acos add1 and append asin atan boolean? char? char=?
char<? check-expect check-within cond cons cons? cos define
e else empty empty? even? exp expt false first inexact?
integer? length list list? log max min not number? odd? or
pi quotient rational? remainder rest reverse second sin sqr
sqrt string? string=? string<? string->list list->string
sub1 symbol? symbol=? tan third true zero?
listof Any anyof Bool Char Int Nat Num Rat Str Sym
```