Trees

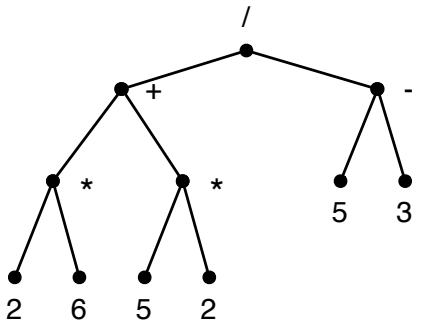
Readings: HtDP, sections 14, 15, 16.

Topics:

- Introductory examples and terminology
- Binary trees
- Binary search trees
- Augmenting trees
- Binary expression trees
- General arithmetic expression trees
- Nested lists

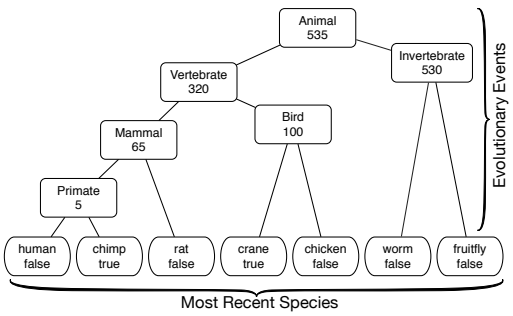
Example: binary expression trees

The expression $((2 * 6) + (5 * 2))/(5 - 3)$ can be represented as a **tree**:

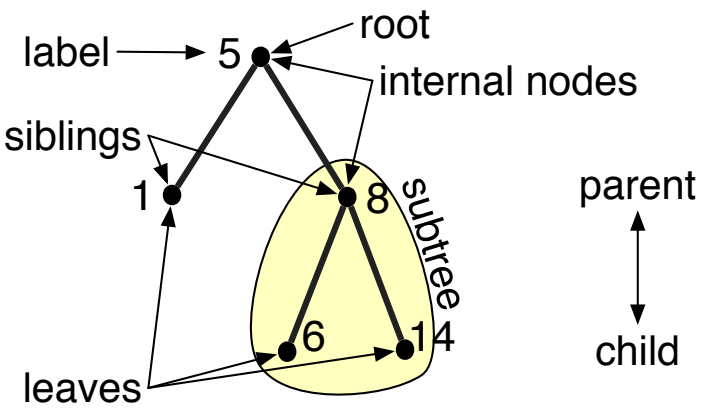


Example: evolution trees

Information related to the evolution of species can also be represented as a tree. This tree shows how species evolved to become new species.



Tree terminology



Characteristics of trees

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

Binary trees

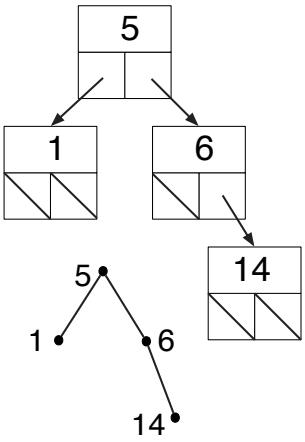
A **binary tree** is a tree with at most two children for each node.

Binary trees are a fundamental part of computer science, independent of what language you use.

Binary arithmetic expression trees and evolution trees are both examples of binary trees.

We'll start with the simplest possible binary tree. It could be used to store a set of natural numbers.

Drawing binary trees



Note: We will consistently use Nats in our binary trees, but it could be a symbol, string, struct, ...

Binary tree data definition

```
(define-struct node (key left right))
;; A Node is a (make-node Nat BT BT)

;; A binary tree (BT) is one of:
;; ⋆ empty
;; ⋆ Node
```

The node's label is called "key" in anticipation of using binary trees to implement dictionaries.

What is the template?

Example functions on a binary tree

Let us fill in the template to make a simple function: count how many nodes in the BT have a key equal to k :

```
;; count-nodes: BT Nat → Nat
(define (count-nodes tree k)
  (cond [(empty? tree) 0]
        [else (+ (cond [(= k (node-key tree)) 1]
                       [else 0])
                 (count-nodes (node-left tree) k)
                 (count-nodes (node-right tree) k))]))
```

Add 1 to every key in a given tree:

```
;; increment: BT → BT
(define (increment tree)
  (cond
    [(empty? tree) empty]
    [else (make-node (add1 (node-key tree))
                     (increment (node-left tree))
                     (increment (node-right tree)))]))
```


Searching binary trees

We are now ready to try to search our binary tree for a given key. It will produce true if it's in the tree and false otherwise.

Our strategy:

- See if the root node contains the key we're looking for. If so, produce true.
- Otherwise, recursively search in the left subtree and in the right subtree. If either recursive search finds the key, produce true. Otherwise, produce false.

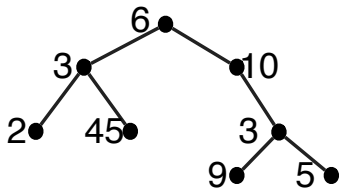
Now we can fill in our BT template to write our search function:

```
;; search-bt: Nat BT → Bool
(define (search-bt k tree)
  (cond [(empty? tree) false]
        [(= k (node-key tree)) true]
        [else (or (search-bt k (node-left tree))
                  (search-bt k (node-right tree)))]))
```

Is this more efficient than searching a list?

Find the path to a key

Write a function, search-bt-path, that searches for an item in the tree. As before, it will return false if the item is not found. However, if it is found search-bt-path will return a list of the symbols 'left and 'right indicating the path from the root to the item.




```
(check-expect (search-bt-path 0 empty) false)
(check-expect (search-bt-path 0 test-tree) false)
(check-expect (search-bt-path 6 test-tree) empty)
(check-expect (search-bt-path 9 test-tree) '(right right left))
(check-expect (search-bt-path 3 test-tree) '(left))
```



```
;; search-bt-path: Nat BT → (anyof false (listof Sym))
(define (search-bt-path k tree)
  (cond
    [(empty? tree) false]
    [(= k (node-key tree)) empty]
    [(list? (search-bt-path k (node-left tree)))
          (cons 'left (search-bt-path k (node-left tree)))]
    [(list? (search-bt-path k (node-right tree)))
          (cons 'right (search-bt-path k (node-right tree)))]
    [else false]))
```

Double calls to search-bt-path. Uggh!

```
;; search-bt-path: Nat BT → (anyof false (listof Sym))
(define (search-bt-path k tree)
  (cond
    [(empty? tree) false]
    [(= k (node-key tree)) empty]
    [else (choose-path (search-bt-path k (node-left tree))
                       (search-bt-path k (node-right tree)))]))

(define (choose-path path1 path2)
  (cond [(list? path1) (cons 'left path1)]
        [(list? path2) (cons 'right path2)]
        [else false]))
```


Binary search trees

We will now make one change that can make searching **much** more efficient. This change will create a a tree structure known as a **binary search tree** (BST).

For any given collection of keys, there is more than one possible tree.

How the keys are placed in a tree can improve the running time of searching the tree when compared to searching the same items in a list.

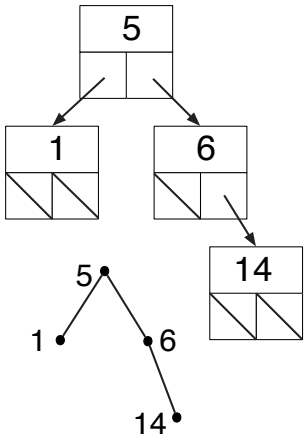

```
;; A Binary Search Tree (BST) is one of:
;; ⋆ empty
;; ⋆ a Node

(define-struct node (key left right))
;; A Node is a (make-node Nat BST BST)
;; requires: key > every key in left BST
;;           key < every key in right BST
```

The BST **ordering property**:

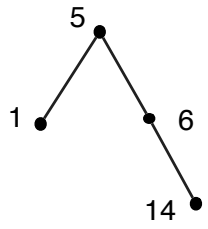
- key is greater than every key in left.
- key is less than every key in right.

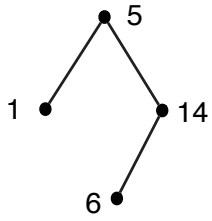
A BST example

```
(make-node 5
  (make-node 1 empty empty)
  (make-node 6
             empty
             (make-node 14
                        empty
                        empty)))
```



There can be several BSTs holding a particular set of keys.





Making use of the ordering property

Main advantage: for certain computations, one of the recursive function applications in the template can always be avoided.

This is more efficient (sometimes considerably so).

In the following slides, we will demonstrate this advantage for searching and adding.

We will write the code for searching, and briefly sketch adding, leaving you to write the Racket code.

Searching in a BST

How do we search for a key n in a BST?

We reason using the data definition of BST.

If the BST is empty, then n is not in the BST.

If the BST is of the form (make-node k l r), and k equals n, then we have found it.

Otherwise it might be in either of the trees l, r.

If n $<$ k, then n must be in l if it is present at all, and we only need to recursively search in l.

If n $>$ k, then n must be in r if it is present at all, and we only need to recursively search in r.

Either way, we save one recursive function application.


```
;; (search-bst k t) produces true if k is in t; false otherwise.
;; search-bst: Nat BST → Bool
(define (search-bst k t)
  (cond[(empty? t) false]
       [(= k (node-key t)) true]
       [(< k (node-key t)) (search-bst k (node-left t))]
       [(> k (node-key t)) (search-bst k (node-right t))]))
```

Adding to a BST

How do we add a new key, k, to a BST t?

If t is empty, then the result is a BST with only one node.

Otherwise t is of the form (make-node n l r).
If k = n, the key is already in the tree and we can simply return t.

If $k < n$, then the new key must be added to l, and if $k > n$, then the pair must be added to r. Again, we need only make one recursive function application.

Creating a BST from a list

How do we create a BST from a list of keys?

We reason using the data definition of a list.

If the list is empty, the BST is empty.

If the list is of the form (cons k lst), we add the key k to the BST created from the list lst. The first key is inserted **last**.

It is also possible to write a function that inserts keys in the opposite order.

Binary search trees in practice

If the BST has all left subtrees empty, it looks and behaves like a sorted list, and the advantage is lost.

In later courses, you will see ways to keep a BST "balanced" so that "most" nodes have nonempty left and right children. We will also cover better ways to analyze the efficiency of algorithms and operations on data structures.

Augmenting trees

So far nodes have been (define-struct node (key left right)).

We can **augment** the node with additional data:
(define-struct node (key val left right)).

- The name val is arbitrary – choose any name you like.
- The type of val is also arbitrary: could be a number, string, structure, etc.
- You could augment with multiple values.
- The set of keys remains unique; could have duplicate values.

BST dictionaries

An **augmented BST** can serve as a dictionary that can perform significantly better than an association list.

Recall from Module 08 that a dictionary stores a set of (key, value) pairs, with at most one occurrence of any key. A dictionary supports lookup, add, and remove operations.

We implemented dictionaries using an association list, a list of two-element lists. Seach could be inefficient for large lists.

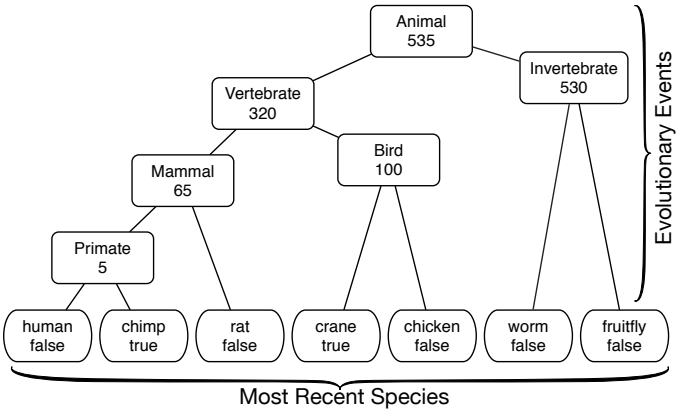
We need to modify node to include the value associated with the key. search needs to return the associated value, if found.


```
(define-struct node (key val left right))
;; A binary search tree dictionary (BSTD) is either
;; empty or (make-node Nat Str BSTD BSTD)

;; (search-bst-dict k t) produces the value associated with k
;;        if k is in t; false otherwise.
;; search-bst: Nat BSTD → anyof(Str false)
(define (search-bst-dict k t)
  (cond[(empty? t) false]
       [(= k (node-key t)) (node-val t)]
       [(< k (node-key t)) (search-bst-dict k (node-left t))]
       [(> k (node-key t)) (search-bst-dict k (node-right t))]))
```

```
(define test-tree (make-node 5 "Susan"
                             (make-node 1 "Juan" empty empty)
                             (make-node 14 "David"
                                        (make-node 6 "Lucy" empty
                                        empty)))

(check-expect (search-bst-dict 5 empty) false)
(check-expect (search-bst-dict 5 test-tree) "Susan")
(check-expect (search-bst-dict 6 test-tree) "Lucy")
(check-expect (search-bst-dict 2 test-tree) false)
```


Evolutionary trees

Evolutionary trees are another kind of **augmented tree**.



Evolutionary trees are binary trees that show the evolutionary relationships between species. Biologists believe that all life on Earth is part of a single evolutionary tree, indicating common ancestry.

Internal nodes represent an **evolutionary event** when a common ancestor species split into two new species. Internal nodes are augmented with the common ancestor species name and an estimate of how long ago the evolutionary event took place (in millions of years).

Leaves represent a **most recent species**. They are augmented with a name and whether the species is endangered.

The fine print

We've simplified a lot...

- The correct terms are "phylogenetic tree" and "speciation event". Nodes are often called "taxonomic units". This is an active area of research; see Wikipedia on "phylogenetic tree".
- Evolutionary trees are built with incomplete data and theories, so there could be many different evolution trees.
- Leaves could represent extinct species that died off before splitting. Hence the term "most recent species".

Representing evolutionary trees

Internal nodes each have exactly two children. Each internal node has the name of the common ancestor species and the estimated date of the evolutionary event.

Leaves have names and endangerment status of the most recent species.

The order of children does not matter.

The structure of the tree is dictated by a hypothesis about evolution.

Data definitions for evolutionary trees

```
;; An EvoTree (Evolution Tree) is one of:
;; ⋆ a RSpecies (recent species)
;; ⋆ a EvoEvent (evolutionary event)

(define-struct rspecies (name endangered))
;; A RSpecies is a (make-rspecies Str Bool)
(define-struct evoevent (name age left right))
;; A EvoEvent is a (make-evoevent Str Num EvoTree EvoTree)
```

Note that the EvoEvent data definition uses a pair of EvoTrees.

Constructing the example evolutionary tree

```
(define-struct rspecies (name endangered))
(define-struct evoevent (name age left right))

(define human (make-rspecies "human" false))
(define chimp (make-rspecies "chimp" true))
(define rat (make-rspecies "rat" false))
(define crane (make-rspecies "crane" true))
(define chicken (make-rspecies "chicken" false))
(define worm (make-rspecies "worm" false))
(define fruit-fly (make-rspecies "fruit fly" false))
```



```
(define primate (make-evoevent "Primate" 5 human chimp))
(define mammal (make-evoevent "Mammal" 65 primate rat))
(define bird (make-evoevent "Bird" 100 crane chicken))
(define vertebrate
  (make-evoevent "Vertebrate" 320 mammal bird))
(define invertebrate
  (make-evoevent "Invertebrate" 530 worm fruit-fly))
(define animal
  (make-evoevent "Animal" 535 vertebrate invertebrate))
```


Derive the EvoTree template from the data definition.

```
;; evotree-template: EvoTree → Any
(define (evotree-template t)
  (cond [(rspecies? t) (rspecies-template t)]
        [(evoevent? t) (evoevent-template t)]))
```

This is a straightforward implementation based on the data definition. It's also a good strategy to take a complicated problem (dealing with an EvoTree) and decompose it into simpler problems (dealing with a RSpecies or an EvoEvent).

Functions for these two data definitions are on the next slide.

```
;; rspecies-template: RSpecies → Any
(define (rspecies-template rs)
  (... (rspecies-name rs) ...
       (rspecies-endangered rs) ...))

;; evoevent-template: EvoEvent → Any
(define (evoevent-template ee)
  (... (evoevent-name ee) ...
       (evoevent-age ee) ...
       (evoevent-left ee) ...
       (evoevent-right ee) ...))
```


We know that (evoevent-left ee) and (evoevent-right ee) are EvoTrees, so apply the EvoTree-processing function to them.

```
;; evoevent-template: EvoEvent → Any
(define (evoevent-template ee)
  (... (evoevent-name ee) ...
       (evoevent-age ee) ...
       (evotree-template (evoevent-left ee)) ...
       (evotree-template (evoevent-right ee)) ...))
```

evoevent-template uses evotree-template and evotree-template uses evoevent-template. This is called **mutual recursion**.

A function on EvoTrees

This function counts the number of recent species within an evotree.

```
;; (count-species t): Counts the number of recent species
;;       (leaves) in the EvoTree t.
;; count-species: EvoTree → Nat
(define (count-species t)
  (cond [(rspecies? t) (count-recent t)]
        [(evoevent? t) (count-evoevent t)]))

(check-expect (count-species animal) 7)
(check-expect (count-species human) 1)
```

```
;; count-recent RSpecies → Nat
(define (count-recent t)
  1)

;; count-evoevent EvoEvent → Nat
(define (count-evoevent t)
  (+ (count-species (evoevent-left t))
     (count-species (evoevent-right t))))
```


Traversing a tree

A **tree traversal** refers to the process of visiting each node in a tree exactly once. The increment example from binary trees is one example of a traversal.

We'll now traverse an EvoTree to produce a list of all the names it contains.

We'll solve this problem two different ways: using append and using accumulative recursion.


```
;; list-names: EvoTree → listof(Str)
(define (list-names t)
  (cond [(rspecies? t) (list-rs-names t)]
        [(evoevent? t) (list-ee-names t)]))

;; list-rs-names: RSpecies → listof(Str)
(define (list-rs-names rs)
  (... (rspecies-name rs) ...))
;; list-ca-names: EvoEvent → listof(Str)
(define (list-ee-names ee)
  (... (evoevent-name ee) ...
       (list-names (evoevent-left ee)) ...
       (list-names (evoevent-right ee)) ...))
```

list-names with an accumulator

```
;; list-names: EvoTree → listof(String)
(define (list-names t)
  (list-names/acc t empty))

;; list-names/acc: EvoTree listof(String) → listof(String)
(define (list-names/acc t names)
  (cond [(rspecies? t) (list-rs-names t names)]
        [(evoevent? t) (list-ee-names t names)]))
```



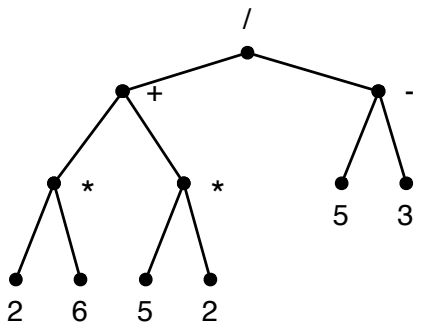
```
;; list-rs-names: RSpecies listof(String) → listof(Str)
(define (list-rs-names rs names)
  (cons (rspecies-name rs) names))

;; list-ee-names: EvoEvent listof(String) → listof(Str)
(define (list-ee-names ee names)
  (cons (evoevent-name ee)
        (list-names/acc (evoevent-left ee)
                        (list-names/acc (evoevent-right ee) names))))

(check-expect (list-names human) '("human"))
(check-expect (list-names mammal) '("Mammal" "Primate" "human"
```


Practice problems with EvoTrees

- Count the number of evolutionary events (internal nodes) with and age less than n. For example, the sample tree has 4 events that are less than 400 million years old.
- Count the number of evolutionary events that occurred to produce a given recent species.
- Find the evolutionary path between the root of a (sub)tree and a recent species. For example, the path from animal to rat is '(animal vertebrate mammal rat).
- Modify list-names to produce the names of endangered species.

Binary expression trees

The expression $((2 * 6) + (5 * 2))/(5 - 3)$ can be represented as a **tree**:



Representing binary arithmetic expressions

Internal nodes each have exactly two children.

Leaves have number labels.

Internal nodes have symbol labels.

We care about the order of children.

The structure of the tree is dictated by the expression.

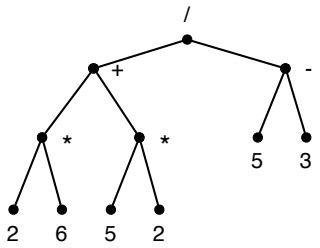

```
(define-struct binode (op left right))
;; A Binary arithmetic expression Internal Node (BINode)
;;    is a (make-binode (anyof '* '+ '/ '-) BinExp BinExp)

;; A binary arithmetic expression (BinExp) is one of:
;; * a Num
;; * a BINode
```

Some examples of binary arithmetic expressions:

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

A more complex example:

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



Templates for binary arithmetic expressions

```
;; binexp-template: BinExp → Any
(define (binexp-template ex)
  (cond [(number? ex) ... ex ...]
        [(binode? ex) (binode-template ex)]))

;; binode-template: BINode → Any
(define (binode-template node)
  (... (binode-op node) ...
       (binexp-template (binode-left node)) ...
       (binexp-template (binode-right node)) ...))
```


Evaluating expressions

```
;; (eval ex) evaluates the expression ex and produces its value.
;; eval: BinExp → Num
(check-expect (eval 5) 5)
(check-expect (eval (make-binode '+ 2 5)) 7)
(check-expect (eval (make-binode '/ (make-binode '- 10 2)
                                    (make-binode '+ 2 2))) 2)

(define (eval ex)
  (cond [(number? ex) ex]
        [(binode? ex) (eval-binode ex)]))
```

```
;; (eval-binode node) evaluates the node.
;; eval-binode BINode → Num
(define (eval-binode node)
  (cond [(symbol=? '* (binode-op node))
         (* (eval (binode-left node)) (eval (binode-right node)))]
        [(symbol=? '/ (binode-op node))
         (/ (eval (binode-left node)) (eval (binode-right node)))]
        [(symbol=? '+ (binode-op node))
         (+ (eval (binode-left node)) (eval (binode-right node)))]
        [(symbol=? '- (binode-op node))
         (- (eval (binode-left node)) (eval (binode-right node)))]))
```


Eval, refactored

```
(define (eval ex)
  (cond [(number? ex) ex]
        [(binode? ex) (eval-binode (binode-op ex)
                                   (eval (binode-left ex))
                                   (eval (binode-right ex)))]))

(define (eval-binode op left right)
  (cond [(symbol=? op '*) (* left right)]
        [(symbol=? op '/) (/ left right)]
        [(symbol=? op '+) (+ left right)]
        [(symbol=? op '-) (- left right)]))
```


General trees

Binary trees can be used for a large variety of application areas.

One limitation is the restriction on the number of children.

How might we represent a node that can have up to three children?

What if there can be any number of children?

General arithmetic expressions

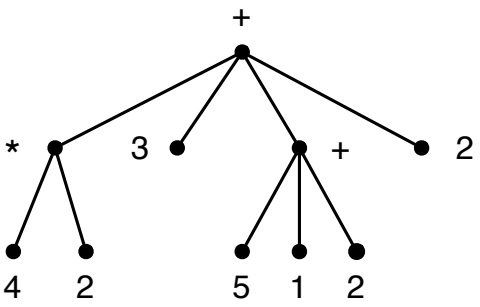
For binary arithmetic expressions, we formed binary trees.

Racket expressions using the functions $+$ and $*$ can have an unbounded number of arguments.

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

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

We can visualize an arithmetic expression as a general tree.



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

For a binary arithmetic expression, we defined a structure with three fields: the operation, the first argument, and the second argument.

For a general arithmetic expression, we define a structure with two fields: the operation and a list of arguments (which is a list of arithmetic expressions).

```
(define-struct ainode (op args))
;; a Arithmetic expression Internal Node (AINode)
;;      is a (make-ainode (anyof '* '+) (listof AExp))

;; An Arithmetic Expression (AExp) is one of:
;; * a Num
;; * an AINode
```

Each definition depends on the other, and each template will depend on the other. Examples: An EvoTree was defined in terms of RSpecies and EvoEvent. EvoEvent was defined in terms of EvoTree. BINode and BinExp depend on each other.

Examples of arithmetic expressions:

```
3
(make-ainode '+ (list 3 4))
(make-ainode '* (list 3 4))
(make-ainode '+ (list (make-ainode '* '(4 2))
                                    3
                      (make-ainode '+ '(5 1 2))
                                    2))
(make-ainode '+ (list))
```


Templates for arithmetic expressions

```
;; aexp-template: AExp → Any
(define (aexp-template ex)
  (cond [(number? ex) ...]
        [(ainode? ex) (... (ainode-op ex)
                           (listof-aexp-template (ainode-args ex)))]))

;; listof-aexp-template: (listof AExp) → Any
(define (listof-aexp-template args)
  (cond [(empty? args) ...]
        [else (... (aexp-template (first args)) ...
                   (listof-aexp-template (rest args)) ...)]))
```

The function eval

```
;; (eval ex) evaluates the arithmetic expression ex.
;; eval: AExp → Num
(check-expect (eval 3) 3)
(check-expect (eval (make-ainode '+ (list 3 4))) 7)
(check-expect (eval (make-ainode '+ ())) 0)

(define (eval ex)
  (cond [(number? ex) ex]
        [(ainode? ex) (apply (ainode-op ex)
                             (ainode-args ex))]))
```



```
;; (apply op exlist) applies op to the list of arguments.
;; apply: op (listof AExp) → Num
(define (apply op args)
  (cond [(empty? args) (cond [(symbol=? op '+) 0]
                             [(symbol=? op '*) 1])]
        [(symbol=? op '+)  (+ (eval (first args))
                              (apply op (rest args)))]
        [(symbol=? op '*)  (* (eval (first args))
                              (apply op (rest args)))]))
```


Condensed trace of aexp evaluation

```
(eval (make-ainode '+ (list (make-ainode '* '(3 4))
                            (make-ainode '* '(2 5)))))
⇒ (apply '+ (list (make-ainode '* '(3 4))
                  (make-ainode '* '(2 5))))
⇒ (+ (eval (make-ainode '* '(3 4)))
     (apply '+ (list (make-ainode '* '(2 5)))))
⇒ (+ (apply '* '(3 4))
     (apply '+ (list (make-ainode '* '(2 5)))))
```

⇒ (+ (∗ (eval 3) (apply '∗ '(4)))
(apply '+ (list (make-ainode '∗ '(2 5)))))
⇒ (+ (∗ 3 (apply '∗ '(4)))
(apply '+ (list (make-ainode '∗ '(2 5)))))
⇒ (+ (∗ 3 (∗ (eval 4) (apply '∗ empty)))
(apply '+ (list (make-ainode '∗ '(2 5)))))
⇒ (+ (∗ 3 (∗ 4 (apply '∗ empty)))
(apply '+ (list (make-ainode '∗ '(2 5)))))

⇒ (+ (∗ 3 (∗ 4 1))
(apply '+ (list (make-ainode '∗ '(2 5)))))
⇒ (+ 12
(apply '+ (list (make-ainode '∗ '(2 5)))))
⇒ (+ 12 (+ (eval (make-ainode '∗ '(2 5)))
(apply '+ empty)))
⇒ (+ 12 (+ (apply '∗ '(2 5))
(apply '+ empty)))
⇒ (+ 12 (+ (∗ (eval 2) (apply '∗ '(5)))
(apply '+ empty)))

⇒ (+ 12 (+ (∗ 2 (apply '∗ '(5)))
(apply '+ empty)))
⇒ (+ 12 (+ (∗ 2 (∗ (eval 5) (apply '∗ empty)))
(apply '+ empty)))
⇒ (+ 12 (+ (∗ 2 (∗ 5 (apply '∗ empty)))
(apply '+ empty)))
⇒ (+ 12 (+ (∗ 2 (∗ 5 1))
(apply '+ empty)))
⇒ (+ 12 (+ (∗ 2 5) (apply '+ empty)))
⇒ (+ 12 (+ 10 (apply '+ empty)))
⇒ (+ 12 (+ 10 0)) ⇒ (+ 12 10) ⇒ 22

Alternate data definition

In Module 6, we saw how a list could be used instead of a structure holding tax record information.

Here we could use a similar idea to replace the structure ainode and the data definitions for AExp.


```
;; An alternate arithmetic expression (AltAExp) is one of:
;; ⋆ a Num
;; ⋆ (cons (anyof '* '+) (listof AltAExp))
```

Each expression is a list consisting of a symbol (the operation) and a list of expressions.

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


Templates: AltAExp and (listof AltAExp)

```
(define (altaexp-template ex)
  (cond [(number? ex) ...]
        [else (... (first ex) ...
                   (listof-altaexp-template (rest ex)) ...)]))

(define (listof-altaexp-template exlist)
  (cond [(empty? exlist) ...]
        [(cons? exlist) (... (altaexp-template (first exlist)) ...
                             (listof-altaexp-template (rest exlist)) ...)]))
```

```
;; eval: AltAExp → Num

(define (eval aax)
  (cond [(number? aax) aax]
        [else (apply (first aax) (rest aax))]))
```



```
;; apply: Sym (listof AltAExp) → Num

(define (apply f aaxl)
  (cond [(and (empty? aaxl) (symbol=? f '*)) 1]
        [(and (empty? aaxl) (symbol=? f '+)) 0]
        [(symbol=? f '*)
         (* (eval (first aaxl)) (apply f (rest aaxl)))]
        [(symbol=? f '+)
         (+ (eval (first aaxl)) (apply f (rest aaxl)))]))
```


A condensed trace

```
(eval '(* (+ 1 2) 4))
⇒ (apply '* '((+ 1 2) 4))
⇒ (* (eval '(+ 1 2)) (apply '* '(4)))
⇒ (* (apply '+ '(1 2)) (apply '* '(4)))
⇒ (* (+ 1 (apply '+ '(2))) (apply '* '(4)))
⇒ (* (+ 1 (+ 2 (apply '+ '())))) (apply '* '(4)))
⇒ (* (+ 1 (+ 2 0)) (apply '* '(4)))
```

⇒ (∗ (+ 1 2) (apply ’∗ ’(4)))

⇒ (∗ 3 (apply ’∗ ’(4)))

⇒ (∗ 3 (∗ 4 (apply ’∗ ’())))

⇒ (∗ 3 (∗ 4 1))

⇒ (∗ 3 4)

⇒ 12

Structuring data using mutual recursion

Mutual recursion arises when complex relationships among data result in cross references between data definitions.

The number of data definitions can be greater than two.

Structures and lists may also be used.

In each case:

- create templates from the data definitions and
- create one function for each template.

Other uses of general trees

We can generalize from allowing only two arithmetic operations and numbers to allowing arbitrary functions and variables.

In effect, we have the beginnings of a Racket interpreter.

But beyond this, the type of processing we have done on arithmetic expressions can be applied to tagged hierarchical data, of which a Racket expression is just one example.

Organized text and Web pages provide other examples.

```
'(chapter
   (section
     (paragraph "This is the first sentence."
                "This is the second sentence.")
     (paragraph "We can continue in this manner."))
   (section ...)
   ...
)
```



```
'(webpage
   (title "CS 135: Designing Functional Programs")
   (paragraph "For a course description,"
              (link "click here." "desc.html")
              "Enjoy the course!")
   (horizontal-line)
   (paragraph "(Last modified yesterday.)"))
```


Nested lists

We have discussed flat lists (no nesting):

```
'( a 1 "hello" x)
```

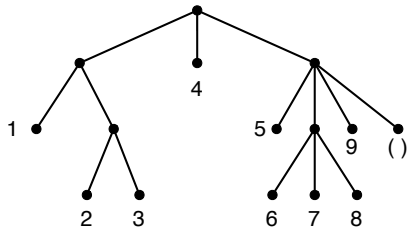
and lists of lists (one level of nesting):

```
'((1 "a") (2 "b"))
```

We now consider **nested lists** (arbitrary nesting):

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

It is often useful to visualize a nested list as a tree, in which the leaves correspond to the elements of the list, and the internal nodes indicate the nesting:




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


Examples of nested lists:

```
empty
'(4 2)
'((4 2) 3 (4 1 6))
'((3) 2 () (4 (3 6)))
```

Each nonempty tree is a list of subtrees.

The first subtree in the list is either

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

Data definition for nested lists

```
;; A nested list of numbers (Nest-List-Num) is one of:
;; ⋆ empty
;; ⋆ (cons Num Nest-List-Num)
;; ⋆ (cons Nest-List-Num Nest-List-Num)
```

This can be generalized to generic types: (Nest-List-X)

Template for nested lists

The template follows from the data definition.

```
nest-lst-template: Nest-List-Num → Any
(define (nest-lst-template lst)
  (cond [(empty? lst) ...]
        [(number? (first lst))
         (... (first lst) ... (nest-lst-template (rest lst)) ...)]
        [else
         (... (nest-lst-template (first lst)) ...
              (nest-lst-template (rest lst)) ...)]))
```


The function count-items

```
;; count-items: Nest-List-Num → Nat
(define (count-items nln)
  (cond [(empty? nln) 0]
        [(number? (first nln))
         (+ 1 (count-items (rest nln)))]
        [else (+ (count-items (first nln))
                 (count-items (rest nln)))]))
```


Condensed trace of count-items

```
(count-items '((10 20) 30))
⇒ (+ (count-items '(10 20)) (count-items '(30)))
⇒ (+ (+ 1 (count-items '(20))) (count-items '(30)))
⇒ (+ (+ 1 (+ 1 (count-items '()))) (count-items '(30)))
⇒ (+ (+ 1 (+ 1 0)) (count-items '(30)))
⇒ (+ (+ 1 1) (count-items '(30)))
⇒ (+ 2 (count-items '(30)))
⇒ (+ 2 (+ 1 (count-items '())))
⇒ (+ 2 (+ 1 0)) ⇒ (+ 2 1) ⇒ 3
```

Flattening a nested list

flatten produces a flat list from a nested list.

```
;; (flatten lst) produces a single-level list with all the
;;       elements of lst.
;; flatten: Nest-List-Num → (listof Num)
(check-expect (flatten '(1 2 3)) '(1 2 3))
(check-expect (flatten '((1 2 3) (a b c))) '(1 2 3 a b c))
(define (flatten lst) ... )
```

We make use of the built-in Racket function append.

(append '(1 2) '(3 4)) ⇒ '(1 2 3 4)


```
;; flatten: Nest-List-Num → (listof Num)
(define (flatten lst)
  (cond [(empty? lst) empty]
        [(number? (first lst))
         (cons (first lst) (flatten (rest lst)))]
        [else (append (flatten (first lst))
                      (flatten (rest lst)))]))
```


Condensed trace of flatten

```
(flatten '((10 20) 30))
⇒ (append (flatten '(10 20)) (flatten '(30)))
⇒ (append (cons 10 (flatten '(20))) (flatten '(30)))
⇒ (append (cons 10 (cons 20 (flatten '()))) (flatten '(30)))
⇒ (append (cons 10 (cons 20 empty)) (flatten '(30)))
⇒ (append (cons 10 (cons 20 empty)) (cons 30 (flatten '())))
⇒ (append (cons 10 (cons 20 empty)) (cons 30 empty))
⇒ (cons 10 (cons 20 (cons 30 empty)))
```

Goals of this module

You should be familiar with tree terminology.

You should understand the data definitions for binary trees, binary search trees, evolutionary trees, and binary arithmetic expressions.

You should understand how the templates are derived from those definitions, and how to use the templates to write functions that consume those types of data.

You should understand the definition of a binary search tree and its ordering property.

You should be able to write functions which consume binary search trees, including those sketched (but not developed fully) in lecture.

You should be able to develop and use templates for other binary trees, not necessarily presented in lecture.

You should understand the idea of mutual recursion for both examples given in lecture and new ones that might be introduced in lab, assignments, or exams.

You should be able to develop templates from mutually recursive data definitions, and to write functions using the templates.