Trees

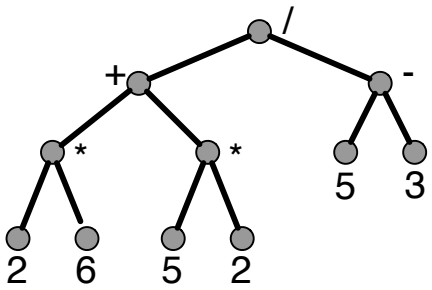
Readings: HtDP, sections 14, 15, 16.

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

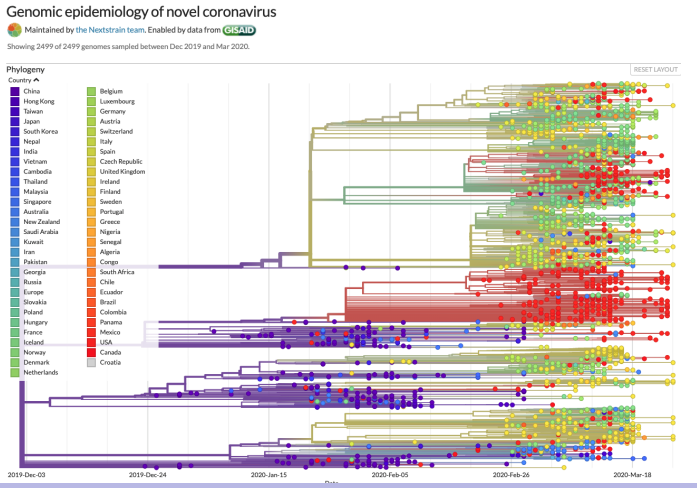
The expression $((2 * 6) + (5 * 2))/(5 - 3)$ can be represented as a **tree**:



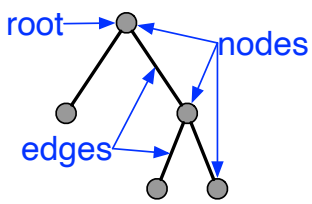
> Example: Phylogentic trees

This phylogentic tree tracks the evolution of COVID-19 in the first four months of the recent pandemic.

Image: `nextstrain.org/ncov`



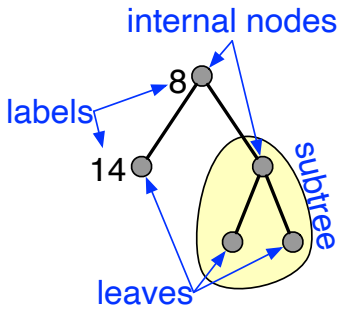
> Tree terminology (1/2)



A **tree** is a set of **nodes** and **edges** where an edge connects two distinct nodes.
A tree has three requirements:

- One node is identified as the **root**.
- Every node *c* other than the root is connected by an edge to some other node *p*. *p* is called the **parent** and *c* is called the **child**.
- A tree is **connected**:
  for every node *n* other than the root,
  the parent of *n* or
  the parent of the parent of *n* or
  the parent of the parent ... of *n* will be the root.

> Tree terminology (2/2)



Other useful terms:

- **leaves**: nodes with no children
- **internal nodes**: nodes that have children
- **labels**: data attached to a node
- **ancestors** of node *n*: *n* itself, the parent of *n*, the parent of the parent of *n*, etc. up to the root
- **descendents** of *n*: all the nodes that have *n* as an ancestor
- **subtree** rooted at *n*: *n* and all of its descendants

> 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

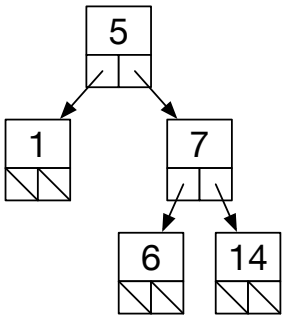
A **binary tree** is a tree with at most two children for each node.

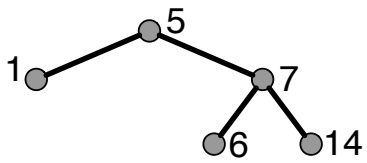
Binary trees are a fundamental part of computer science, independent of what language you use.

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 `Nat`s 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?

Aside: Tips for building templates

Follow the data definition. For each part of the data definition, if it

- is a defined data type, apply that type's template.
- says "one of" or is mixed data, include a **`cond`** to distinguish the cases.
- is compound data (a structure), extract each of the fields, in order.
- is a list, extract the `first` and `rest` of the list.

Add elipses (. . .) around each of the above.

Apply the above recursively.

> Binary tree template

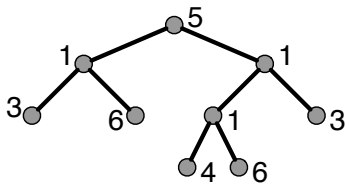
```
;; bt-template:  BT → Any
(define (bt-template t)
```


> Example: sum-keys

```
;; (sum-keys bt) sums the keys in the binary tree bt.
;; sum-keys: BT → Nat

(check-expect (sum-keys empty) 0)
(check-expect (sum-keys (make-node 10 empty empty)) 10)
(check-expect (sum-keys (make-node 10
                                   (make-node 5 empty empty)
                                   empty)) 15)

(define (sum-keys bt)
```

> Example: count-nodes

```
(define test-tree (count-nodes (make-node 5 (make-node 1 ...
                                             (make-node 1 ...)))))
;; (count-nodes tree k) counts the number of nodes in the tree that
;;    have a key equal to k.
;; count-nodes: BT Nat → Nat
(check-expect (count-nodes empty 5) 0)
(check-expect (count-nodes test-tree 1) 3)

(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))]))
```



> Example: Increment keys

```
;; (increment tree) adds 1 to each key in the 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 search our binary tree for a given key. It will produce `true` if the key is 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`.

> Searching binary trees

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

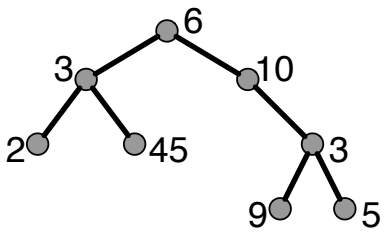
```
;; (search k tree) produces true if k is in tree; false otherwise.
;; search: Nat BT → Bool
(define (search-bt k tree)
  (cond [(empty? tree) false]
        [(= k (node-key tree)) true]
        [else (or (search-bt k (node-left tree))
                  (search-bt k (node-right tree)))]))
```

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.

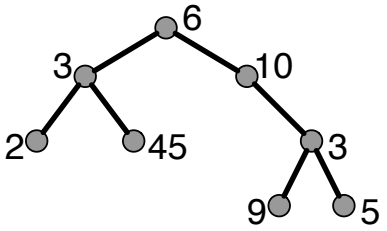
If the tree contains a duplicate, produce the path to the left-most item.



The path from 6 to 9 is `'(right right left)`.

> Find the path to a key

```
(define test-tree
   (make-node 6 (make-node 3 ...)
                (make-node 10 ...)))
```




```
(check-expect (search-bt-path-v1 0 empty) false)
(check-expect (search-bt-path-v1 6 test-tree) empty)
(check-expect (search-bt-path-v1 3 test-tree) '(left))
(check-expect (search-bt-path-v1 9 test-tree) '(right right left))
(check-expect (search-bt-path-v1 0 test-tree) false)
```

> search-bt-path

```
;; search-bt-path-v1: Nat BT → (anyof false (listof Sym))
(define (search-bt-path-v1 k tree)
  (cond
    [(empty? tree) false]
    [(= k (node-key tree)) '()]
    [(list? (search-bt-path-v1 k (node-left tree)))
     (cons 'left (search-bt-path-v1 k (node-left tree)))]
    [(list? (search-bt-path-v1 k (node-right tree)))
     (cons 'right (search-bt-path-v1 k (node-right tree)))]
    [else false]))
```

Double calls to `search-bt-path`. Uggh!

> Improved search-bt-path

```
;; search-bt-path-v2: Nat BT → (anyof false (listof Sym))
(define (search-bt-path-v2 k tree)
  (cond
    [(empty? tree) false]
    [(= k (node-key tree)) '()]
    [else (choose-path-v2 (search-bt-path-v2 k (node-left tree))
                          (search-bt-path-v2 k (node-right tree)))]))

(define (choose-path-v2 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
```

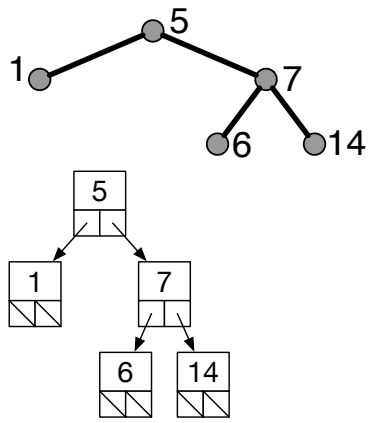
The BST **ordering property**:

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

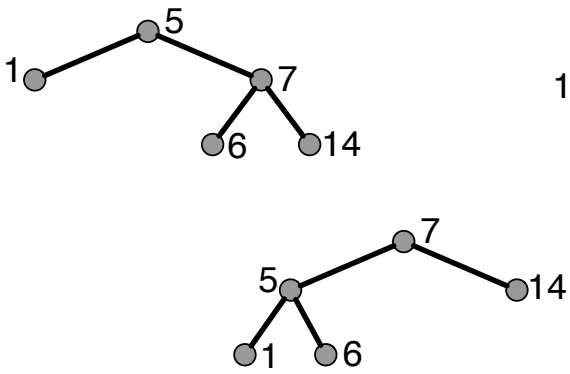
Note: the ordering property holds in every subtree

> A BST example

```
(make-node 5
  (make-node 1 empty empty)
  (make-node 7
             (make-node 6 empty 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.

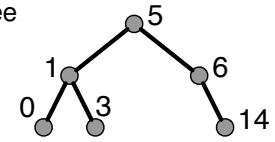
We will write the code for searching, and briefly sketch adding, leaving you to write the Racket code.

> Searching in a BST

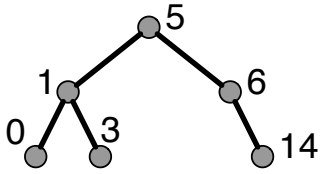
How do we search for a key `n` in a BST? We reason using the data definition of `BST`.

- If the BST is `empty`, then `n` is not in the BST.
- If the BST is of the form `(make-node k left right)`, and `k` equals `n`, then we have found it.
- Otherwise it might be in either the `left` or `right` subtree
  - If `n` $<$ `k`, then `n` must be in `left` if it is present at all, and we only need to recursively search in `left`.
  - If `n` $>$ `k`, then `n` must be in `right` if it is present at all, and we only need to recursively search in `right`.

  Either way, we save one recursive function application.




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



> Adding to a BST

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

Reasoning from the data definition for a BST:

- If `t` is `empty`, then the result is a BST with only one node containing `n`.
- If `t` is of the form `(make-node k left right)` and `k` = `n`, the key is already in the tree and we can simply produce `t`.
- Otherwise, `n` must go in either the `left` or `right` subtree.
  - If `n` $<$ `k`, then the new key must be added to `left`.
  - If `n` $>$ `k`, then the new key must be added to `right`.

  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 in the list 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.
- The tree 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.

> search-bst-dict

```
(define-struct node (key val left right))
;; A binary search tree dictionary (BSTD) is either:
;; * empty
;; * (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-dict: 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))]))
```

> search-bst-dict tests

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

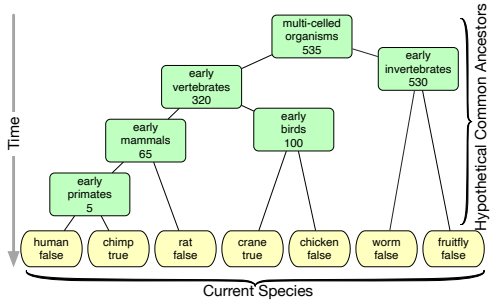
Leaves represent a **current species**. They are augmented with a name and whether the species is endangered.

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

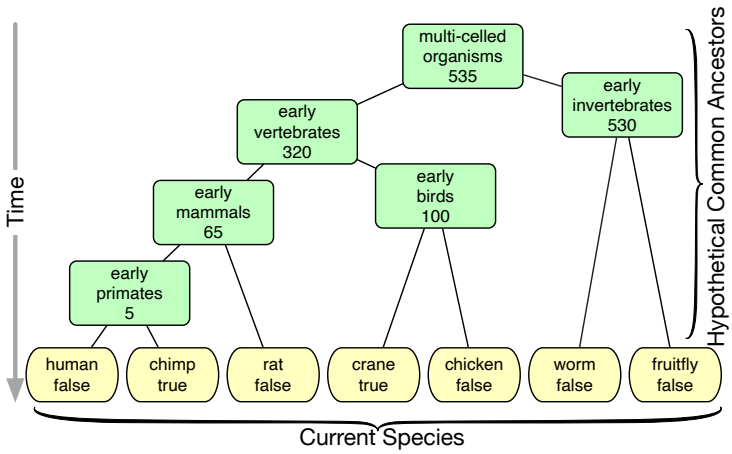
Evolutionary trees

Evolutionary trees are constructed by evolutionary biologists.

- Start with current species.
- Based on common attributes (including DNA sequences), hypothesize common ancestor species.
- Keep going with more and more common ancestor species,
- back to a single common ancestor (the root).



> A very incomplete, sample evolutionary tree



> The fine print

We've simplified a lot...

- The correct term is **phylogenetic tree**.
- Evolutionary trees are built with incomplete data and theories, so biologists have created many different trees.
- Leaves could represent species that became extinct before splitting into current species. We're going to ignore that possibility.

This is an active area of research; see Wikipedia on "phylogenetic tree". UWaterloo has a CS research group that works on these problems (including a tool to build these trees). See `https://uwaterloo.ca/bioinformatics-group/`.

» Representing evolutionary trees

Internal nodes each have exactly two children. Each internal node has:

- the name of the common ancestor species
- how long ago the common ancestor split into two new species
- the two species that resulted from the split

Leaves have:

- the name of the current species
- the endangerment status (`true` if endangered; `false` otherwise)

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 (Evolutionary Tree) is one of:
;; * a Current (current species)
;; * an Ancestor (common ancestor species)

(define-struct current (name endangered))
;; A Current is a (make-current Str Bool)
(define-struct ancestor (name age left right))
;; An Ancestor is a (make-ancestor Str Num EvoTree EvoTree)
```

Note that the `Ancestor` data definition uses a pair of `EvoTree`s.

» Constructing the example evolutionary tree (1/2)

```
(define-struct current (name endangered))
;; A Current is a (make-current Str Bool)
(define-struct ancestor (name age left right))
;; An Ancestor is a (make-ancestor Str Num EvoTree EvoTree)

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


» Constructing the example evolutionary tree (2/2)

```
(define e-primates (make-ancestor "early primates" 5 human chimp))
(define e-mammals (make-ancestor "early mammals" 65 e-primates rat))
(define e-birds (make-ancestor "early birds" 100 crane chicken))
(define e-vertebrates
  (make-ancestor "early vertibrates" 320 e-mammals e-birds))
(define e-invertebrates
  (make-ancestor "early invertibrates" 530 worm fruit-fly))
(define mco
  (make-ancestor "multi-celled organisms"
                 535 e-vertebrates e-invertebrates))
```

> EvoTree Template (1/3)

```
;; An EvoTree (Evolutionary Tree) is one of:
;; * a Current (current species)
;; * an Ancestor (common ancestor species)

(define-struct current (name endangered))
;; A Current is a (make-current Str Bool)
(define-struct ancestor (name age left right))
;; An Ancestor is a (make-ancestor Str Num EvoTree EvoTree)


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


» EvoTree Template (2/3)

```
;; current-template: Current → Any
(define (current-template cs)
  (... (current-name cs) ...
       (current-endangered cs) ...))

;; ancestor-template: Ancestor → Any
(define (ancestor-template as)
  (... (ancestor-name as) ...
       (ancestor-age as) ...
       (ancestor-left as) ...
       (ancestor-right as) ...))
```

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 `Current` or an `Ancestor`).

» EvoTree Template (3/3)

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

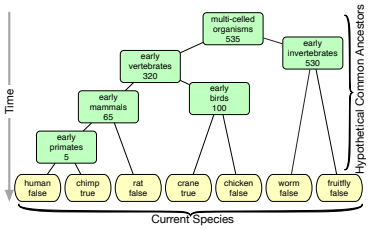
```
;; ancestor-template:  Ancestor → Any
(define (ancestor-template as)
  (... (ancestor-name as) ...
       (ancestor-age as) ...
       (evotree-template (ancestor-left as)) ...
       (evotree-template (ancestor-right as)) ...))
```

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

> A function on EvoTrees (1/2)

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

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

(check-expect (count-current-species mco) 7)
(check-expect (count-current-species human) 1)
```



» A function on EvoTrees (2/2)

```
;; count-current Current → Nat
(define (count-current t)
  1)

;; count-ancestor Ancestor → Nat
(define (count-ancestor t)
  (+ (count-current-species (ancestor-left t))
     (count-current-species (ancestor-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

```
;; list-names: EvoTree → (listof Str)
(define (list-names t)
  (cond [(current? t) (list-cnames t)]
        [(ancestor? t) (list-anames t)]))

;; list-cnames: Current → _______
(define (list-cnames cs)
  (... (current-name cs) ...))

;; list-anames: Ancestor → _______
(define (list-anames as)
  (... (ancestor-name as) ...
       (list-names (ancestor-left as)) ...
       (list-names (ancestor-right as)) ...))
```

The contracts give important information that can guide the development.

What are they?

» list-names with an accumulator (1/2)

```
;; list-names: EvoTree → (listof Str)
(define (list-names t)
  (list-names/acc t '()))

;; list-names/acc: EvoTree (listof Str) → (listof Str)
(define (list-names/acc t names)
  (cond [(current? t) (list-cnames t names)]
        [(ancestor? t) (list-anames t names)]))
```


» list-names with an accumulator (2/2)

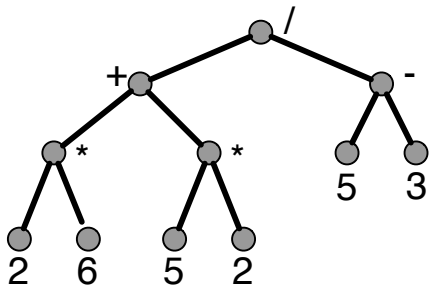
```
;; list-cnames: Current (listof Str) → (listof Str)
(define (list-cnames cs names)
  (cons (current-name cs) names))

;; list-ee-names: EvoEvent (listof Str) → (listof Str)
(define (list-anames as names)
  (cons (ancestor-name as)
        (list-names/acc (ancestor-left as)
                        (list-names/acc (ancestor-right as) names))))


(check-expect (list-names human) '("human"))
(check-expect (list-names e-mammals)
              '("early mammals" "early primates" "human" "chimp" "rat"))
```

» Practice problems with EvoTrees

- Count the number of ancestors (internal nodes) that split less than n (million) years ago. For example, the sample tree has 4 ancestor species that split less than 400 million years ago.
- Count the number of common ancestors for a given recent species.
- Find the evolutionary path between the root of a (sub)tree and a current species. For example, the path from `mco` to `rat` is `'("multi-celled organism" "early vertebrates" "early mammals" "rat")`.
- Modify `list-names` to produce the names of only endangered species.

Binary expression trees

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

> Representing binary arithmetic expressions

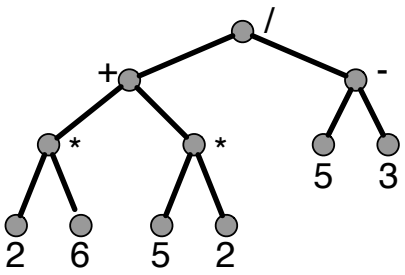
```
;; A binary arithmetic expression (BinExp) is one of:
;; * a Num
;; * a BINode

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

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 (1/2)

```
;; (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)]))
```


> Evaluating expressions (2/2)

```
;; (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.

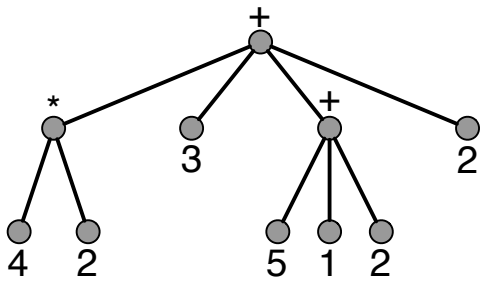
Racket expressions using the functions $+$ and $*$ can have an unbounded number of arguments. For example,

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

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

> Example general tree

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).


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

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

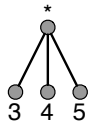
Each definition depends on the other, and each template will depend on the other.

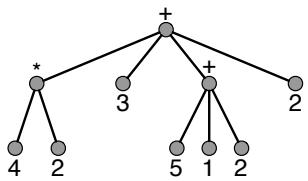
Examples: An `EvoTree` was defined in terms of `Current` and `Ancestor`. `Ancestor` was defined in terms of `EvoTree`.

`BINode` and `BinExp` also depend on each other.

Examples of arithmetic expressions:

```
3

(make-ainode '+ (list 3 4))

(make-ainode '* '(3 4 5))

(make-ainode '+ '((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) (... 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)) ...)]))
```


> eval and apply (1/2)

```
;; (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))]))
```


> eval and apply (2/2)

```
;; (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 (1/4)

```
 (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)))))
```


> Condensed trace of aexp evaluation (2/4)

```
⇒ (+ (* (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)))))
```


> Condensed trace of aexp evaluation (3/4)

```
⇒ (+ (* 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)))
```

> Condensed trace of aexp evaluation (4/4)

```
⇒ (+ 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 (+ 10 (apply '+ empty)))
⇒ (+ 12 (+ 10 0)) ⇒ 22
```


> Alternate data definition

In Module 8, 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`.

» Alternate data definition

```
;; 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))
```

Developing the alternative versions of `eval` and `apply` is left as an exercise.

> 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.

» Representing organized text

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

» Representing a web page

```
'(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") (3 "c"))
```

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

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

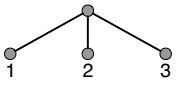

> Visualizing nested lists

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:

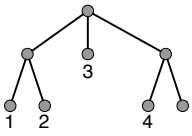
```
'()
```

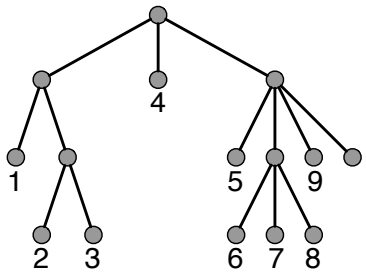
```
'(1)
```


```
'(1 2 3)
```




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






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

> Data definition for nested lists

Sample nested lists:

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

Observations:

- A nested list might be empty
- The first item of a non-empty nested list is either:
  - a nested list
  - a single item (a number, not a list)
- The rest of a non-empty nested list is a nested list

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

This can be generalized to generic types: `(nested-listof X)`.

> Template for nested lists

The template follows from the data definition.

```
;; nest-lst-template: (nested-listof X) → Any
(define (nest-lst-template lst)
  (cond [(empty? lst) ...]
        [(cons? (first lst)) (... (nest-lst-template (first lst)) ...
                                  (nest-lst-template (rest lst))  ...)]
        [else                (... (first lst) ...
                                  (nest-lst-template (rest lst)) ...)]))
```


> The function count-items

```
;; (count-items nl) counts the number of items in nl.
;; count-items: (nested-listof X) → Nat
(check-expect (count-items '()) 0)
(check-expect (count-items '((10 20) 30)) 3)

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

» 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 flat list with all the elements of lst.
;; flatten: (nested-listof X) → (listof X)

(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)
```


> Flattening a nested list

```
;; (flatten lst) produces a flat list with all the elements of lst.
;; flatten: (nested-listof X) → (listof X)

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


Exercise 1

Following the template, write a function that adds up all the values in a `(nested-listof Num)`.

```
(check-expect (nest-lst-sum '()) 0)
(check-expect (nest-lst-sum '(1 1)) 2)
(check-expect (nest-lst-sum '(1 2 (3 4) 7 ((1 4) 1))) 23)
```

Goals of this module (1/2)

- You should be familiar with tree terminology.
- You should understand the data definitions for binary trees, binary search trees, evolutionary trees, binary arithmetic expressions, general arithmetic expressions, and nested lists.
- 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.

Goals of this module (2/2)

- 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.