Trees

Readings: HtDP, sections 14, 15, 16.

We will cover the ideas in the text using different examples and different terminology. The readings are still important as an additional source of examples.

Binary arithmetic expressions

A binary arithmetic expression is made up of numbers joined by binary operations $*$, $+$, $/$, and $-$.

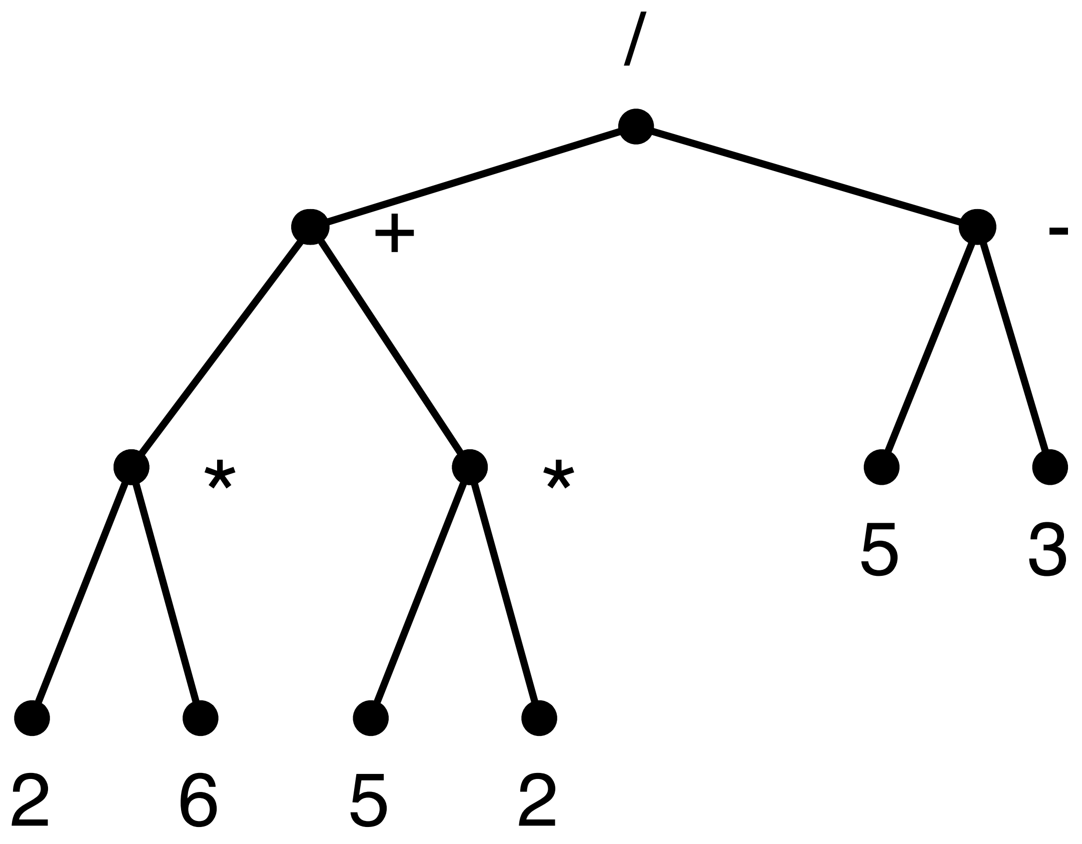
$((2*6)+(5*2))/(5-3)$ can be defined in terms of *two* smaller binary arithmetic expressions, $(2*6)+(5*2)$ and $5-3$.

Each smaller expression can be defined in terms of even smaller expressions.

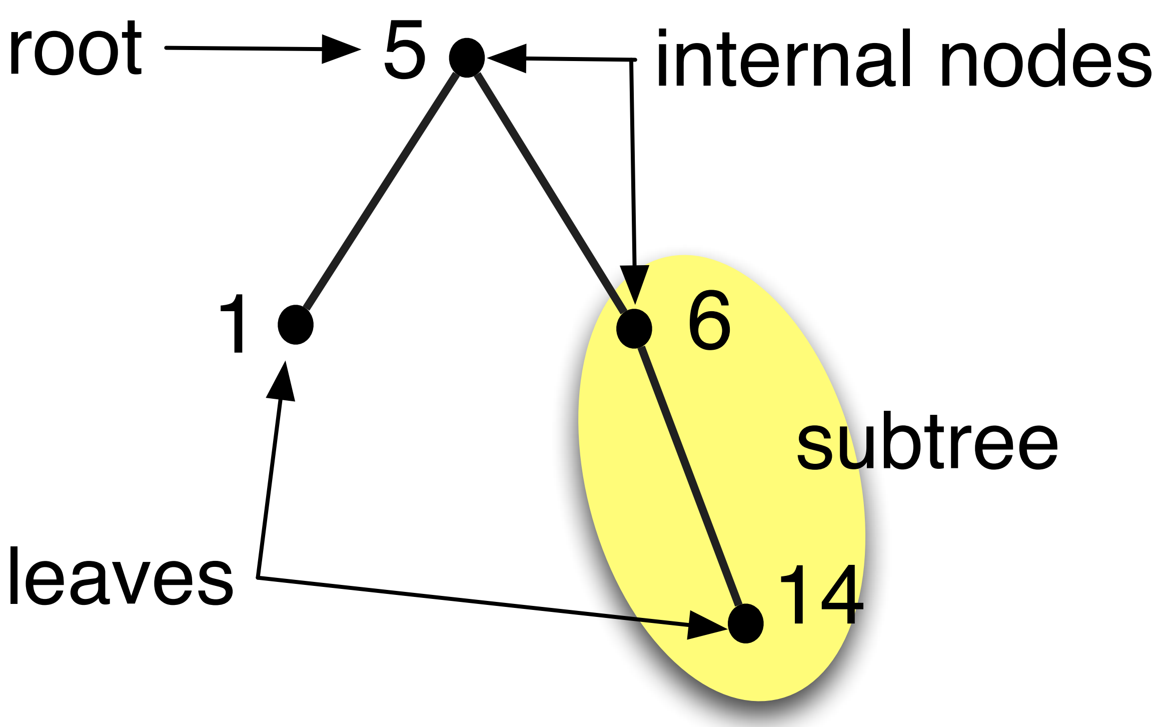
The smallest expressions are numbers.

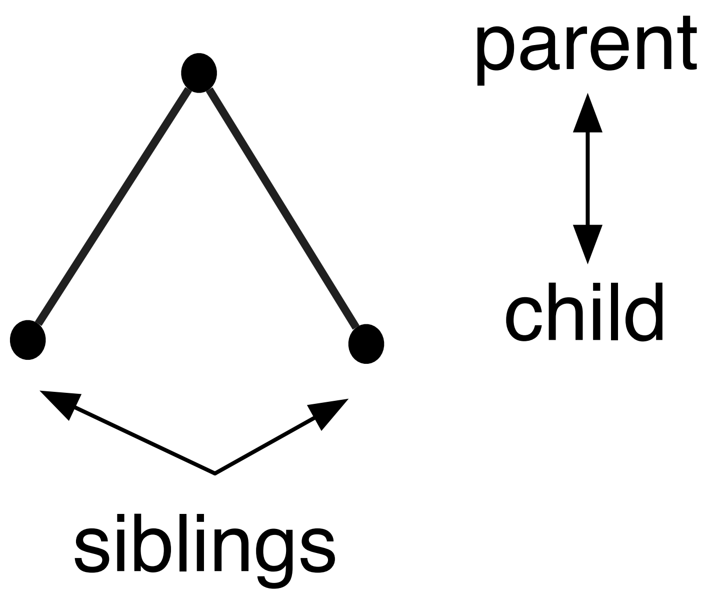
Visualizing binary arithmetic expressions

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



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)

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.

How can we group together information for an internal node?

How can we allow different definitions for leaves and internal nodes?

```
(define-struct binode (op arg1 arg2))
;; 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
;; Examples:
    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))
```

Template for binary arithmetic expressions

The only new idea in forming the template is the application of the recursive function to *each* piece that satisfies the data definition.

```
;; my-binexp-fn: BinExp → Any
(define (my-binexp-fn ex)
  (cond [(number? ex) ...]
        [else  (... (binode-op ex) ...
                    (my-binexp-fn (binode-arg1 ex)) ...
                    (my-binexp-fn (binode-arg2 ex)) ...)]))
```

Evaluation of expressions

```
(define (eval ex)
  (cond [(number? ex) ex]
        [else (cond [(symbol=? (binode-op ex) '*)
                      (* (eval (binode-arg1 ex)) (eval (binode-arg2 ex)))]
                     [(symbol=? (binode-op ex) '+)
                      (+ (eval (binode-arg1 ex)) (eval (binode-arg2 ex)))
                     [(symbol=? (binode-op ex) '/)
                      (/ (eval (binode-arg1 ex)) (eval (binode-arg2 ex)))]
                     [(symbol=? (binode-op ex) '-)
                      (- (eval (binode-arg1 ex)) (eval (binode-arg2 ex)))
```

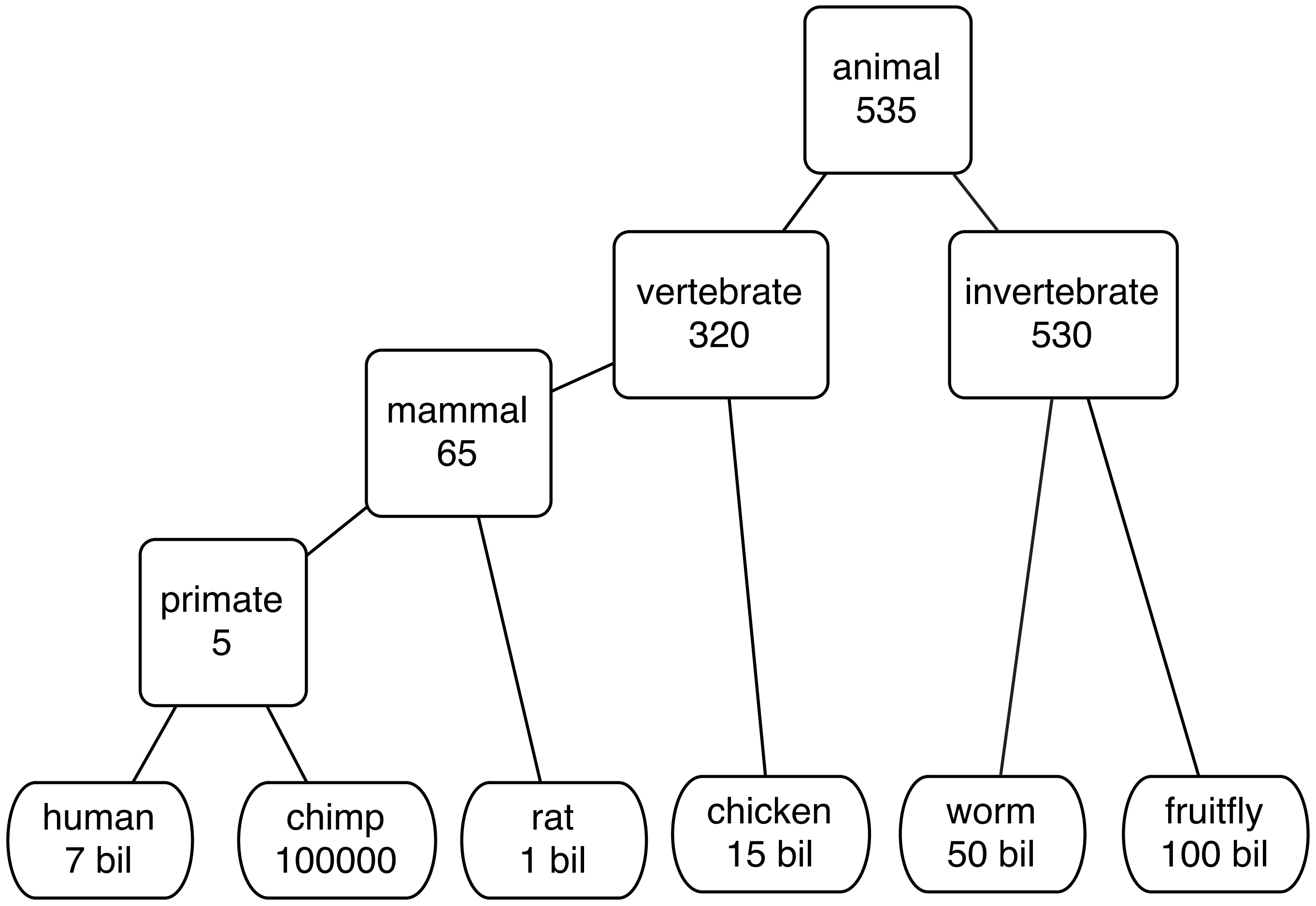
```
(define (eval ex)
  (cond [(number? ex) ex]
        [else (eval-binode (binode-op ex)
                           (eval (binode-arg1 ex))
                           (eval (binode-arg2 ex)))]))

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

Evolution trees

- a data structure recording information about the evolution of species
- contains two kinds of information:
  - modern species (e.g. humans) with population count
  - evolutionary history via **evolution events**, with an estimate of how long ago the event occurred

An “evolution event” leads to a splitting of one species into two distinct species (for example, through physical separation).


animal
535
vertebrate
320
invertebrate
530
mammal
65
primate
5
human
7 bil
chimp
100000
rat
1 bil
chicken
15 bil
worm
50 bil
fruitfly
100 bil

Representing evolution trees

Internal nodes each have exactly two children.

Leaves have names and populations of modern species.

Internal nodes have names and dates of evolution events.

The order of children does not matter.

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

Data definitions for evolution trees

```
;; A Taxon is one of:
;; ⋆ a Modern
;; ⋆ an Ancient

(define-struct modern (name pop))
;; A Modern is a (make-modern Str Nat)
(define-struct ancient (name age left right))
;; An Ancient is a (make-ancient Str Num Taxon Taxon)
```

Note that the Ancient data definition uses a pair of Taxons.

Here are the definitions for our example tree drawing.

```
(define-struct modern (name pop))
(define-struct ancient (name age left right))

(define human (make-modern "human" 6.8e9))
(define chimp (make-modern "chimpanzee" 1.0e5))
(define rat (make-modern "rat" 1.0e9))
(define chicken (make-modern "chicken" 1.5e10))
(define worm (make-modern "worm" 5.0e10))
(define fruit-fly (make-modern "fruit fly" 1.0e11))
```

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

Derive the template for taxon computations from the data definition.

```
;; my-taxon-fn: Taxon → Any
(define (my-taxon-fn t)
  (cond  [(modern? t)  (my-modern-fn t)]
         [(ancient? t) (my-ancient-fn t)]))
```

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

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

```
;; my-modern-fn: Modern → Any
(define (my-modern-fn t)
  (... (modern-name t) ...
       (modern-pop t) ...))

;; my-ancient-fn: Ancient → Any
(define (my-ancient-fn t)
        (... (ancient-name t) ...
             (ancient-age t) ...
             (ancient-left t) ...
             (ancient-right t) ... ))
```

We know that (ancient-left t) and (ancient-right t) are Taxons, so apply the Taxon-processing function to them.

```
;; my-ancient-fn: Ancient → Any
(define (my-ancient-fn t)
  (... (ancient-name t) ...
       (ancient-age t) ...
       (my-taxon-fn (ancient-left t)) ...
       (my-taxon-fn (ancient-right t)) ...))
```

my-ancient-fn uses my-taxon-fn and my-taxon-fn uses my-ancient-fn. This is called *mutual recursion*.

A function on taxons

This function counts the number of modern descendant species of a taxon. A taxon is its own descendant.

```
;; ndesc-species: Taxon → Nat
(define (ndesc-species t)
  (cond [(modern? t) (ndesc-modern t)]
        [(ancient? t) (ndesc-ancient t)]))

(check-expect (ndesc-species animal) 6)
(check-expect (ndesc-species human) 1)
```

```
;; ndesc-modern: Modern → Nat
(define (ndesc-modern t)
  1)

;; ndesc-ancient: Ancient → Nat
(define (ndesc-ancient t)
  (+ (ndesc-species (ancient-left t))
     (ndesc-species (ancient-right t))))
```

Counting evolution events

```
;; (recent-events t n) produces the number of evolution events in t
;;     taking place within the last n million years
;; recent-events: Taxon Num → Nat
;; Examples:
(check-expect (recent-events worm 500) 0)
(check-expect (recent-events animal 500) 3)
(check-expect (recent-events animal 530) 4)

(define (recent-events t n) . . . )
```

For a more complicated computation, try to compute the list of species that connects two specified taxons, if such a list exists.

The function ancestors consumes a taxon from and a modern species to. If from is the ancestor of to, ancestors produces the list of names of the species that connect from to to (including the names of from and to). Otherwise, it produces false.

What cases should the examples cover?

- The taxon from can be either a modern or a ancient.
- The function can produce either false or a list of strings.

```
;; (ancestors from to) produces chain of names from...to
;; ancestors: Taxon Modern → (anyof (listof Str) false)
;; Examples
(check-expect (ancestors human human) '("human"))
(check-expect (ancestors mammal worm) false)
(check-expect (ancestors mammal human)
              '("Mammal" "Primate" "human"))
(define (ancestors from to)
  (cond [(modern? from) (ancestors-modern from to)]
        [(ancient? from) (ancestors-ancient from to)]))
```

```
(define (ancestors-modern from to)
  (cond [(equal? from to)  (list (modern-name to))]
        [else false]))

(define (ancestors-ancient from to)
      (. . . (ancient-name from) . . .
             (ancient-age from) . . .
             (ancestors (ancient-left from) to) . . .
             (ancestors (ancient-right from) to) . . . ))
```

```
;; ancestors-ancient: Ancient Modern → (anyof (listof Str) false)
(check-expect (ancestors-ancient mammal worm) false)
(check-expect (ancestors-ancient mammal rat) '("Mammal" "rat"))
(check-expect (ancestors-ancient mammal human)
              '("Mammal" "Primate" "human"))
(define (ancestors-ancient from to)
  (cond
    [(cons? (ancestors (ancient-left from) to))
     (cons (ancient-name from) (ancestors (ancient-left from) to))]
    [(cons? (ancestors (ancient-right from) to))
     (cons (ancient-name from) (ancestors (ancient-right from) to))]
    [else false]))
```

When we try filling in the recursive case, we notice that we have to use the results of the recursive call more than once.

To avoid repeating the computation, we can pass the results of the recursive calls (plus whatever other information is needed) to a helper function.

A more efficient ancestors-ancient function

```
(define (ancestors-ancient from to)
  (extend-list (ancient-name from)
               (ancestors (ancient-left from) to)
               (ancestors (ancient-right from) to)))

(define (extend-list from-n from-l from-r)
  (cond [(cons? from-l) (cons from-n from-l)]
        [(cons? from-r) (cons from-n from-r)]
        [else false]))
```

Binary trees

Evolution trees are an example of **binary trees**: trees with at most two children for each node.

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

Next we will use another type of binary tree to provide a more efficient implementation of dictionaries.

Dictionaries revisited

Recall from module 06 that a dictionary stores a set of (key, value) pairs, with at most one occurrence of any key.

It supports lookup, add, and remove operations.

We implemented a dictionary as an association list of two-element lists.

This implementation had the problem that a search could require looking through the entire list, which will be inefficient for large dictionaries.

Our new implementation will put the (key,value) pairs into a **binary tree** instead of a **list** in order to quickly search large dictionaries. (We'll see how soon.)

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

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

What is the template?

Counting values in a Binary Tree

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

```
;; count-values: BT Str → Nat
(define (count-values tree v)
  (cond [(empty? tree) 0]
        [else (+ (cond [(string=? v (node-val tree)) 1]
                       [else 0])
                 (count-values (node-left tree) v)
                 (count-values (node-right tree) v))]))
```

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))
                         (node-val tree)
                         (increment (node-left tree))
                         (increment (node-right tree)))]))
```

Searching binary trees

We are now ready to try to search our binary tree to produce the value associated with the given key (or false if the key is not in the dictionary).

Our strategy:

- See if the root node contains the key we're looking for. If so, produce the associated value.
- Otherwise, recursively search in the left subtree, and in the right subtree. If either recursive search comes up with a string, produce that string. Otherwise, produce false.

Much as with ancestors-ancient, where we were searching an evolution tree, we will find a helper function to be useful:

```
;; (pick-string a b) produces a if it is a string,
;;     or b if it is a string, otherwise false
;; pick-string: (anyof Str false) (anyof Str false) → (anyof Str false)
(check-expect (pick-string false "hi") "hi")
(check-expect (pick-string "there" false) "there")
(check-expect (pick-string false false) false)
(define (pick-string a b)
  (cond [(string? a) a]
        [(string? b) b]
        [else false]))
```

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

```
;; search-bt: Num BT → (anyof Str false)
(define (search-bt n tree)
  (cond [(empty? tree) false]
        [(= n (node-key tree)) (node-val tree)]
        [else (pick-string
                (search-bt n (node-left tree))
                (search-bt n (node-right tree)))]))
```

Is this more efficient than searching an AL?

Binary search trees

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

The (key,value) pairs are stored at the nodes of a tree.

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

The placement of pairs in nodes can make it possible to improve the running time compared to association lists.

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

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

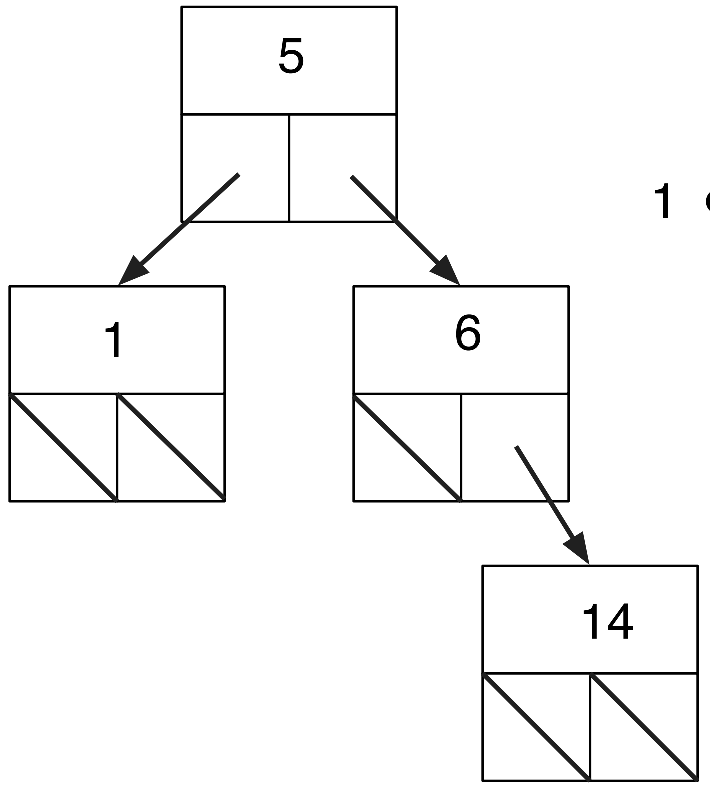
The BST **ordering property**:

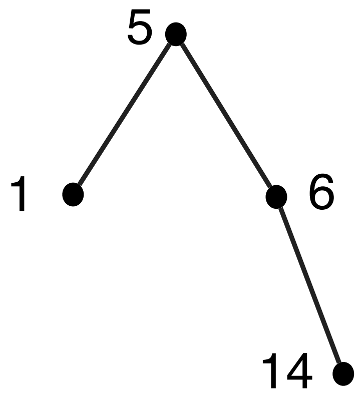
- key is greater than every key in left.
- key is less than every key in right.

A BST example

```
(make-node 5 "John"
  (make-node 1 "Alonzo" empty empty)
  (make-node 6 "Alan"
             empty
             (make-node 14 "Ada"
                        empty
                        empty)))
```

Drawing BSTs





(Note: the value field is not represented.)

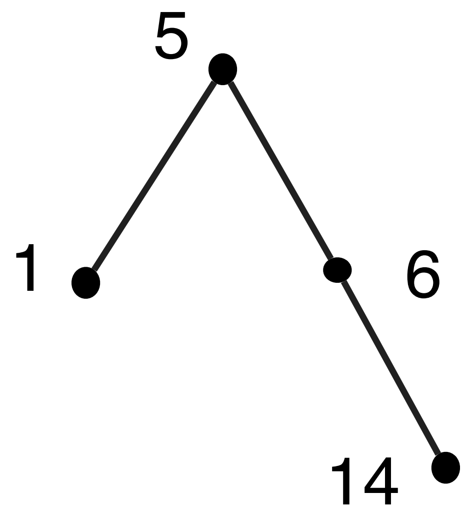
We have made several minor changes from HtDP:

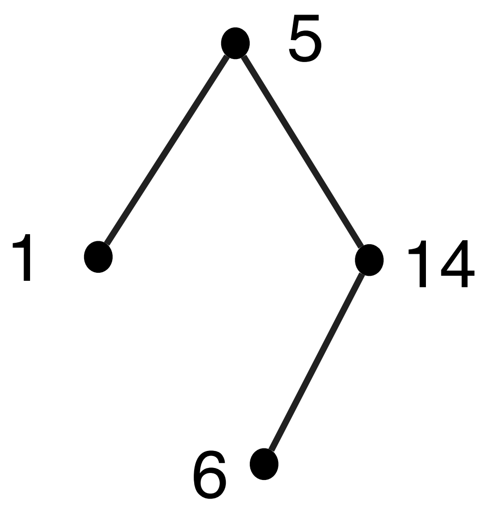
- We use empty instead of false for the base case.
- We use key instead of ssn, and val instead of name.

The value can be any Racket value.

We can generalize to other key types with an ordering (e.g. strings, using string<?).

There can be several BSTs holding a particular set of (key, value) pairs.





Making use of the ordering property

Main advantage: for certain computations, one of the recursive calls 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 v 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 cannot be in r, and we only need to recursively search in l.

If n $>$ k, then n cannot be in l, and we only need to recursively search in r.

Either way, we save one recursive call.

```
;; (search-bst n t) produces the value associated with n,
;;     or false if n is not in t
;; search-bst: Num BST → (anyof Str false)
(define (search-bst n t)
  (cond [(empty? t) false]
        [(= n (node-key t)) (node-val t)]
        [(< n (node-key t)) (search-bst n (node-left t))]
        [(> n (node-key t)) (search-bst n (node-right t))]))
```

Creating a BST

How do we create a BST from a list of (key, value) pairs?

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 (list k v) lst), we add the pair (k, v) to the BST created from the list lst.

Adding to a BST

How do we add a pair (k, v) to a BST bstree?

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

Otherwise bstree is of the form (make-node n w l r).

If k = n, we form the tree with k, v, l, and r (we replace the old value by v).

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

Binary search trees in practice

If the BST has all left subtrees empty, it looks and behaves like a sorted association 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.

By that time you will better understand how to analyze the efficiency of algorithms and operations on data structures.

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

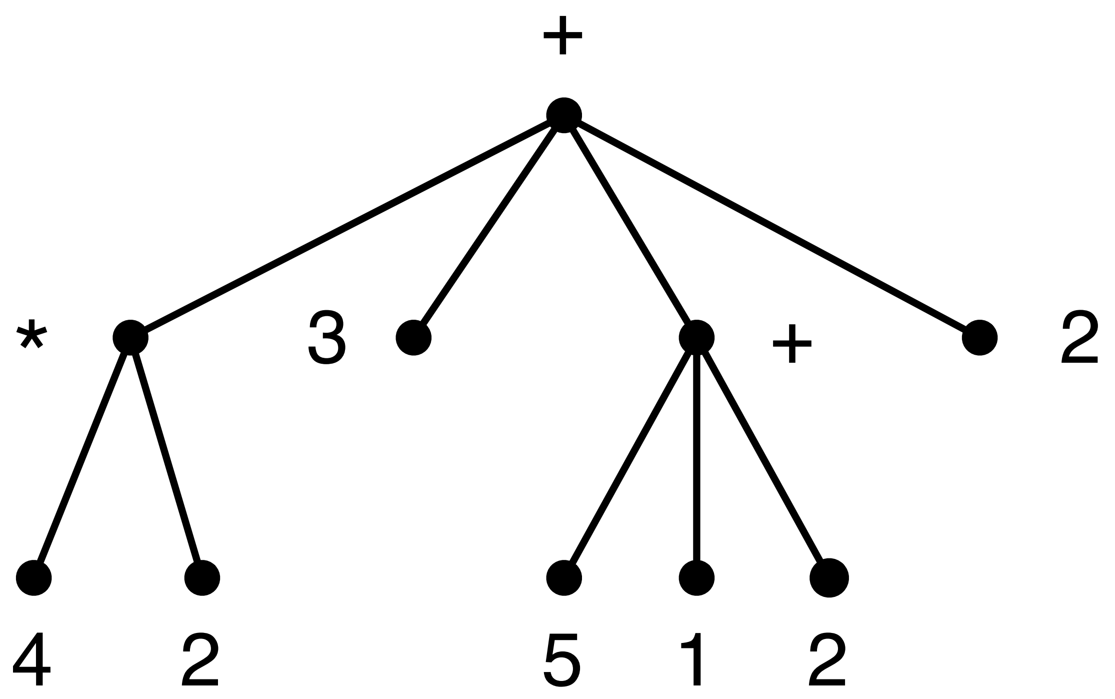
For binary arithmetic expressions, we formed binary trees.

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).

We also need the data definition of 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. We saw this in the evolution trees example, as well. A Taxon was defined in terms of a modern species and an ancient species. The ancient species was defined in terms of a Taxon.

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))
```

It is also possible to have an operation and an empty list; recall substitution rules for and and or.

Templates for arithmetic expressions

```
(define (my-aexp-fn ex)
   (cond [(number? ex) . . . ]
         [else (. . . (ainode-op ex) . . .
                      (my-listof-aexp-fn (ainode-args ex)) . . . )]))

(define (my-listof-aexp-fn exlist)
   (cond [(empty? exlist) . . . ]
         [else  (. . . (my-aexp-fn (first exlist)) . . .
                       (my-listof-aexp-fn (rest exlist)) . . . )]))
```

The function eval

```
;; eval: AExp → Num

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

```
;; apply: Sym (listof AExp) → Num
(define (apply f exlist)
  (cond [(empty? exlist)
           (cond [(symbol=? f '*) 1]
                 [(symbol=? f '+) 0])]
        [else
           (cond [(symbol=? f '*)
                  (* (eval (first exlist)) (apply f (rest exlist)))]
                 [(symbol=? f '+)
                  (+ (eval (first exlist)) (apply f (rest exlist)))])]))
```

A simplified apply

```
;; apply: Sym (listof AExp) → Num
(define (apply f exlist)
  (cond [(and (empty? exlist) (symbol=? f '*)) 1]
        [(and (empty? exlist) (symbol=? f '+)) 0]
        [(symbol=? f '*)
         (* (eval (first exlist)) (apply f (rest exlist)))]
        [(symbol=? f '+)
         (+ (eval (first exlist)) (apply f (rest exlist)))]))
```

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 student 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 (my-altaexp-fn ex)
  (cond [(number? ex) . . . ]
        [else (. . . (first ex) . . .
                (my-listof-altaexp-fn (rest ex)) . . . )]))

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

```
;; eval: AltAExp → Num

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

```
;; apply: Sym AltAExpList → 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)))
```

$\Rightarrow$ (* (+ 1 2) (apply '* '(4)))

$\Rightarrow$ (* 3 (apply '* '(4)))

$\Rightarrow$ (* 3 (* 4 (apply '* '())))

$\Rightarrow$ (* 3 (* 4 1))

$\Rightarrow$ (* 3 4)

$\Rightarrow$ 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 115: Introduction to Computer Science 1")
   (paragraph "For a course description,"
              (link "click here." "desc.html")
               "Enjoy the course!")
   (horizontal-line)
   (paragraph "(Last modified yesterday.)"))
```

Nested lists

So far, we have discussed flat lists (no nesting):

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

and lists of lists (one level of nesting):

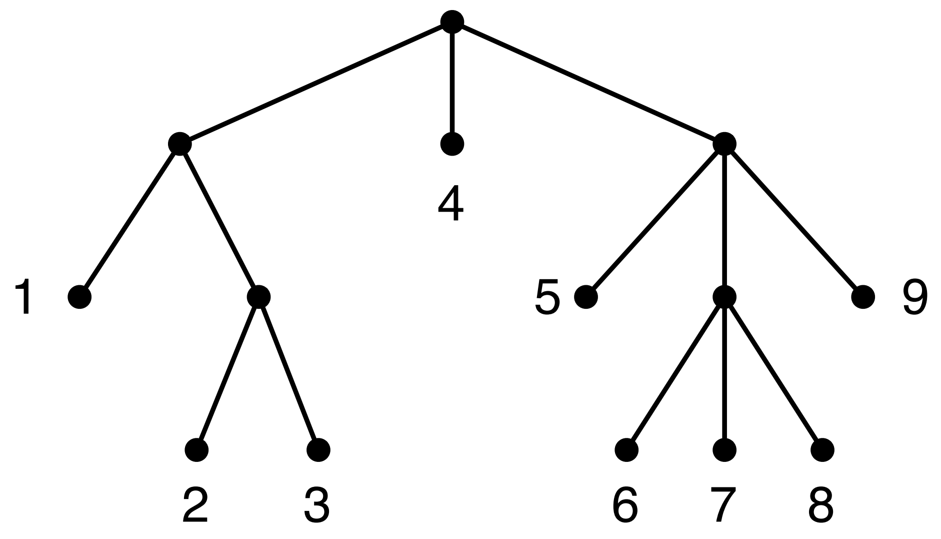
```
(list (list 1 "a") (list 2 "b"))
```

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

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

Nested lists as leaf-labelled trees

It is often useful to visualize a nested list as a **leaf-labelled 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)
```

Generic data definition for nested lists

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

Template for nested lists

The template follows from the data definition.

```
(define (my-nest-lst-fn lst)
  (cond [(empty? lst) . . . ]
        [(X? (first lst))
         (. . . (first lst) . . . (my-nest-lst-fn (rest lst)) . . . )]
        [else
         (. . . (my-nest-lst-fn (first lst)) . . .
                (my-nest-lst-fn (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: Nest-List-Num → (listof Num)
(define (flatten lst) . . . )
```

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

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

Remember: use append only when the first list has length greater than one, or there are more than two lists.

```
;; 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 arithmetic expressions, evolution trees, and binary search trees, 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.