10: Binary Trees

Binary Trees

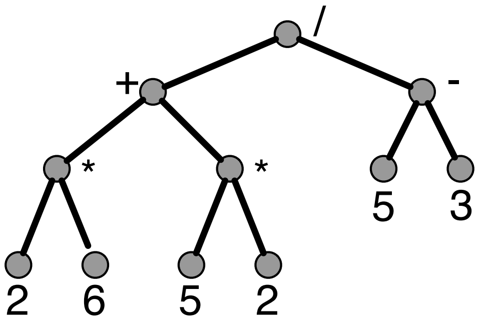
We now want to be able to store information in more complicated ways.

It turns out that the structure of a **tree** has many, many applications in computer science. Sometimes trees work well because the problem itself has a tree-like structure. Sometimes trees work well because we can impose a tree-like structure on the problem in a way that allows our code to run faster, often dramatically so.

You will see trees used in many contexts in future courses.

> Example: binary expression trees

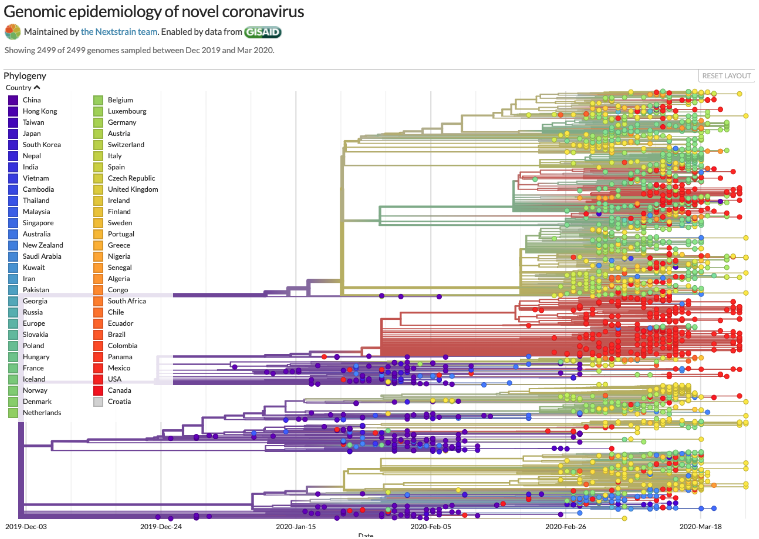
The expression $((2 * 6) + (5 * 2))/(5 - 3)$ can be represented as a **tree**:



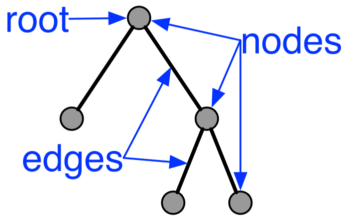
> Example: Phylogenetic trees

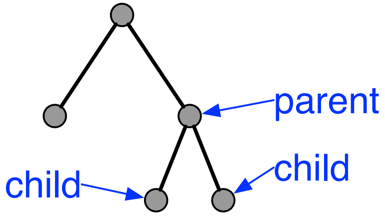
This phylogenetic tree tracks the evolution of COVID-19 in the first four months of the recent pandemic.

Image: `nextstrain.org/ncov`



> Tree terminology (1/2)

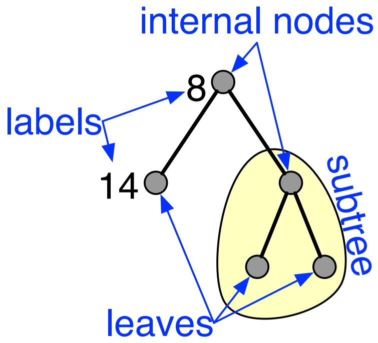




A **tree** is a set of **nodes** and **edges** where an edge connects two distinct nodes.
A tree has three requirements:

- 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)

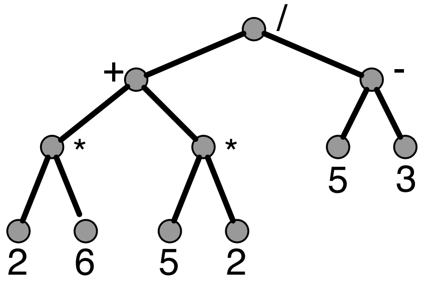


Other useful terms:

- **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 (which includes *n*)
- **subtree** rooted at *n*: all of the descendents of *n*

> 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

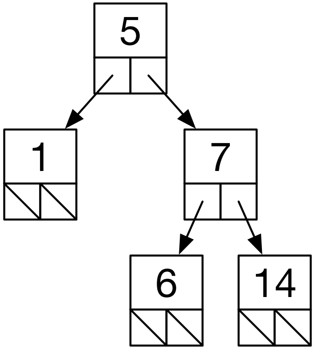
A **binary tree** is a tree with at most two children for each node.

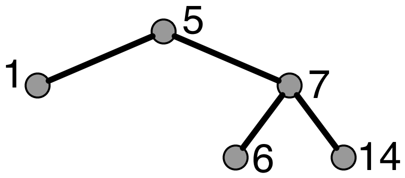
Binary trees are a fundamental part of computer science, independent of what language you use.

Binary arithmetic expression trees and phylogenetic 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





- We will consistently use `Nat`s in our binary trees, but we could use symbols, strings, structures, ...
- CS135 tends to use drawings like on the right; it's more succinct and easier to draw. However, it omits empty subtrees.

> 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.

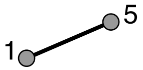
How are trees encoded using the `node` structure? What is the template?

Sample Trees

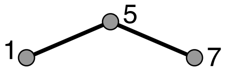
An empty tree

```
empty
```

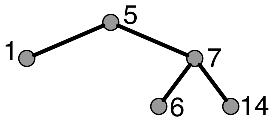
```
(make-node 5 empty empty)
```



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



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



```
(make-node 5 (make-node 1 empty empty)
             (make-node 7 (make-node 6 empty empty)
                          (make-node 14 empty empty)))
```

Aside: Tips for building templates

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

- says “one of”, include a `cond` to distinguish the cases.
- is a simple type (`Int`, `Str`, `Sym`, `empty`, etc), do nothing.
- is a defined data type, apply that type’s template.
- is compound data (a structure, fixed-length list), extract each of the fields, in order.
- is a `listof`, extract the `first` and `rest` of the list.

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

Apply the above recursively.

> Binary tree template

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

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

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

> Example: sum-keys

```
;; (sum-keys bt) sums the keys in bt.
;; Examples
(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)

;; sum-keys: BT → Nat
(define (sum-keys bt)
```

Key Questions

We had four key questions for recursion on a list and four very similar questions for recursion on a natural number. What are the equivalent questions for a tree?

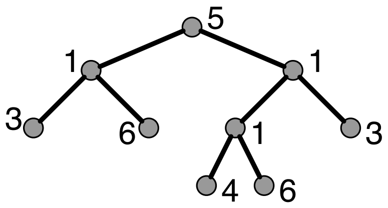
1. What should the function produce in the base case?
2. What should calling the function on the rest of the list produce?
3. What should the function do to the first element in a non-empty list?
4. How should the function combine #2 and #3 to produce the answer for the entire list?

1. What should the function produce in the base case?
2. What should calling the function on the left subtree produce?
3. What should calling the function on the right subtree produce?
4. What should the function do to the root node in a non-empty tree?
5. How should the function combine #2, #3, and #4 to produce the answer for the entire tree?

> Example: count-nodes

```
(define test-tree (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.
(check-expect (count-nodes empty 5) 0)
(check-expect (count-nodes test-tree 1) 3)

;; count-nodes: BT Nat → Nat
(define (count-nodes tree k)
  (cond [(empty? tree) 0]
        [(node? tree) (+ (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]
    [(node? tree) (make-node (add1 (node-key tree))
                             (increment (node-left tree))
                             (increment (node-right tree)))]))
```

Ex. 1 Write `(count-leaves t)` which consumes a binary tree and produces the number of leaf nodes.

Ex. 2 Write `(count-evens t)` which consumes a binary tree and produces the number of nodes with an even key.

Ex. 3 Write `(reverse-tree t)` which consumes a binary tree and produces a new tree that has the same keys as `t` but every node has the left and right subtrees reversed (the left subtree becomes the right; the right subtree becomes the left).

> 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.

1. What should the function produce in the base case?
2. What should calling the function on the left subtree produce?
3. What should calling the function on the right subtree produce?
4. What should the function do to the root node in a non-empty tree?
5. How should the function combine #2, #3, and #4 to produce the answer for the entire tree?

> Searching binary trees

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

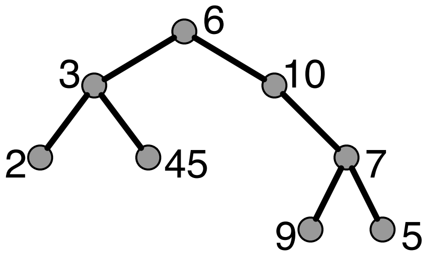
```
;; (search k tree) produces true if k is in tree; false otherwise.
;; search: Nat BT → Bool
(define (search-bt k tree)
  (cond [(empty? tree) false]
        [else (or (= k (node-key tree))
                  (search-bt k (node-left tree))
                  (search-bt k (node-right tree)))]))
```

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.

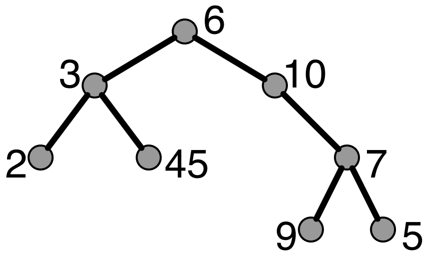
Assume, for now, that the tree does not contain duplicates.



The path from the root to 9 is `(list '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 0 empty) false)
(check-expect (search-bt-path 6 test-tree) empty)
(check-expect (search-bt-path 3 test-tree) (list 'left))
(check-expect (search-bt-path 9 test-tree) (list 'right 'right 'left))
(check-expect (search-bt-path 0 test-tree) false)
```

> search-bt-path

```
;; search-bt-path: Nat BT → (anyof false (listof (anyof 'right 'left)))
(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!

> Improved search-bt-path

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

(define (choose-path-v2 left-path right-path)
  (cond [(list? left-path) (cons 'left left-path)]
        [(list? right-path) (cons 'right right-path)]
        [else false]))
```

Ex. 4

Write `insert-path` which consumes a binary tree, a path such as `(list 'left 'left 'right)` and a value. It produces the same binary tree but with the new value in the node reached by the path in the second argument. Assume that the path reaches a non-empty node in the tree.

Ex. 5

Write `add-path` which consumes a binary tree, a path such as `(list 'left 'left 'right)` and a value. It produces the same binary tree but with the new value inserted into the tree at the end of the path. If the path ends at an interior node, that node is replaced by a new node containing the new value, the existing node as the left subtree, and an empty right subtree.

Binary search trees

We will now make one change that can make searching **much** more efficient. This change will create 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.

BST Data Definitions

```
;; 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
```

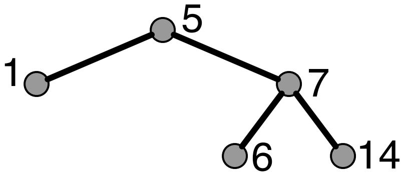
The BST **ordering property**:

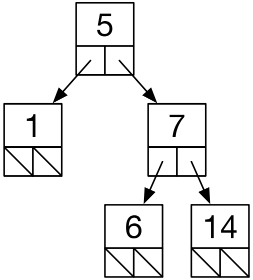
- `key` is greater than every key in `left`.
- `key` is less than every key in `right`.

Note: the ordering property holds in every subtree

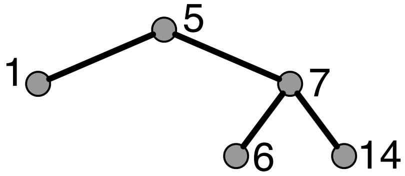
> A BST example

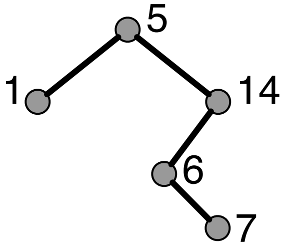
```
(make-node 5
  (make-node 1 empty empty)
  (make-node 7
             (make-node 6 empty empty)
             (make-node 14 empty empty)))
```

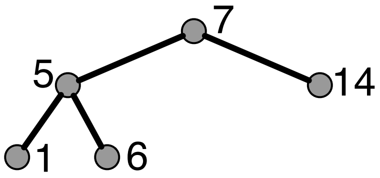




There can be several BSTs holding a particular set of keys.







> 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).

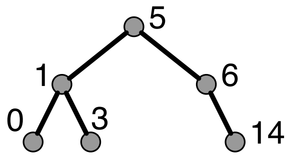
In the following slides, we will demonstrate this advantage for searching and adding.

> Searching in a BST

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

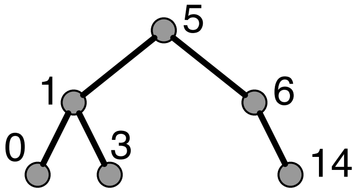
- If the BST is `empty`, then `n` is not in the BST.
- If the BST is of the form `(make-node k left right)`, and `k` equals `n`, then we have found it.
- Otherwise it might be in either the `left` or `right` subtree.
  - If `n` $<$ `k`, then `n` must be in `left` if it is present at all, and we only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

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



Ex. 6

Write a function (count-smaller n t) that consumes a Nat and a BST, and returns the number of keys in t that are less than n.

```
(define example
  (make-node 5
             (make-node 1 empty empty)
             (make-node 7
                        (make-node 6 empty empty)
                        (make-node 14 empty empty))))
(check-expect (count-smaller 8 example) 4)
(check-expect (count-smaller 1 example) 0)
(check-expect (count-smaller 100 example) 5)
```

For efficiency: don’t always search both children!

> 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

Consider a function, `build-bst-from-list`, which builds a BST from a list of keys. How would it work?

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**.

We could use accumulative recursion to insert the 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.

Ex. 7

Practise problems on binary search trees:

- Write a function, `bst-min`, which consumes a non-empty BST and produces the minimum value in the tree.
- Write a function, `bst-max`, which consumes a non-empty BST and produces the maximum value in the tree.
- Write a function, `bst-add`, which consumes a BST, `t`, and a new key, `n`. It produces a BST with all of the keys found in `t` as well as `n`. If `n` happens to already exist in `t` it produces a BST just like `t`.
- Write a function, `bst-from-list`, which consumes a `(listof Nat)` and produces a BST containing those same values. Use simple recursion.
- Write a function, `bst-from-list/acc`, which consumes a `(listof Nat)` and produces a BST containing those same values. Use accumulative recursion.
- Rewrite `search-bst` without a **`cond`** (using a single Boolean expression).

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` (a.k.a `search`), `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

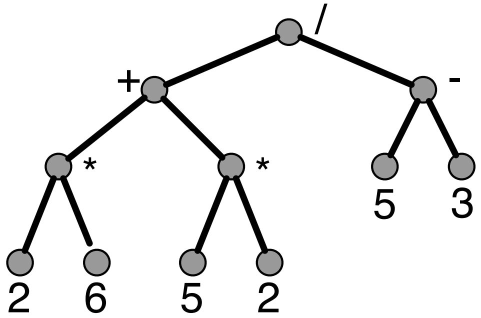
```
(define test-tree (make-node 5 "Susan"
                             (make-node 1 "Juan" empty empty)
                             (make-node 14 "David"
                                        (make-node 6 "Lucy" empty empty)
                                        empty)))

(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)
```

Binary arithmetic expression trees

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

- 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.



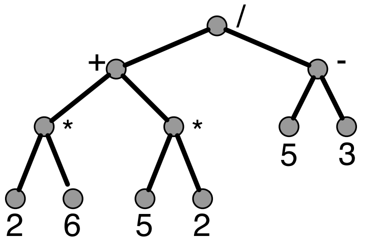
> Representing binary arithmetic expressions

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



> Templates for binary arithmetic expressions

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


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

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

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

> Evaluating expressions (1/2)

```
;; (eval ex) evaluates the expression ex and produces its value.
(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)

;; eval: BinExp → Num
(define (eval ex)
  (cond [(number? ex) ex]
        [(binode? ex) (eval-binode ex)]))
```

> Evaluating expressions (2/2)

```
;; (eval-binode node) evaluates the expression represented by 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-val right-val)
  (cond [(symbol=? op '*) (* left-val right-val)]
        [(symbol=? op '/) (/ left-val right-val)]
        [(symbol=? op '+) (+ left-val right-val)]
        [(symbol=? op '-) (- left-val right-val)]))
```

Goals of this module

- You should be familiar with tree terminology.
- You should understand the data definitions for binary trees, binary search 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 binary expression trees and be able to extend them to handle more operations.

Summary: built-in functions

The following functions and special forms have been introduced in this module:

You should complete all exercises and assignments using only these and the functions and special forms introduced in earlier modules. The complete list is:

`*` `+` `-` `...` `/` `<` `<=` `=` `>` `>=` `abs` `add1` **`and`** `append` `boolean?` `ceiling` `char-alphabetic?` `char-downcase` `char-lower-case?` `char-numeric?` `char-upcase` `char-upper-case?` `char-whitespace?` `char<=?` `char<?` `char=?` `char>=?` `char>?` `char?` `check-error` `check-expect` `check-within` **`cond`** `cons` `cons?` `cos` **`define`** **`define-struct`** `define/trace` `e` `eighth` **`else`** `empty?` `equal?` `error` `even?` `exp` `expt` `fifth` `first` `floor` `fourth` `integer?` `length` `list` `list->string` `list?` `log` `max` `min` `modulo` `negative?` `not` `number->string` `number?` `odd?` **`or`** `pi` `positive?` `quotient` `remainder` `rest` `reverse` `round` `second` `seventh` `sgn` `sin` `sixth` `sqr` `sqrt` `string->list` `string-append` `string-downcase` `string-length` `string-lower-case?` `string-numeric?` `string-upcase` `string-upper-case?` `string<=?` `string<?` `string=?` `string>=?` `string>?` `string?` `sub1` `substring` `symbol=?` `symbol?` `tan` `third` `zero?`