CS115 - Module 10 - General Trees

Cameron Morland

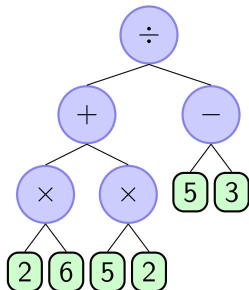
Fall 2017

Reminder: if you have not already, ensure you:

- Read *How to Design Programs*, Sections 15 and 16.

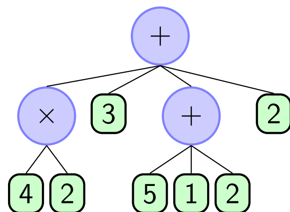
Recall with binary trees we could represent an expression containing binary operators (which have exactly 2 arguments), such as

$$((2 \times 6) + (5 \times 2)) \div (5 - 3)$$



But in Racket we are't required to have just 2 arguments. How could we represent `(+ (* 4 2) 3 (+ 5 1 2) 2)` ?

We can make a new structure where each node can have any number of children, instead of just 2.



Here we still label each node; the only difference is the number of children.

Representing binary vs general arithmetic expressions

Previously, our data definitions were as follows:

```
;; an Operator is (anyof '+ '- '* '/)

(define-struct binode (op arg1 arg2))
;; a binary arithmetic expression internal node (BINode)
;; is a (make-binode Operator BinExp BinExp)

;; A binary arithmetic expression (BinExp) is either:
;;   a Num or
;;   a BINode
```

Now we want all same, except: instead of `arg1` and `arg2`, we will use a list:

```
(define-struct ainode (op args))
;; an arithmetic expression internal node (AINode)
;; is a (make-ainode Operator (listof AExp))

;; An arithmetic expression (AExp) is either:
;;   a Num or
;;   a AINode
```

So given the data definition:

```
(define-struct ainode (op args))
;; an arithmetic expression internal node (AINode)
;; is a (make-ainode Operator (listof AExp))

;; An arithmetic expression (AExp) is either:
;;   a Num or
;;   a AINode
```

How can we represent an expression such as `(+ (* 4 2) 3 (+ 5 1 2) 2)` ?

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

When we evaluated binary trees we used the following function:

```
;; (eval-binexp expr) return the value of expr.
;; eval-binexp: BinExp -> Num
(define (eval-binexp expr)
  (cond [(number? expr) expr] ; numbers evaluate to themselves
        [(symbol=? (binode-op expr) '*)
         (* (eval-binexp (binode-arg1 expr))
            (eval-binexp (binode-arg2 expr)))]
        [(symbol=? (binode-op expr) '+)
         (+ (eval-binexp (binode-arg1 expr))
            (eval-binexp (binode-arg2 expr)))]))
```

All that is different now is that we have a list of values, `args`, instead of just `arg1` and `arg2`.

Our code for binary expressions is almost perfect. Instead of evaluating exactly two items, `(binode-arg1 expr)` and `(binode-arg2 expr)`, then combining them with `*` or `+`, we have this list. We need to replace the code:

```
(+ (eval-binexp (binode-arg1 expr))
   (eval-binexp (binode-arg2 expr)))]))
```

We need to use evaluate each argument in the list. We could do this recursively. Or use **`map`**!

```
(map eval-ainexp (ainode-args expr))
```

This takes a list of expressions, evaluates each one, and returns the resulting list.
But now we need to combine them. Again, we could do this recursively. But we don't need to!
Use **`foldr`** instead:

```
(foldr + 0
       (map eval-ainexp (ainode-args expr))
       )
```

The whole code then becomes:

```
;; (eval-ainexp expr) return the value of expr.
;; eval-ainexp: AExp -> Num
;; Examples:
(check-expect (eval-ainexp (make-ainode '* (list 2 3 5))) 30)

(define (eval-ainexp expr)
  (cond [(number? expr) expr] ; numbers evaluate to themselves
        [(symbol=? (ainode-op expr) '*)
         (foldr * 1
                (map eval-ainexp (ainode-args expr))
                )
         ]
        [(symbol=? (ainode-op expr) '+)
         (foldr + 0
                (map eval-ainexp (ainode-args expr))
                )
         ]
        ))
```

Working with general trees

Recalling the following data definition:

```
(define-struct ainode (op args))
;; an arithmetic expression internal node (AINode)
;; is a (make-ainode Operator (listof AExp))

;; An arithmetic expression (AExp) is either:
;;    a Num or
;;    a AINode
```

Exercise: Complete `count-leaves`.

```
;; (count-leaves expr) return the number of leaves in expr.
;; count-leaves: AExp -> Nat
;; Examples:
(check-expect (count-leaves
               (make-ainode
                '+ (list 2 3 (make-ainode '* (list 6 7 42))))) 5)
```

Mutual Recursion

When one function calls a second, and the second calls the first, it is called **mutual recursion**. Our approach to writing such code is unchanged: write a template for each datatype, based on the data definition; then build functions following the template.

Consider:

```
;; an EvenNat is either
;;   0 or
;;   (add1 x) where x is an OddNat.

;; an OddNat is a Nat that is not even.
```

```
;; (is-even? n) return #true if n is an
;;   EvenNat, otherwise #false.
;; is-even?: Nat -> Bool

(define (is-even? n)
  (cond [(= n 0) #true]
        [else (is-odd? (sub1 n))]))

(define (is-odd? n)
  (not (is-even? n)))
```

Mutual Recursion

```
(define-struct ainode (op args))
;; an arithmetic expression internal node (AINode)
;; is a (make-ainode Operator (listof AExp))

;; An arithmetic expression (AExp) is either:
;;    a Num or
;;    a AINode
```

We have defined:

1. `AINode`, which is defined in terms of `(listof AExp)`
2. `(listof AExp)`, which is defined in terms of `AINode`.

If we wrote a recursive function `evaluate-and-add` to work through the `(listof AExp)`, the function would itself call `eval-ainexp`. But `eval-ainexp` would call `evaluate-and-add`.
This is a mutually-recursive definition.
An example is worked in the official notes.
The greater simplicity of the solution using **`filter`** and **`map`** shows their usefulness!

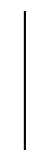
Leaf Labelled Trees

Earlier we worked with trees which had labels on all the nodes. Each node consisted of a `(make-node Label (listof Node))`; the `Label` indicating something about the node (in our example, the operator, `'+` or `'*`).

We can simplify trees even further by not labelling internal nodes.

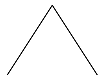
```
;; a leaf-labelled tree (LLT) is a (listof (anyof Num LLT)).
```

```
(list 2)
```



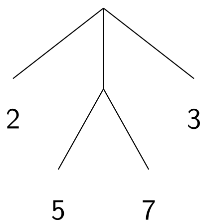
2

```
(list 5 7)
```

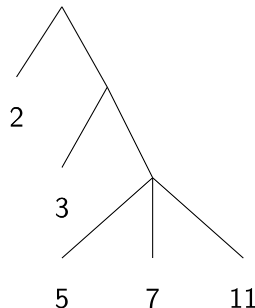


5 7

```
(list 2 (list 5 7) 3)
```




```
(list 2 (list 3 (list 5 7 11)))
```



Template for Leaf Labelled Trees

```
;; a leaf-labelled tree (LLT) is a (listof (anyof Num LLT)).
```

Since a `LLT` is a list, its template comes from the list template, reproduced here:

```
;; my-listof-X-fun: (listof X) -> Any
(define (my-listof-X-fun L)
  (cond [(empty? L) ... ]
        [else (... (first L) ... (my-listof-X-fun (rest L)) ...)]))
```

Since each item in the list may be a `Num` or a `LLT`, we need to distinguish these.
We could recognize the `LLT` using `cons?`; for an example, see the official notes.
Instead, I will identify the leaves using `number?`.

```
;; my-LLT-fun: LLT -> Any
(define (my-LLT-fun L)
  (cond [(empty? L) ... ]
        [(number? (first L)) ;; first is a leaf.
         (... (first L) ... (my-LLT-fun (rest L)) ...)]
        [else                ;; first is not a leaf, so it must be an LLT.
         (... (my-LLT-fun (first L))
              ... (my-LLT-fun (rest L)) ...)]))
```

```
;; a leaf-labelled tree (LLT) is a (listof (anyof Num LLT)).

;; my-LLT-fun: LLT -> Any
(define (my-LLT-fun L)
  (cond [(empty? L) ... ]
        [(number? (first L)) ;; first is a leaf.
         (... (first L) ... (my-LLT-fun (rest L)) ...)]
        [else                ;; first is not a leaf, so it must be an LLT.
         (... (my-LLT-fun (first L))
              ... (my-LLT-fun (rest L)) ...)]))
```

Exercise: Following the template, write a function to count the leaves of a `LLT`.

Exercise: Following the template, complete `depth`.

```
;; (depth tree) return the max distance from the root to a leaf of tree.
;; depth: LLT -> Nat
;; Examples:
(check-expect (depth (list 6 7)) 1)
(check-expect (depth (list 2 (list 3 (list 5)))) 3)
```

More work with **filter**, **map**, **foldr**...

```
;; a leaf-labelled tree (LLT) is a (listof (anyof Num LLT)).
```

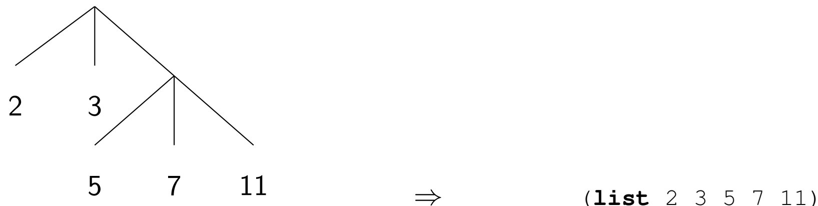
Complete these exercises without recursion on a list. (You will still need to recurse on the LLT.)

Exercise: Write a function to count the leaves of a LLT.

Exercise: Complete depth.

```
;; (depth tree) return the max distance from the root to a leaf of tree.
;; depth: LLT -> Nat
;; Examples:
(check-expect (depth (list 6 7)) 1)
(check-expect (depth (list 2 (list 3 (list 5)))) 3)
```

Sometimes we want to extract the leaves from a leaf-labelled tree. For example:



Exercise: Complete `flatten`. Hint: use the **append** function.

```
;; (flatten tree) return the list of leaves in tree.
;; flatten: LLT -> (listof Num)
;; Examples:
(check-expect (flatten (list 1 (list 2 3) 4)) (list 1 2 3 4))
(check-expect (flatten (list 1 (list 2 (list 3 4)))) (list 1 2 3 4))
```

Be able to work with general trees and leaf-labelled trees.

Write programs that use recursion on trees, and either recursion or higher order functions on lists within the trees.

Create templates from data definitions. Use data definitions and templates to guide design of functions, both recursive and non-recursive.

Wrapping up CS115

I want to highlight:

- The importance of communication, to other programmers but even to yourself.
- That data definitions can guide the design of our programs.

These big ideas transcend language. They will influence your work in CS116 and beyond.

CS116 uses Python. The syntax of this language is less restricted, which means

- There are more ways to say things (which makes things easier to say!)
- There are more ways to say things (there is more syntax to learn)

We will work on

- Writing programs that are longer, or deal with large amounts of data
- Designing programs so they run efficiently

Options for continuing CS

CS major:

- Take CS 116 and then CS 136.
- Talk to a CS advisor about the process.
- Many students have been successful in CS after starting in CS 115.

Computing Technology Option:

- Take CS 116.
- Talk to an advisor to understand your choices.

Note: Participate in course selection! Otherwise, you may not be able to enroll in your preferred courses.