12: 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?

Trees with an arbitrary number of children (subtrees) in each node are called **general trees**.

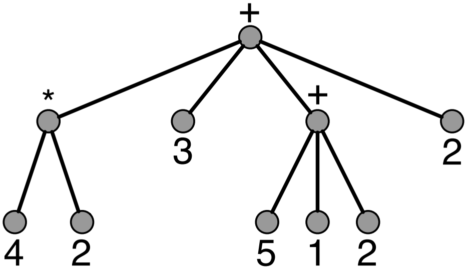
Our example of a general tree will be arithmetic expressions.

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)
```

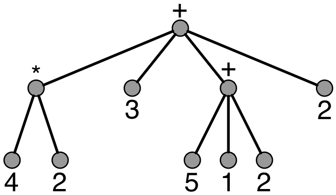


For simplicity, we will restrict the operations to $+$ and $*$.

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).

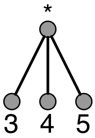
```
;; An Arithmetic Expression (AExp) is one of:
;; * Num
;; * OpNode

(define-struct opnode (op args))
;; An OpNode (operator node) is a
;; (make-opnode (anyof '* '+) (listof AExp)).
```



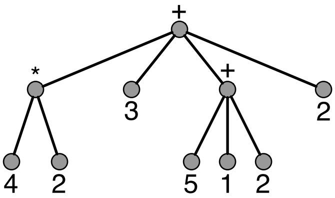
`AExp` is defined using `OpNode` and `OpNode` is defined using `AExp`. This will lead to mutual recursion.

```
3
```

```
(make-opnode '* (list 3 4 5))
```




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



Templates

```
;; An Arithmetic Expression (AExp) is one of:
;; * Num
;; * OpNode

(define-struct opnode (op args))
;; An OpNode (operator node) is a
;; (make-opnode (anyof '* '+) (listof AExp)).
```

What are the templates?

Template writing refresher (from M11)

Follow the data definition. For each part:

- is defined data type, apply it's template
- says "one of", include a `cond`
- is compound data (structure), extract each field
- is a list, extract `first` and `rest`

Do the above recursively.

```
;; (eval exp) evaluates the arithmetic expression exp.
;; Examples:
(check-expect (eval 5) 5)
(check-expect (eval (make-opnode '+ (list 1 2 3 4))) 10)
(check-expect (eval (make-opnode '* (list 2 3 4))) 24)
(check-expect (eval (make-opnode '+ (list 1
                                          (make-opnode '* (list 2 3))
                                          3))) 10)

;; eval: AExp → Num
(define (eval exp)
...)
```

```
;; (eval exp) evaluates the arithmetic expression exp.
;; Examples:
(check-expect (eval 5) 5)
(check-expect (eval (make-opnode '+ (list 1 2 3 4))) 10)
(check-expect (eval (make-opnode '* (list 2 3 4))) 24)
(check-expect (eval (make-opnode '+ (list 1
                                          (make-opnode '* (list 2 3))
                                          3))) 10)

;; eval: AExp → Num
(define (eval exp)
  (cond [(number? exp) exp]
        [(opnode? exp) (apply (opnode-op exp)
                              (opnode-args exp))]))
```

```
;; (apply op args) applies the arithmetic operator op to args.
;; Examples:
(check-expect (apply '+ (list 1 2 3 4)) 10)
(check-expect (apply '* (list 2 3 4)) 24)
(check-expect (apply '+ (list 1 (make-opnode '* (list 2 3)))) 7)
(check-expect (apply '+ (list )) 0)
(check-expect (apply '* (list )) 1)


;; apply: (anyof '+ '*) (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 (1/3)

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

> Condensed trace of aexp evaluation (2/3)

```
⇒ (+ (* 3 (* 4 (apply '* empty)))
      (apply '+ (list (make-opnode '* (list 2 5)))))
⇒ (+ (* 3 (* 4 1))
     (apply '+ (list (make-opnode '* (list 2 5)))))
⇒ (+ 12
     (apply '+ (list (make-opnode '* (list 2 5)))))
⇒ (+ 12 (+ (eval (make-opnode '* (list 2 5)))
           (apply '+ empty)))
⇒ (+ 12 (+ (apply '* (list 2 5))
           (apply '+ empty)))
⇒ (+ 12 (+ (* (eval 2) (apply '* (list 5)))
           (apply '+ empty)))
```

```
⇒ (+ 12 (+ (* 2 (apply '* (list 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 10, we saw how a list could be used instead of a structure.

Here we could use a similar idea to replace the structure `opnode` 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
(list '+ 3 4)
(list '+ (list '* 4 2) 3 (list '+ 5 1 2) 2)
```

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

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 with parameters.

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.

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

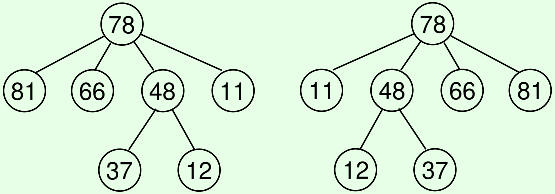
```
(list 'webpage
      (list 'title "CS135: Designing Functional Programs")
      (list 'paragraph "For a course description,"
                       (list 'link "click here." "desc.html")
                       "Enjoy the course!")
      (list 'horizontal-line)
      (list 'paragraph "(Last modified yesterday.)"))
```

Ex. 1

Here is a definition of a generalized tree where any node can have many children:

```
(define-struct gnode (key children))
;; A GT (Generalized Tree) is a (make-gnode Nat (listof GT))
```

Write a function `reverse-gt` which consumes a GT and produces its reverse. Consider a GT node with k children at positions 0 to $(k-1)$. The reverse of this node is the same node except that each child that was at position i is now at position $(k-i)$, for $0 \leq i < k$. The reverse of a GT is the result of each node being reversed.



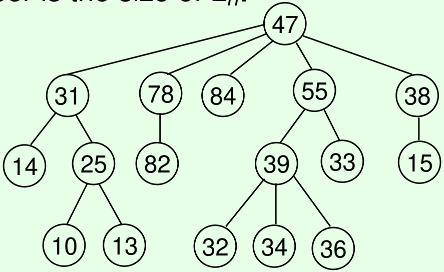
Ex. 2

Here is a definition of a generalized tree where any node can have many children:

```
(define-struct gnode (key children))
;; A GT (Generalized Tree) is a (make-gnode Nat (listof GT))
```

A node's *level* is the number of edges from the root to the node. L_n is the set of all nodes at level n. Write a function `most-populated-level` that consumes a GT and produces a pair based on the set of nodes, L_n, that contains the most nodes. The first member of the pair is n and the second member is the size of L_n.

For example, when called on the tree below, `most-populated-level` produces `(list 2 6)`.



Goals of this module

- You should be able to write mutually recursive functions that consume and process general trees, including general arithmetic expressions.

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?
```