Module 10: General trees

Readings: HtDP, Sections 15 and 16

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

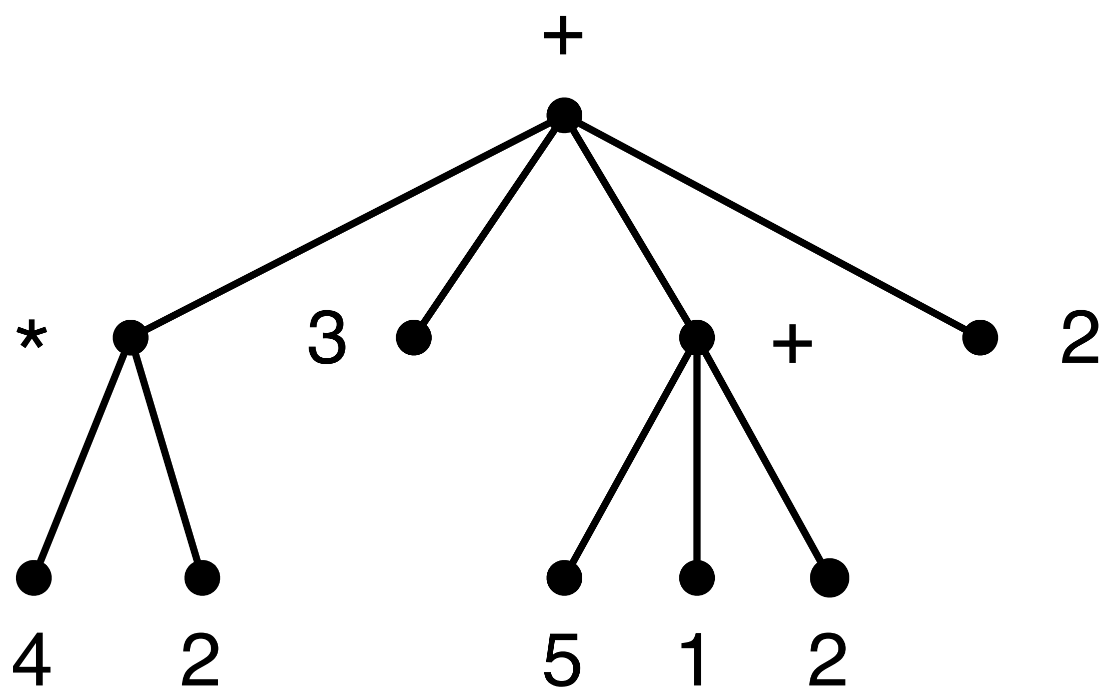
For binary arithmetic expressions, we formed binary trees.

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))
;; An 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. This is called mutual recursion.

Examples of arithmetic expressions:

```
3
(make-ainode '+ (list 3 4))
(make-ainode '* (list 3 4))
(make-ainode '+ (list (make-ainode '* (list 4 2)) 3
                      (make-ainode '+ (list 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-fun ex)
  (cond
    [(number? ex) . . . ]
    [else . . . (ainode-op ex) . . .
          . . . (my-listof-aexp-fun (ainode-args ex)) . . . ]))

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

The function eval

```
;; eval: AExp → Num

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

```
;; apply: (anyof '* '+) (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: (anyof '* '+) (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 '* (list 3 4))
                            (make-ainode '* (list 2 5)))))
⇒ (apply '+ (list (make-ainode '* (list 3 4))
                  (make-ainode '* (list 2 5))))
⇒ (+ (eval (make-ainode '* (list 3 4)))
     (apply '+ (list (make-ainode '* (list 2 5)))))
⇒ (+ (apply '* (list 3 4))
     (apply '+ (list (make-ainode '* (list 2 5)))))
```

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

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

Write a function remainder-n that consumes an AExp (in which all numbers are non-negative integers) and a positive integer n, and produces a new AExp in which all numbers have been replaced with their remainder when divided by n.

```
;; remainder-n: AExp Nat → AExp
;; requires: n > 0
(define (remainder-n ex n)
    ...)
```

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 and (listof AExp).

Each expression is a list consisting of a symbol (the operation) and a list of expressions.

```
;; An Alternate arithmetic expression (AltAExp) is one of:
;; * a Num
;; * (cons (anyof '* '+) (listof AltAExp))
;; Examples:
3
(list '+ 3 4)
(list '+ (list '* 4 2 3) (list '+ (list '* 5 1 2) 2))
```

Templates for AltAExp and (listof AltAExp)

```
(define (my-altaexp-fun ex)
  (cond
    [(number? ex) . . . ]
    [else . . . (first ex) . . .
          . . . (my-listof-altaexp-fun (rest ex)) . . . ]))

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

Some uses of general trees

The contents of organized text and web pages can be stored as a general list.

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

In lab, you will develop templates and write functions for general trees.

Nested lists

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

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

and lists of lists (one level of nesting):

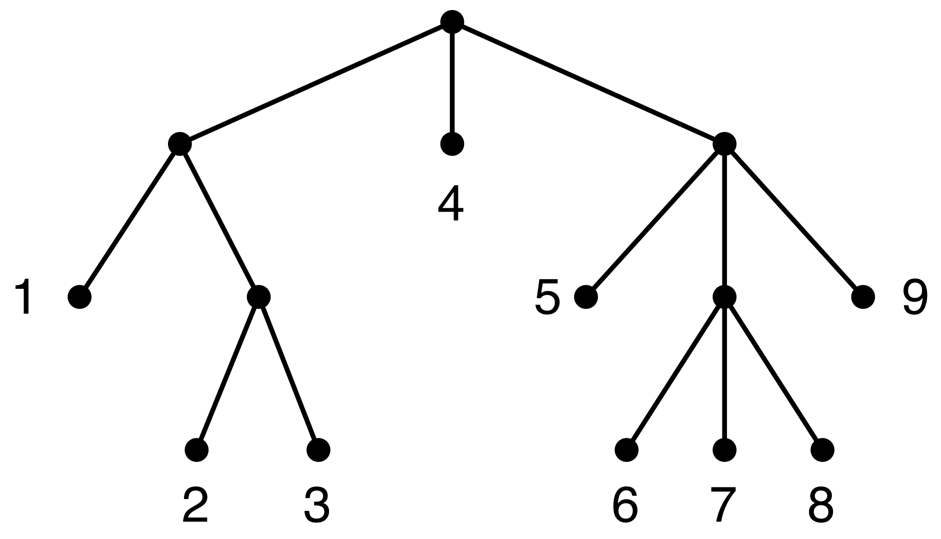
```
(list (list 1 "a") (list 2 "b"))
```

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

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

Nested lists as leaf-labelled trees

It is often helpful 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.



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

Examples of leaf-labelled trees:

```
empty
(list 4 2)
(list (list 4 2) 3 (list 4 1 6))
(list (list 3) 2 (list 5) (list 4 (list 3 6)))
```

Each non-empty 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 leaf-labelled trees

```
;; A leaf-labelled tree (LLT) is one of the following
;; * empty
;; * (cons Num LLT)
;; * (cons LLT LLT) where first LLT is nonempty.
```

The labels could be any non-list Racket value, but we will just use Num here.

Template for leaf-labelled trees

The template follows from the data definition.

```
(define (my-llt-fun l)
  (cond
    [(empty? l) . . . ]
    [(cons? (first l))
         . . . (my-llt-fun (first l)) . . .
         . . . (my-llt-fun (rest l))  . . . ]
    [else . . . (first l) . . . (my-llt-fun (rest l)) . . . ]))
```

The function count-leaves

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

Condensed trace of count-leaves

```
(count-leaves (list (list 'a 'b) 'c))
⇒ (+ (count-leaves (list 'a 'b)) (count-leaves (list 'c)))
⇒ (+ (+ 1 (count-leaves (list 'b))) (count-leaves (list 'c)))
⇒ (+ (+ 1 (+ 1 (count-leaves (list)))) (count-leaves (list 'c)))
⇒ (+ (+ 1 (+ 1 0)) (count-leaves (list 'c)))
⇒ (+ (+ 1 1) (count-leaves (list 'c)))
⇒ (+ 2 (count-leaves (list 'c)))
⇒ (+ 2 (+ 1 (count-leaves (list))))
⇒ (+ 2 (+ 1 0)) ⇒ (+ 2 1) ⇒ 3
```

Flattening a nested list

flatten produces a flat list from a nested list.

```
;; flatten: LLT → (listof Num)
(define (flatten l) . . . )
```

We make use of the built-in Racket function, append, which we examined in Module 7.

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

Remember: You should continue to use cons when constructing a list from a single element and another list.

```
;; (flatten l) produces a flat list from l.
;; flatten: LLT → (listof Num)
(define (flatten l)
  (cond
    [(empty? l) empty]
    [(cons? (first l)) (append (flatten (first l))
                               (flatten (rest l)))]
    [else (cons (first l) (flatten (rest l)))]))
```

Condensed trace of flatten

```
(flatten (list (list 'a 'b) 'c))
⇒ (append (flatten (list 'a 'b)) (flatten (list 'c)))
⇒ (append (cons 'a (flatten (list 'b))) (flatten (list 'c)))
⇒ (append (cons 'a (cons 'b (flatten (list)))) (flatten (list 'c)))
⇒ (append (cons 'a (cons 'b empty)) (flatten (list 'c)))
⇒ (append (cons 'a (cons 'b empty)) (cons 'c (flatten (list))))
⇒ (append (cons 'a (cons 'b empty)) (cons 'c empty))
⇒ (cons 'a (cons 'b (cons 'c empty)))
```

Summary: Structuring data using mutual recursion

Mutual recursion arises when complex relationships among data result in two (or more) definitions referencing each other.

Structures and list may be used as part of the definitions.

In each case:

- create templates from the data definitions and
- create one function for each template.

Goals of this module

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.

Summing up CS 115

With only a few language constructs we have described and implemented ideas from introductory computer science in a brief and natural manner.

We have done so without many of the features (static types, mutation, I/O) that courses using conventional languages have to introduce on the first day. The ideas we have covered carry over into languages in more widespread use.

We hope you have been convinced that the goal of computer science is to implement useful computation in a way that is correct and efficient as far as the machine is concerned, but that is understandable and extendable as far as other humans are concerned.

These themes will continue in CS 116, but new themes will be added, and a new programming language using a different paradigm will be studied.

Looking ahead to CS 116 and beyond

We have been fortunate to work with very small languages (the teaching languages) writing very small programs which operate on small amounts of data.

In CS 116, we will broaden our scope, moving towards the messy but also rewarding realm of the real world.

In subsequent courses we will address such questions as:

- How do we organize a program that is bigger than a few screenfuls, or large amounts of data?
- How do we reuse and share code, apart from cutting-and-pasting it into a new program file?
- How do we design programs so that they run efficiently?

These are issues which arise not just for computer scientists, but for anyone making use of computation in a working environment.

We can build on what we have learned this term in order to meet these challenges with confidence.

Options for continuing CS studies:

- 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.