Module 9: 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

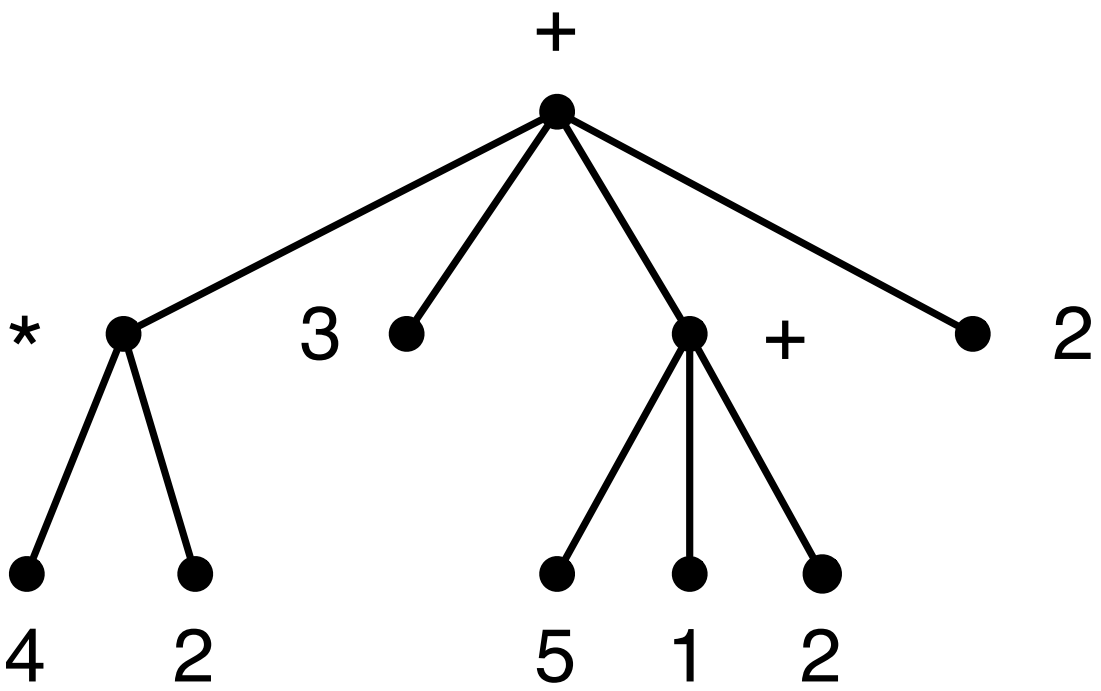
For binary arithmetic expressions, we formed binary trees.

Scheme 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 ae (fn args))

An arithmetic expression (**aexp**) is either

- a number or
- a structure (make-ae f alist), where
  - ⋆ f is a symbol and
  - ⋆ alist is an *aexplist*.

An **aexplist** is either

- empty or
- (cons a alist), where
  - ⋆ a is an *aexp* and
  - ⋆ alist is an *aexplist*.

Each definition depends on the other, and each template will depend on the other.

We first look at a simpler example of **mutual recursion** to understand how to use it.

Mutual recursion

We consider pairs of recursive definitions that refer to *each other*.

The definitions lead to templates that refer to each other, one for each definition.

The templates lead to functions that refer to each other, one for each template.

Defining evens and odds

A new definition of evens, plus a definition of odds:

An **even natural number** is either

- 0 or
- 1 plus an *odd natural number*.

An **odd natural number** is either

- 1 or
- 1 plus an *even natural number*.

These are **mutually-recursive** data definitions.

Templates for evens and odds

```
(define (my-even-fun n)
  (cond
    [(zero? n) ...]
    [else ... (my-odd-fun (sub1 n)) ...]))

(define (my-odd-fun n)
  (cond
    [(= 1 n) ...]
    [else ... (my-even-fun (sub1 n)) ...]))
```

These are **mutually-recursive** functions.

Functions using evens and odds

Since evens depend on odds, to write a function that consumes an even number we need to write a function for evens and a function for odds.

Example 1: produce the number of positive multiples of d from n down to 0, where n is even.

Example 2: produce the number of positive even multiples of d from n down to 0, where n is even.

For both, we modify the templates to take d along for the ride.

We also use the helper function is-divisor?.

```
(define (num-mults-even n d)
  (cond
    [(zero? n) 0]
    [(is-divisor? n d) (+ 1 (num-mults-odd (sub1 n) d))]
    [else (num-mults-odd (sub1 n) d)]))
(define (num-mults-odd n d)
  (cond
    [(= 1 n) (cond [(= 1 d) 1] [else 0])]
    [(is-divisor? n d) (+ 1 (num-mults-even (sub1 n) d))]
    [else (num-mults-even (sub1 n) d)]))
```

Number of even multiples

```
(define (num-even-mults-even n d)
  (cond
    [(zero? n) 0]
    [(is-divisor? n d) (+ 1 (num-even-mults-odd (sub1 n) d))]
    [else (num-even-mults-odd (sub1 n) d)]))

(define (num-even-mults-odd n d)
  (cond
    [(= 1 n) 0]
    [else (num-even-mults-even (sub1 n) d)]))
```

(define-struct ae (fn args))

An arithmetic expression (**aexp**) is either

- a number or
- (make-ae f alist), where
  - ⋆ f is a symbol and
  - ⋆ alist is an *aexplist*.

An **aexplist** is either

- empty or
- (cons a alist), where
  - ⋆ a is an *aexp* and
  - ⋆ alist is an *aexplist*.

Examples of arithmetic expressions:

```
3
(make-ae '+ (list 3 4))
(make-ae '* (list 3 4))
(make-ae '+ (list (make-ae '* '(4 2)) 3
                  (make-ae '+ '(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) ...]
    [(ae? ex) ... (ae-fn ex) ...
                  ... (my-aexplist-fun (ae-args ex)) ... ]))
(define (my-aexplist-fun exlist)
  (cond
    [(empty? exlist) ...]
    [(cons? exlist) ... (my-aexp-fun (first exlist)) ...
                    ... (my-aexplist-fun (rest exlist)) ...]))
```

The function eval

```
;; eval: aexp → num

(define (eval ex)
  (cond
    [(number? ex) ex]
    [else
      (apply (ae-fn ex) (ae-args ex))]))
```

```
;; apply: symbol aexplist → num
(define (apply f exlist)
  (cond [(empty? exlist)
         (cond [(symbol=? f '*) 1]
               [(symbol=? f '+) 0])]
        [(cons? exlist)
         (cond [(symbol=? f '*)
                (* (eval (first exlist)) (apply f (rest exlist)))]
               [(symbol=? f '+)
                (+ (eval (first exlist)) (apply f (rest exlist)))])]))
```

A simplified apply

```
;; apply: symbol aexplist → 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-ae '+ (list (make-ae '* '(3 4))
                        (make-ae '* '(2 5)))))
⇒ (apply '+ (list (make-ae '* '(3 4))
                  (make-ae '* '(2 5))))
⇒ (+ (eval (make-ae '* '(3 4)))
     (apply '+ (list (make-ae '* '(2 5)))))
⇒ (+ (apply '* '(3 4))
     (apply '+ (list (make-ae '* '(2 5)))))
```

$\Rightarrow$ (+ (* (eval 3) (apply '* '(4)))
(apply '+ (list (make-ae '* '(2 5)))))

$\Rightarrow$ (+ (* 3 (apply '* '(4)))
(apply '+ (list (make-ae '* '(2 5)))))

$\Rightarrow$ (+ (* 3 (* (eval 4) (apply '* empty)))
(apply '+ (list (make-ae '* '(2 5)))))

$\Rightarrow$ (+ (* 3 (* 4 (apply '* empty)))
(apply '+ (list (make-ae '* '(2 5)))))

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

$\Rightarrow$ (+ 12 (+ (* 2 (apply '* '(5)))
(apply '+ empty)))

$\Rightarrow$ (+ 12 (+ (* 2 (* (eval 5) (apply '* empty)))
(apply '+ empty)))

$\Rightarrow$ (+ 12 (+ (* 2 (* 5 (apply '* empty)))
(apply '+ empty)))

$\Rightarrow$ (+ 12 (+ (* 2 (* 5 1))
(apply '+ empty)))

$\Rightarrow$ (+ 12 (+ (* 2 5) (apply '+ empty)))

$\Rightarrow$ (+ 12 (+ 10 (apply '+ empty)))

$\Rightarrow$ (+ 12 (+ 10 0)) $\Rightarrow$ (+ 12 10) $\Rightarrow$ 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 ae and the data definitions for aexp and aexplist.

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))

An alternate arithmetic expression (**alt-aexp**) is either

- a number or
- (cons f exlist), where
  - ⋆ f is a symbol and
  - ⋆ exlist is an *alt-aexplist*.

An **alt-aexplist** is either

- empty or
- (cons a exlist), where
  - ⋆ a is an *alt-aexp* and
  - ⋆ exlist is an *alt-aexplist*.

Templates for alt-aexp and alt-aexplist

```
(define (my-alt-aexp-fun ex)
  (cond
    [(number? ex) ...]
    [(cons? ex) ... (first ex) ...
                ... (my-alt-aexplist-fun (rest ex)) ...]))
(define (my-alt-aexplist-fun exlist)
  (cond
    [(empty? exlist) ...]
    [(cons? exlist) ... (my-alt-aexp-fun (first exlist)) ...
                    ... (my-alt-aexplist-fun (rest exlist))...]))
```

Other uses of general trees

For other applications, a node label can be any non-list Scheme value.

```
'(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.)"))
```

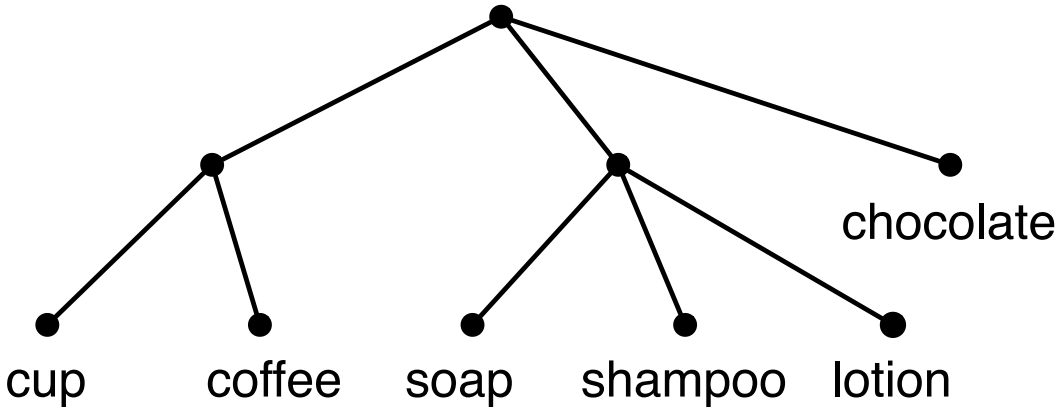
In lab, you will develop templates and write functions for general trees.

Leaf-labelled trees

For some applications:

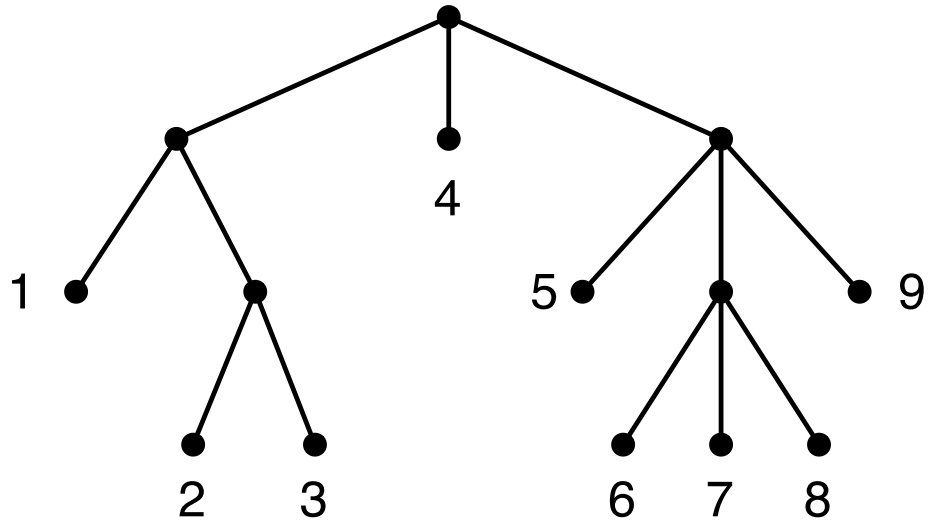
- leaves are used to store data and
- internal nodes are used to show how data is structured.

Items to place in a hotel room are grouped by location.



Representing leaf-labelled trees

A leaf-labelled tree can be represented as a list, where each leaf is a data item (here, an integer) and each internal node is a list of leaf-labelled trees. These were called “nested lists” in Module 6.




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

Examples of leaf-labelled trees:

```
empty
'(4 2)
'((4 2) 3 (4 1 6))
'((3) 2 (5) (4 (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 l1 l2), where
  - ⋆ l1 is a non-empty *llt* and
  - ⋆ l2 is a *llt*
- (cons v l), where
  - ⋆ v is an integer and
  - ⋆ l is a *llt*

We could make v be any non-list Scheme value.

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 '((a b) c))

⇒ (+ (count-leaves '(a b)) (count-leaves '(c)))

⇒ (+ (+ 1 (count-leaves '(b))) (count-leaves '(c)))

⇒ (+ (+ 1 (+ 1 (count-leaves '()))) (count-leaves '(c)))

⇒ (+ (+ 1 (+ 1 0)) (count-leaves '(c)))

⇒ (+ (+ 1 1) (count-leaves '(c)))

⇒ (+ 2 (count-leaves '(c)))

⇒ (+ 2 (+ 1 (count-leaves '())))

⇒ (+ 2 (+ 1 0)) ⇒ (+ 2 1) ⇒ 3

Flattening a nested list

flatten produces a flat list from a nested list.

```
;; flatten: llt → (listof any)
(define (flatten l) . . . )
```

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

(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: llt → (listof any)
;; Produces a flat list from l.
(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 '((a b) c))

⇒ (append (flatten '(a b)) (flatten '(c)))

⇒ (append (cons 'a (flatten '(b))) (flatten '(c)))

⇒ (append (cons 'a (cons 'b (flatten '()))) (flatten '(c)))

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

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

⇒ (append (cons 'a (cons 'b empty)) (cons 'c empty))

⇒ (cons 'a (cons 'b (cons 'c empty)))

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.

Chemical compounds

A compound can be seen as a collection of parts, where each part consists of one or more smaller compounds or elements.

For example, calcium phosphate consists of two parts: three units of calcium and two units of the compound PO_4. The compound PO_4 consists of two parts: one unit phosphorus and four units oxygen.

We define structure and data definitions for compounds, parts, and elements.

For each compound, we will store a symbol and the list of parts.

For each part, we will store a number and an element or compound.

For each element we will store the name and the molar mass.

```
(define-struct compound (name pl))
```

A **compound** is a structure (make-compound n l), where

- n is a symbol and
- l is a *partlist*.

(define-struct part (size eoc))

A **part** is a structure (make-part s e), where

- s is a positive integer representing the number of units of e and
- e is an *element* or a *compound*.

A **partlist** is either

- empty or
- (cons p pl), where
  - ⋆ p is a *part* and
  - ⋆ pl is a *partlist*.

(define-struct element (name mmass))

An **element** is a structure (make-element n m) where

- n is a symbol and
- m is a positive number (the molar mass, that is, the mass of one mole of the substance, 6.02 x 10^{23} atoms).

Templates

```
(define (my-compound-fun c)
    . . . (compound-name c) . . .
    . . . (my-pl-fun (compound-pl c)) . . . )

(define (my-part-fun p)
    . . . (part-size p) . . .
    (cond
        [(element? (part-eoc p)) . . . (my-element-fun (part-eoc p)) . . . ]
        [(compound? (part-eoc p))
            . . . (my-compound-fun (part-eoc p)) . . . ]))
```

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

(define (my-element-fun e)
    ... (element-name e) ...
    ... (element-mmass e) ...)
```

Fixing mass error

Given a compound with an error in the molar mass of an element, we wish to produce a compound with the mass corrected.

```
;; fix-mass: symbol num[>0] compound → compound
;; Produces a compound from c with the element of name el
;; changed to mass of newmass.
(define (fix-mass el newmass c)
  (make-compound
   (compound-name c)
   (fix-mass-pl el newmass (compound-pl c))))
```

We need not only a function for compound, but one for each other template.

```
;; fix-mass-pl: symbol num[>0] partlist → partlist
;; Produces a partlist from l with the element of name
;; el changed to mass of newmass.
(define (fix-mass-pl el newmass l)
  (cond
    [(empty? l) empty]
    [else (cons (fix-mass-part el newmass (first l))
                (fix-mass-pl el newmass (rest l)))]))
```

```
;; fix-mass-part: symbol num[>0] part → part
;; Produces a part from p with the element of name el
;; changed to mass of newmass.
(define (fix-mass-part el newmass p)
  (make-part
    (part-size p)
    (cond
      [(element? (part-eoc p))
       (fix-mass-element el newmass (part-eoc p))]
      [(compound? (part-eoc p))
       (fix-mass el newmass (part-eoc p))])))
```

```
;; fix-mass-element: symbol num[>0] element → element
(define (fix-mass-element el newmass e)
  (make-element
    (element-name e)
    (cond
      [(equal? (element-name e) el) newmass]
      [else (element-mmass e)])))
```

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.