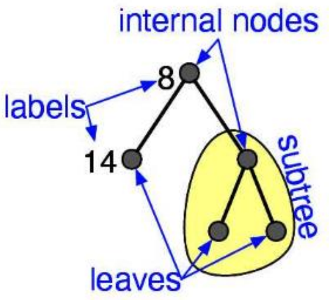
- M11 BT

Tree

a set of **nodes** and **edges**
最顶上的 node —> **root**
A tree is **connected**



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*

Binary tree

A tree with two children for each node
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
```

Path: 轨迹 '(right left right)

Binary search tree: Ordering property: 左边小于右边的 tree

Augmenting tree:

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 M11 P33

Binary expression tree

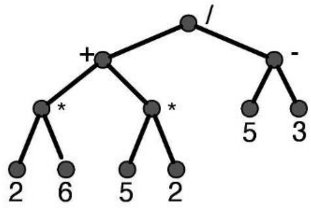
Internal nodes each have exactly two children.

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.



Template

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

(define-struct binode (op left right))
;; A Binary arithmetic expression Internal Node (BINode)
;;     is a (make-binode (anyof '* '+ '/ '-) BinExp BinExp)


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

● M12 Mutual Recursion

Occurs when two or more functions apply each other

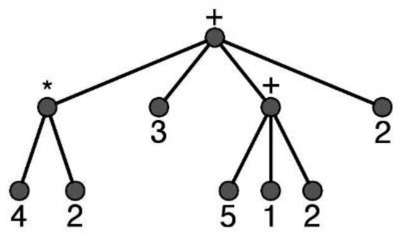
Template

```
;; binexp-template-v2: BinExp → Any
(define (binexp-template-v2 ex)
  (cond [(number? ex) (... ex)]
        [(binode? ex) (binode-template ex)]))

;; binode-template-v2: BINode → Any
(define (binode-template node)
  (... (binode-op node)
       (binexp-template-v2 (binode-left node))
       (binexp-template-v2 (binode-right node))))
```

● M13 General Tree

Trees with an arbitrary number of children (subtrees) in each node



> Representing general trees

```
;; 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)).
```

- M14 Local

Local definitions M14 1/40

The functions and special forms we've seen so far can be arbitrarily nested—except **`define`** and `check-expect`.

So far, definitions have to be made "at the top level", outside any expression.

The Intermediate language provides the special form **`local`**, which contains a series of local definitions plus an expression using them.

```
(local [(define x_1 exp_1) ... (define x_n exp_n)] bodyexp)
```

What use is this?

```
(define (f x y)
  (local [ (define g ...) ]
         (...)))
```

Fresh identifier (fresh name)

A new, unique name that has not been used anywhere else in the program

Reasons to use local (advantage)

Clarity: Naming subexpressions

Efficiency: Avoid recomputation

Encapsulation: Hiding stuff

Scope: Reusing parameters

* **Encapsulation**: the process of grouping things together in a capsule

* **information hiding**: we did not see with structures

* **behavior encapsulation**: evaluating local expression creates new, unique names for the functions just as for the values.

Behaviour encapsulation allows us to move helper functions within the function that usesthem, so they:

1. are invisible outside the function.
2. do not clutter the "namespace" at the top level.
   （在不同的 scope 中可以出现相同的 function name）
3. cannot be used by mistake. （check-expect 不能 check local）、

```
(define(my-func x) 3)
(define (testm m)
  (local
    [(define (my-func x) 5)]
    (my-func m)))

(check-expect (testm 9) 5)
```

function name 重复时

会先调用最内层函数

```
;; Full Design Recipe for isort goes here...
(define (isort lon)
  (local [;; (insert n slon) inserts n into slon, preserving the order
          ;; insert: Num (listof Num) → (listof Num)
          ;; requires: slon is sorted in nondecreasing order
          (define (insert n slon)
            (cond [(empty? slon) (cons n empty)]
                  [(<= n (first slon)) (cons n slon)]
                  [else (cons (first slon) (insert n (rest slon)))]))]

    (cond [(empty? lon) empty]
          [else (insert (first lon) (isort (rest lon)))])))
```

M15 Functions as values

Racket functions are **First class values** (与其他 values 相比可以 take 参数)

Like other values, they can be:
- Consumed by functions (e.g. filter)
- Produced by functions (e.g. make-adder)
- Bound to identifiers
- Stored in lists and structures

Higher order function either consume a function or produce a function

- **Filter**

```
;; (filter pred? lst) -> lst (所有element符合pred)

(define (my-filter pred? lst)
  (cond [(empty? lst) empty]
        [(pred? (first lst))
         (cons (first lst) (my-filter pred? (rest lst)))]
        [else (my-filter pred? (rest lst))]))
```

```
;;foldr表示filter
(define (my-filter pred? lst)
  (foldr (lambda(x rror)
           (cond [(pred? x) (cons x rror)]
                 [else rror]))
         empty
         lst))
```

Functional abstraction 抽象化 is the process of creating abstract functions such as filter. Advantages:

1. Reducing code size
2. Avoiding cut-and-paste.
3. Fixing bugs in one place instead of many.
4. Improving one functional abstraction improves many applications.

- **type variable**： a symbol that stands for some specific, but currently unknown, type

ep. `;; filter: (X → Bool) (listof X) → (listof X)`

polymorphic / generic 适用于很多个 list

- # M16 Functions as abstraction

Abstraction : the process of finding similarities or common aspects

not explicitly recursive : a function where recursion is done via a higher order function such as foldr

Function name is **anonymous**

Higher order function

- **Lambda** (相当于 make function)

```
(lambda(x_1 x_2 ... x_n) exp)
```

- **Map**

```
(map f lst)-> new list
```

new list 中每一个 element 经过 function 处理

length-list 不变

```
(define (my-map f lst)
  (cond [(empty? lst) empty]
        [else (cons (f (first lst)) (my-map f (rest lst)))]))
```

;; foldr 表示 map

```
(define (my-map f lst)
  (foldr (lambda(x rror)
           (cons (f x) rror))    "combine"
         empty                   Base case
         lst))                   Whole list
```

- **Foldr**

```
(foldr f base lst) -> new list
```

Ep.

lst 里面的数字还没执行

将 base 带入最里层 list

```
(foldr + 5 '(1 2 3 4))
-> (+ 1 (foldr + 5 '(2 3 4)))
-> ...
-> (+ 1 (+ 2 (+ 3 (+ 4 (foldr + 5 empty)))))
-> (+ 1 (+ 2 (+ 3 (+ 4 5))))
```

```
(define (my-foldr combine basecase lst)
  (cond [(empty? lst) basecase]
        [else (combine (first lst)
                       (my-foldr combine basecase (rest lst)))]))
```

一般与 lambda(x rror)结合

rror: Result of Recursing On the Rest

- **Foldl**

```
(define (my-foldl combine basecase done-lst)
  (local
    [(define (foldl/acc lst acc)
       (cond [(empty? lst) acc]
             [else (foldl/acc (rest lst)
                              (combine (first lst) acc))]))]
    (foldl/acc done-lst base)))
```

Differences:

the initial value of the accumulator

the computation of the new value of the accumulator, given the old value of the accumulator and the first element of the list.

```
(foldr string-append "2B" '("To" "be" "or" "not"))      r 从前往后
⇒ "Tobeornot2B"
(foldl string-append "2B" '("To" "be" "or" "not"))      l 从后往前
⇒ "notorbeTo2B"
```

```
(foldr + 0 '(1 2 3 4))
⇒ (+ 1 (foldr '(2 3 4)))  ; omitting + and 0, since they don't change
⇒ (+ 1 (+ 2 (foldr '(3 4))))
⇒ (+ 1 (+ 2 (+ 3 (foldr '(4)))))
⇒ (+ 1 (+ 2 (+ 3 (+ 4 foldr '()))))
⇒ (+ 1 (+ 2 (+ 3 (+ 4 0)))) ⇒* 10
```

```
(foldl + 0 '(1 2 3 4))
⇒ (f/acc '(1 2 3 4) 0)  foldl: tail recursion
⇒ (f/acc '(2 3 4) 1)
⇒ (f/acc '(3 4) 3)
⇒ (f/acc '(4) 6)
⇒ (f/acc '() 10) ⇒* 10
```

Similarity:

Both consume a combining function.

Both consume a base value.

Both consume a list.

- **build-list**

```
(build-list num f) -> list
;;length list = num
;;从0到(num-1)执行f
```

- **list-ref**

```
(list-ref list num) -> list中的第num个元素
```

有时 num 要加一减一

- # M17 Generative recursion

Def: recursive cases are <u>generated</u> based on the problem to be solved

depth of recursion :the number of applications of the functionbefore arriving at a base case

```
 (sum-list (list 3 6 5 4))                          ;; 1
⇒ (+ 3 (sum-list (list 6 5 4)))                     ;; 2
⇒ (+ 3 (+ 6 (sum-list (list 5 4))))                 ;; 3
⇒ (+ 3 (+ 6 (+ 5 (sum-list (list 4)))))             ;; 4
⇒ (+ 3 (+ 6 (+ 5 (+ 4 (sum-list (list ))))))        ;; arrived at base case
⇒ (+ 3 (+ 6 (+ 5 (+ 4 0)))) ⇒ ... ⇒ 18
```

- **Quicksort**

The Quicksort algorithm is an example of **divide and conquer**:

- divide a problem into smaller subproblems;
- recursively solve each one:
- combine the solutions to solve the original problem.

Quicksort sorts a list of numbers into non-decreasing order by first choosing a **pivot** lement from the list.

```
(quicksort (listof Any) (...>/<)) -> (排好序的list)
```

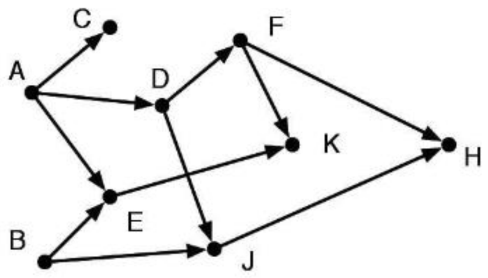
适用于一切能比较大小的 list

Ep. 从小到大排数

```
;;quicksort
(define (my-quicksort lon)
  (cond [(empty? lon) empty]
        [else
         (local
           [(define pivot (first lon))
            (define less (filter (lambda(x) (<= x pivot)) (rest lon)))
            (define more (filter (lambda(x) (> x pivot)) (rest lon)))]
           (append (my-quicksort less)
                   (list pivot)
                   (my-quicksort more)))]))
```

- # M18 Graphs

- **Directed graph**



Def. each edge has a direction
点：nodes / vertices
线(->)：edge

(A,D): A is **out-neighbour** of D
D is **in-neighbour** of A

ADFH：path / route

回路：cycle
DAG：directed acyclic graphs（没有 cycle 的）

（‘A ‘(C D E)）: adjacency list

- **Data definition**

```
;; A Node is a Sym

;; A Graph is one of:
;; * empty
;; * (cons (list v (list w_1 ... w_n)) g)
;;   where g is a Graph
;;         v, w_1, ... w_n are Nodes
;;         v is the in-neighbour to w_1 ... w_n in the Graph
;;         v does not appear as an in-neighbour in g
```

- **Template**

```
;; graph-template: Graph → Any
(define (graph-template g)
  (cond
    [(empty? g) ...]
    [(cons? g)
     (... (first (first g))             ; first node in graph list
      ... (listof-node-template
                (second (first g)))      ; list of adjacent nodes
      ... (graph-template (rest g)) ...)]))
```

● M19 History

Charles Babbage

- Developed mechanical computation for military applications
  - Difference Engine
  - Analytical Engine

Ada Augusta Byron

- Assisted Babbage
- Write article about the use and operation of analytical engine

David Hilbert

- Formalized the axiomatic treatment of Euclidean geometry
- 23 problems, meaning of proof
- Believe the answer would be yes for the first three questions

Kurt Godel

- Incompleteness theorem to Hilbert's questions

Alonzo Church

- Give a 'no' answer to the third question (deciding provability of formula) with his student Kleene
- Created notation to describe functions on the natural numbers - Lambda

Alan Turning

- Two machine halting proof
- Help to break encrypted German radio traffic
- Co-worders developed what we now know to be the world first working electronic computer (Colossue)
- Turning Test in AI

John von Neumann

- Erased the distinction between specification and execution, or program and data

Grace Murray Hopper

Wrote the first compiler

- COBOL, (Common Business-Oriented Language)

- Defined first English-like data processing language

John Backus

- Fortran, language for numerical and scientific computation

John McCarthy

- Lisp ○ Dominant language for AI implementations ○

- Stepper

local

Substitution rule M14 36/40

An expression of the form `(local [d_1 ... d_n] bodyexp)` is rewritten as follows:

- `d_i` will be of the form `(define x_i exp_i)` or `(define (x_i p_1 ... p_m) exp_i)`. In either case, `x_i` is replaced with a fresh identifier (call it `x_i_new`) everywhere in the `local` expression, for $1 \leq i \leq n$.
- The definitions `d_1 ... d_n` are then lifted out (all at once) to the top level of the program, preserving their ordering.
- What remains looks like `(local [] bodyexp')`, where `bodyexp'` is the rewritten version of `bodyexp`. Replace the `local` expression with `bodyexp'`.

All of this (the renaming, the lifting, and removing the `local` with an empty definitions list) is a **single step**.

Terminology associated with local M14 37/40

The **binding occurrence** of a name is its use in a definition, or formal parameter to a function.

The associated **bound occurrences** are the uses of that name that correspond to that binding.

The **lexical scope** of a binding occurrence is all places where that binding has effect, taking note of holes caused by reuse of names.

Global scope is the scope of top-level definitions.

Binding functions

```
(define add2 (make-adder 2))
(define add3 (make-adder 3))

(add2 3) ⇒ 5
(add3 10) ⇒ 13
(add3 13) ⇒ 16
```

```
(define add2 (make-adder 2))
⇒ (define add2 (local [(define (f m) (+ 2 m))] f))
⇒ (define (f_1 m) (+ 2 m)) ; rename and lift out f
  (define add2 f_1)

(add2 3)
⇒ (f_1 3)
⇒ (+ 2 3)
⇒ 5
```

Lambda

从左往右带数

先带 lambda 定义的值，再带之前 define 的 constant

```
(define make-adder
  (lambda (x)
    (lambda (y)
      (+ x y))))
```

```
(define make-adder (lambda (x) (lambda (y) (+ x y))))
((make-adder 3) 4) ⇒    ;; substitute the lambda expression
(((lambda (x) (lambda (y) (+ x y))) 3) 4) ⇒
((lambda (y) (+ 3 y)) 4) ⇒
(+ 3 4) ⇒ 7
```

Foldr / foldl

```
(foldl (lambda(x y) (+ x y y)) 1 (list 3 4 5))
(... (+ 3 1 1) ...)
(... 5 ...)
(... (+ 4 5 5) ...)
(... 14 ...)
(... (+ 5 14 14) ...)
33
```

```
(foldr (lambda(x y) (+ x y y)) 1 (list 3 4 5))
(... (+ 5 1 1) ...)
(... 7 ...)
(... (+ 4 7 7) ...)
(... 18 ...)
(... (+ 3 18 18) ...)
39
```

- # Template

List template

```
(define (list-template lst)
  (cond [(empty? lst) ...]
        [else (... (first lst) ...
                   (list-template (rest lst)) ...)]))
```