# Designing Functional Programs

CS135

Jaiden Ratti

Prof. Dave Tompkins
1221

# Contents

# 1 Functions

The computation of certain mathematical values is the motivation for the direction we move in this course.

Two families of programming languages are Imperative and Functional. Imperative languages are based on frequent changes to data. Functional languages are based on the computation of new values (as opposed to transforming old ones).

CS135 uses Racket, a functional language.

A programming language must solve three problems

1. Syntax: The way we can say things
2. Semantics: What the program means
3. Ambiguity: Valid programs having exactly one meaning

Racket allowed us to develop a semantic model via substitution rules.

Values, Expressions, and Functions

Values are numbers or mathematical objects

Expressions combine values with operators and functions. I.e. $\sin(2\pi)$

Functions generalize similar expressions. I.e. $f(x) = x^2 + 4x + 2$

Functions consist of

- The name of the function
- Parameters
- Expression using the parameters as placeholders

Apply functions only to values and when there is a choice of possible substitution, take the leftmost choice.

Use of Parentheses: Harmonization

Treating infix operators like functions, we don't need parenthesis to specify order of operations.

$3 - 2$ becomes $-(3, 2)$

Now parentheses are only used for function application.

In Racket,

$g(1, 3)$ becomes $(g\ 3\ 1)$

$g(g(1, 3), f(3))$ becomes $(g\ (g\ 1\ 3)\ (f\ 3))$

$(6 - 4)/(5 + 7)$ becomes $(/\ (-\ 6\ 4)\ (+\ 5\ 7))$

Extra parentheses are harmful in Racket.

To evaluate an expression in Racket we use substitution.

```
(* (- 6 4) (+ 3 2))
(* 2 (+ 3 2))
(* 2 5)
10
```

A substitution step rewrites the leftmost subexpression eligible.

Defining Functions

*Note, all code blocks in this document are from lecture material*

```
(define (g x y) (+ x y))
```

We can see a: name, list of parameters, and single body expression.

Examples

```
(define (f x) (sqr x))
(define (g x y) (+ x y))
(define (area r) (* pi (sqr r)))
```

Define is a special form that binds a name to an expression.

All instances of a parameter are replaced in a single step.

Observations

- Changing names of parameters does not change what the function does
- Different functions can use same parameter names
- Parameter order matters

Defining Constants

```
(define k 3)
(define p (sqr k))
```

Note, comments start with a semi-colon.

Helper Functions

Consider a function that determines the distance from current location to the closest of two other locations.

```
;; Find the distance from (cx, cy) to the closer of the two locations, (ax,
    ay) and (bx, by)
(define (distance-to-closer cx cy ax ay bx by)
(min (sqrt (+ (sqr (-ax cx)) (sqr (-ay cy))))
    (sqrt (+ (sqr (- bx cx)) (sqr (- by cy))))))
```

Notice there are two instances of almost identical code. Introduce helper functions.

```
(define (distance-to-closer cx cy ax ay bx by)
(min (distance cx cy ax ay)
    (distance cx cy bx by)))

(define (distance x1 y1 x2 y2)
    (sqrt (+ sqr (- x2 x1)) (sqr (- y2 y1))))
```

Helper functions reduce repeated code by generalizing expressions, factor our complicated calculations, and give meaning to operations.

Helper functions are typically placed after the function that uses them.

*check − expect* is a special form that we use to test our functions.

```
(check-expect (distance 0 0 3 4) 5)
(check-expect (distance 3 4 0 0) 5)
```

```
(check-expect expr-test expr-expected)
```

Where *expr − test* is the expression we want to test, and *expr − expected* is the expected result. This ensures our code works properly.

Scope

The scope of an identifier is where it has effect in the program. The smallest enclosing scope has priority.

```
(define x 3)
(define (f x y)
    (- x y))
(define (g a b)
```

```
    (+ a b x))
(+ (f 2 x) 1)
```

# 2 Design Recipe

Every program is an act of communication. The design recipe is a process for developing a function. Its main purposes include: Helping you understand the problem, write understandable code, and write tested code.

5 Components

1. Purpose: Describe what is being computed
2. Examples: Illustrate examples
3. Contract: Describe the types of inputs and types of outputs. Examples include Num, Int, Nat, Any etc.
4. Definition: Header and Body
5. Tests: Test implementation of specific solution (Examples can also serve as tests but mainly test the purpose)

```
;; (sum-of-squares n1 n2) produces the sum of the squares of n1 and n2
(check-expect (sum-of-squares 3 4) 25)

;; sum-of-squares: Num Num -> Num
(define (sum-of-squares n1 n2)
    (+ (sqr n1) (sqr n2))

;; Tests
(check-expect (sum-of-squares 0 0) 0)
(check-expect (sum-of-squares -2 7) 53)
(check-expect (sum-of-squares 0 2.5) 6.25)
```

Additional Contract Requirements

If there are any constraints required for the function to work, we must specify a requires section in our contract.

Contract Enforcement

Racket uses dynamic typing. Thus the contract is just a comment and only accessible to programmers. Dynamic typing allows for flexibility and confusion!

# 3 Simple Data

We now discuss three other types of values: Booleans, Symbols, and Strings

```
(define x 2)
(< x 5)
```

Is asking, "Is it true that the value of $x$ is less than 5? By substitution rules, we get $(2 < 5) \implies true$.

Booleans (Bool)

$<, >, \leq, \geq, =$ are new functions, each which produce a Boolean value. Boolean values may only be true or false.

Predicates

A function which produces a *Bool* is called a predicate. Many predicate function names end with a ?.

```
(define (can-vote? age)
    (>= age 18))
```

String Predicates

Non-numeric types can have predicates.

```
(string=? "pearls" "gems") => false
(string=? "pearls" "pearls") => true

(string<? "pearls" "swine") => true
...
```

Conditional Expressions

Often expressions should take one value under some conditions, and other values under others.

Sin-squared window can be described by: $f(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 1 \\ \sin^2(x\pi/2) & \text{for } 0 \leq x < 1 \end{cases}$

We can compute this with a conditional expression:

```
(cond [(< x 0) 0]
    [(>= x 1) 1]
    [(< x 1) (sqr (sin (* x pi 0.5)))])
```

Each argument is a question/answer pair. The question is a Boolean expression. The answer is a possible value.

We informally evaluate a *cond* by considering the question/answer pairs in order from top to bottom. Evaluate a question, and if it evaluates to true, the entire *cond* produces the corresponding answer. If it doesn't evaluate to true, proceed to the next question.

If none of the questions evaluate to true? We get an error. But what if we only want to describe some conditions?

We could use a question which always evaluates to true:

```
(define (foo n)
    (cond
        [(> n 0) n]
        [(< 3 7) (- n)]))
```

There is a special keyword for this, *else*

```
(define (foo n)
    (cond
        [(> n 0) n]
        [else (- n)]))
```

There should be at least one test for each possible answer in the conditional expression. When boundary conditions are involved, these should be tested specifically.

```
(define (course-after-cs135 grade)
    (cond
        [(< grade 40) 'CS115]
        [(< grade 50) 'CS135]
        [(< grade 60) 'CS116]
        [else 'CS136]))
```

Four intervals with three boundary points $\implies$ seven tests are required.

We can combine predicates using special forms *and*, *or*, *not*. These all consume and produce boolean values.

*and* has value true when all of its arguments have value true (and is false otherwise)

*or* produces value true if at least one of its arguments is true (and is false otherwise)

*not* produces true if its argument is false (and is false otherwise)

```
(and (> 5 4) (> 7 2)) => true
(or (>= 5 4) (7 2)) => true
(not (= 5 4)) => true
(or (> 4 5) (> 2 7) (< 9 4)) => true
```

Racket only evaluates as many arguments of *and* and *or* as necessary to determine the value.

```
(and (odd? x) (> x 2) (prime? x))
```

Tracing *and*

```
(and false ...) => false
(and true ...) => (and ...)
(and) => true
```

Tracing *or*

```
(or true ...) => true
(or false ...) => (or ...)
(or) => false
```

Closed-box vs Open-box Testing

Closed-box tests: Testing without knowledge of the details of the code

Open-box tests: Testing implementation

Simplifying Conditional Expressions

```
(define (course-after-cs135 grade)
    (cond
        [(< grade 40) 'CS115]
        [(and (>= grade 40) (< grade 50)) 'CS135]
        [(and (>= grade 50) (< grade 60)) 'CS116]
        [(>= grade 60) 'CS136]
```

We can simplify to

```
(define (course-after-cs135 grade)
    (cond
        [(< grade 40) 'CS115]
        [(< grade 50) 'CS135]
        [(< grade 60) 'CS116]
        [else 'CS136]
```

Nested Conditionals

Consider: Free admission for people after 5pm, otherwise the cost is the person's age: 10 and under are charged 5 dollars, and everyone else is charged 10 dollars.

Natural solution

```
(define (admission after5? age)
    (cond
        [after5? 0]
        [else
            (cond
                [(<= age 10) 5]
                [else 10])]))
```

But we can further simplify this

```
(define (admission after5? age)
    (cond
        [after5? 0]
        [(<= age 10) 5]
        [else 10]))
```

Symbolic Data

Symbol is defined using an apostrophe.

```
(define home 'Earth)
(symbol=? home 'Mars) => false
(symbol? home) =? true
```

Strings

Strings are a sequence of characters between double quotes. Examples "hello". Strings are compound data and symbols do not have certain characters in them (no spaces). It is more efficient to compare two symbols than two strings.

```
(string=? "alpha" "bet") => false
(string-append "alpha" "bet") => "alphabet"
(string-length "perpetual") => 9
```

# 4 Semantics

(This is just tracing of different functions which have already been outlined when introduced above).

# 5 Lists

We can create a list in Racket:

```
(define wish-list
    (cons "comics"
        (cons "toys"
            (cons "wii"
                (cons "donkey kong" empty)))))

(length wish-list) => 4
(first wish-list) => "comics"
(empty? wish-list) => false
```

Functions can also produce lists. See *rest*:

```
(rest wish-list) =>
(cons "toys"
    (cons "wii"
        (cons "donkey kong" empty)))
```

Appending is as follows:

```
(cons "bicycle" wish-list) =>
(cons "bicycle"
    (cons "comics"
        (cons "toys"
            (cons "wii"
                (cons "donkey kong" empty)))))
```

What is a List?

A list is a recursive structure defined in terms of a smaller list.

A list of 3 concerts is a concert followed by a list of 2 concerts (etc.)

A list of 1 concert is a concert followed by a list of 0 concerts.

A list of zero concerts is special. We call it the empty list. It is represented in Racket by *empty*.

(cons v lst) creates a list by adding the value $v$ to the beginning of list $lst$.

```
(define concerts0 empty)

(define concerts1 (cons "Waterboys" empty))

(define concerts2 (cons "DaCapo" concerts1))
```

Basic List Constructs

- *empty*: A value representing an empty list
- (*cons* *v* *lst*): Consumes a value and a list; producing a new longer list
- (*first* *lst*): Produces the first value of the list
- (*rest* *lst*): Produces the same list without the first value
- (*empty*? *v*): Produces true if empty, and false otherwise
- (*cons*? *v*): Produces true if it is a cons and false otherwise.
- (*list*? *v*): Equivalent to (*or* (*cons*? *v*) (*empty*? *v*))

To extract the second element of a list we can do:

```
(first (rest lst))
```

Simple Functions on Lists

```
;; (next-concert loc) produces the next concert to attend (or empty) if loc is
    empty

;; next-concert: (listof Str) -> Str
(define (next-concert loc)
    (cond
        [(empty? loc) ""]
        [else (first loc)]))
```

```
;; (same-consec? loc) determines if next two concerts are the same
;; Examples
(check-expect (same-consec? (cons "a" (cons "a" empty))) true)
(check-expect (same-consec? (cons "a" (cons "b" empty))) false)
(check-expect (same-consec? (cons "a" empty)) false)

;; same-consec?: (listof Str) -> Bool
(define (same-consec? loc)
    (and (not (empty? loc))
    (not (empty? (rest loc)))
    (string=? (first loc) (first (rest loc)))))
```

List values are

1. *empty*
2. (*cons* *v* *l*), where *v* is any Racket value, and *l* is a list value.

We will develop data definitions and function templates.

```
;; A (listof Str) is one of:
;; empty
;; (cons Str (listof Str))
```

This is a recursive data definition (the definition refers to itself). A base case cannot refer to itself (*empty*).

We can generalize this

```
;; A (listof X) is one of:
;; empty
;; (cons X (listof X))

;; Can be processed to

(define (listof-X-temlpate lox)
    (cond
        [(empty? lox) ...]
        [(cons? lox) ...]))
```

We can go a step further,

```
(define (listof-X-temlpate lox)
    (cond
        [(empty? lox) ...]
        [(cons? lox) (... (first lox)
                          (listof-X-template (rest lox)))]))
```

Because (rest lox) is of type (listof X), we can use recursion in this manner.

We will write a function to count the number of concerts in a list of concerts.

There are four crucial questions to help think about functions when consuming a list:

1. What should the function produce in the base case?
2. What should the function do in the first element (in a non-empty list)
3. What should calling the function on the rest of the list produce?
4. How should the function combine 2 and 3 to produce the answer for the entire list?

```
(check-expect (count-concerts (cons "a" (cons "b" empty))) 2)

;; count-concerts: (listof Str) -> Nat
(define (count-concerts loc)
    (cond
        [(empty? loc) 0]
        [else (+ 1 (count-concerts (rest loc)))]))
```

This is a recursive function. Look closely at how it traces. It is repeatedly adding 1 for every element in the list, and concludes by adding the final term 0 when the list is empty.

Here is the condensed trace of the example

```
(count-concerts (cons "a" (cons "b" empty)))
(+ 1 (count-concerts (cons "b" empty)))
(+ 1 (+ 1 (count-concerts empty)))
(+ 1 (+ 1 0))
2
```

Producing the sum of all values in the list

```
(define (sum-list loc)
    (cond
        [(empty? loc) 0]
        [else (+ (first loc) (count-concerts (rest loc)))]))
```

Termination

It is important that our functions always terminate. In our examples above, there are two conditions. Either we have reached the base case (and we will terminate). Or we are in a recursive case, thus approaching the base case.

Thinking Recursively

- Get the base case correct

- Assume the function correctly solves a problem of size $n$
- Figure out how to use this solution to solve the problem of size $n+1$

This is similar to induction!

Sometimes each $x$ in (listof $x$) requires further processing.

```
(define (listof-X-template lox)
    (cond
        [(empty? lox) ...]
        [(cons? lox) (... (X-template (first lox)) ...
                          ... (listof-X-template (rest lox)) ...)]))
```

These templates provide basic shape. Later we will see an abstraction mechanism (hof) that can reduce the need for templates.

Patterns of Recursion

What we have seen so far is simple recursion. The form of the code matches the form of the data definition.

Note: There are built-in functions for lists, *length* and *member*? which takes an element of any type and a list, and returns true if the element is in the list (and false otherwise).

Producing Lists from Lists

Consider a list which produces the same list with each number negated.

```
;; (negate-list lon) produces a list with every number in lon negated
(check-expect (negate-list (cons 2 (cons -12 empty)))
              (cons -2 (cons 12 empty)))

;; negate-list: (listof Num) -> (listof Num)
(define (negate-list lon)
    (cond
        [(empty? lon) empty]
        [else (cons (- (first lon)) (negate-list (rest lon)))]))
```

Sometimes computations only make sense with a non-empty list. We can outline this as (ne-listof X) in our design recipe.

Counting Characters in a String

```
(check-expect (count-char/list #\e (string->list "beekeeper")) 5)

;; count-char/list: Char (listof Char) -> Nat
(define (count-char/list ch loc)
    (cound
        [(empty? loc) 0]
        [else (+ (cond [(char=? ch (first loc)) 1]
                       [else 0])
                 (count-char/list ch (rest loc)))]))
```

Wrapper Functions

Simple functions that wrap the main function and take care of housekeeping details.

For example,

```
(define (count-char ch s)
    (count-char/list ch (string->list s)))
```

# 6 Natural Numbers

Definition of Natural Numbers

Peano axioms to define the natural numbers include:

- 0 is a natural number
- For every natural number $n$, $S(n)$ is a natural number

1 can be represented as $S(0)$, 2 as $S(S(0))$ and so on.

$S(n)$ is a successor function. Returns the next natural number.

*add*1 in Racket is our successor function.

```
(add1 0) => 1
(add1 1) => 2
(add1 2) => 3
```

Thus

```
;; A Nat is one of:
;; 0
;; (add1 Nat)
```

Nat-template

```
(define (nat-template n)
    (cond
        [(zero? n) ...]
        [else (... n ...
                ... (nat-template (sub1 n)) ...)]))
```

The function *countdown* consumes a natural number $n$ and produces a decreasing list of all natural numbers less than or equal to $n$.

```
(check-expect (countdown 2) (cons 2 (cons 1 (cons 0 empty))))

;; countdown: Nat -> (listof Nat)
(define (countdown n)
    (cond
        [(zero? n) (cons 0 empty)]
        [else (cons n (countdown (sub1 n)))]))
```

Intervals of Natural Numbers

We use $\mathbb{Z}$ to denote the integers. We can add subscripts to define subsets (intervals).

For example $\mathbb{Z}_{\geq 0}$ defines the non-negative integers.

We can use this to make the function *countdown* – *to*

```
(check-expect (countdown-to 4 2) (cons 4 (cons 3 (cons 2 empty))))

;; countdown-to: Int Int -> (listof Int)
;;       requires: n >= base
(define (countdown-to n base)
    (cond
        [(= n base) (cons base empty)]
        [else (cons n (countdown-to (sub1 n) base))]))
```

This also works with negative numbers.

Repetition in other Languages

In imperative languages we often use loops. In Racket we use recursion.

Note, we can use *reverse* to reverse a list. But don't rely on this to fix recursive solutions.

# 7 More Lists

```
;; (sort lon) sorts the elements of lon in non-decreasing order

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

We define insert to be a recursive helper function that consumes a number and a sorted list, and inserts the number into the sorted list.

```
(define (insert n slon)
    (cond
        [(empty? slon) ...]
        [else (... (first slon) ...
                   (insert n (rest slon)) ...)]))
```

We will answer our four questions to determine how to structure the above code. When *slon* is empty, the result is the list just containing $n$. Number 2: the sorted list with $n$ inserted in the correct place. Number 3: $n$ is the first number in the result if it is less than or equal to the first number in *slon*.

```
(define (insert n slon)
    (cond
        [(empty? slon) (cons n empty)]
        [(<= n (first slon)) (cons n slon)]
        [else (cons (first slon) (insert n (rest slon)))]
```

Combining the *sort* and *insert* functions gives us insertion sort.

List Abbreviations

The expression

```
(cons exp1 (cons exp2 (... (cons expn empty) ... )))
```

can be abbreviated as

```
(list exp1 exp2 ... expn)
```

```
(cons 4 (cons 2 (cons 1 (cons 5 empty))))
<=>
(list 4 2 1 5)
```

*cons* contains exactly two arguments. The length is known only when the program is running.

*list* consumes any number of arguments. The length is known when we write the program.

```
(second my-list)
<=>
(first (rest my-list))
```

The same is defined from *third* to *eighth*

Lists Containing Lists

```
(list (list 3 4))
(list (cons 3 (cons 4 empty)))
(cons (list 3 4) empty)
(cons (cons 3 (cons 4 empty)) empty)
```

We can have lists of lists

```
;; A Payroll is one of:
;; empty
;; (cons (list Str Num) Payroll)

(list (list "Asha" 7000)
        (list "Joe" 100000)
        (list "Sam" 50000)

;; A TaxRoll is one of:
;; empty
;; (cons (list Str Num) TaxRoll)

(list (list "Asha" 700)
        (list "Joe" 16500)
        (list "Sam" 5500))
```

We arrive at

```
;; compute-taxes: Payroll -> TaxRol

(define (compute-taxes payroll)
    (cond [(empty? payroll) empty]
            [(cons? payroll)
            (cons (list (name (first payroll))
                        (tax-payable (amount (first payroll))))
                  (compute-taxes (rest payroll)))]))
```

We will later see nested lists. These are lists that contain lists, and so on to arbitrary depth.

Dictionaries

A dictionary contains a number of unique keys, each with an associated value.

Examples:

- Stocks: Keys are symbols, values are prices
- Dictionary: Keys are words, values are definitions

Operations we can perform on dictionaries include lookups, adding (key,value) pairs, and removing (key,value) pairs.

Association List

We can store the pair as a two-element list. We can use natural numbers as keys and strings as values.

```
;; An Association List (AL) is one of:
;; empty
;; (cons (list Nat Str) AL)
;; Requires: each key (Nat) is unique
```

We can use this to make the template

```
(define (al-template alst)
    (cond [(empty? alst) ...]
            [else (... (first (first alst)) ...
                       (second (first alst)) ...
                       (al-template (rest alst)))]))
```

Lookup Operation

```
(check-expect (lookup-al 2 (list (list 8 "Asha")
                                 (list 2 "Joe")
                                 (list 5 "Sam"))) "Joe")
(check-expect (lookup-al 1 (list 8 "Asha")) false)
```

```
(define (key kv) (first kv))
(define (val kv) (second kv))

;; (lookup-al k alst) produces the value corresponding to the key k, or false
    if there is no k

(define (lookup-al k alst)
    (cond
        [(empty? alst) false]
        [(= k (key (first alst))) (val (first alst))]
        [else (lookup-al k (rest alst))]))
```

Two-Dimensional Data

```
(mult-table 3 4) =>
(list (list 0 0 0 0)
        (list 0 1 2 3)
        (list 0 2 4 6))
```

We can make this

```
(define (cols-to c r nc)
    (cond
        [(>= c nc) empty]
        [else (cons (* r c) (cols-to (add1 c) r nc))]))

(define (rows-to r nr nc)
    (cond
        [(>= r nr) empty]
        [else (cons (cols-to 0 r nc) (rows-to (add1 r) nr nc))]))

(define (mult-table nr nc)
    (rows-to 0 nr nc))
```

We will now consider more complicated types of recursion

Case 1: Just one List

```
(define (my-append lst1 lst2)
    (cond
        [(empty? lst1) lst]
        [else (cons (first lst1)
                    (my-append (rest lst1) lst2))]))
```

*ls*2 is just along for the ride.

Case 2: Processing in Lockstep

To process two lists in lockstep, they must be the same length.

Because the two lists must be the same length, (*empty*? *lst*1) $\iff$ (*empty*? *lst*2)

Our template is thus,

```
;; lockstep-template: (listof X) (listof Y) -> Any
;;      Requires: (length lst1) = (length lst2)
(define (lockstep-template lst1 lst2)
    (cond
        [(empty? lst1) ...]
        [else
            (... (first lst1) ... (first lst2) ...
                (lockstep-template (rest lst1) (rest lst2)) ... )]))
```

An example is dot product.

```
(define (dot-product lon1 lon2)
    (cond
    [(empty? lon1) 0]
    [else (+ (* (first lon1) (first lon2))
             (dot-product (rest lon1) (rest lon2)))]))
```

Case 3: Processing at Different Rates

Now all four possibilities of empty/nonempty are possible.

```
(define (twolist-template lon1 lon2)
    (cond
        [(and (empty? lon1) (empty? lon2)) ...]
        [(and (empty? lon1) (cons? lon2)) ...]
        [(and (cons? lon1) (empty? lon2)) ...]
        [(and (cons? lon1) (cons? lon2)) ...]))
```

An example is merging two sorted lists.

```
(check-expect (merge (list 3 4) (list 2)) (list 2 3 4))

(define (merge lon1 lon2)
    (cond
        [(and (empty? lon1) (empty? lon2)) empty]
        [(and (empty? lon1) (cons? lon2)) lon2]
        [(and (cons? lon1) (empty? lon2)) lon1]
        [(and (cons? lon1) (cons? lon2))
        (cond
            [(< (first lon1) (first lon2))
            (cons (first lon1) (merge (rest lon1) lon2))]
            [else (cons (first lon2) (merge lon1 (rest lon2)))])]))
```

Testing List Equality

```
(define (list=? lst1 lst2)
    (cond
        [(and (empty? lst1) (empty? lst2)) true]
        [(and (empty? lst1) (cons? lst2)) false]
        [(and (cons? lst1) (empty? lst2)) false]
        [(and (cons? lst1) (cons? lst2))
        (and (= (first lst1) (first lst2))
            (list=? (rest lst1) (rest lst2)))]))
```

There is built-in list equality (*equal?*)

```
(check-expect (at-least? 3 'red (list 'red 'blue 'red 'green)) false)
(check-expect (at-least? 1 7 (list 5 4 0 5 3 7)) true)

;; at-least? Nat Any (listof Any) -> Bool
(define (at-least? n elem lst)
    (cond
        [(zero? n) true]
        [(equal? (first lst) elem) (at-least? (sub1 n) elem (rest lst))]
        [else (at-least? n elem (rest lst))]))
```

# 8 Patterns of Recursion

All the recursion done so far has been simple recursion. We will now use accumulative recursion, and learn how to recognize mutual recursion and generative recursion.

For certain applications ($max - list$), simple recursion can make up to $2^n - 1$ recursive applications on a list of length $n$. This is exponential blowup.

Fast $O(n)$

```
(define (max-list-v1 lon)
    (cond
        [(empty? (rest lon)) (first lon)]
        [else (max (first lon) (max-list-v1 (rest lon)))]))
```

Slow $O(2^n)$

```
(define (max-list-v2 lon)
    (cond
        [(empty? (rest lon)) (first lon)]
        [(> (first lon) (max-list-v2 (rest lon))) (first lon)]
        [else (max-list-v2 (rest lon))]))
```

Accumulative Recursion

We can pass the largest value we've seen through a parameter called an accumulator.

```
;; max-list/acc: (listof Num) Num -> Num
(define (max-list/acc lon max-so-far)
    (cond
        [(empty? lon) max-so-far]
        [(> (first lon) max-so-far)
        (max-list/acc (rest lon) (first lon))]
        [else (max-list/acc (rest lon) max-so-far)]))

(define (max-list-v3 lon)
    (max-list/acc (rest lon) (first lon)))
```

The accumulatively recursive function usually has a wrapper function that sets the initial values of the accumulators.

Reversing a list using simple recursion is $O(n^2)$ worst-case. Using an accumulator it becomes $O(n)$.

Fibonacci

Using simple recursion,

```
(define (fib n)
    (cond
        [(< n 2) n]
        [else (+ (fib (- n 1)) (fib (- n 2)))]))
```

This works for small $n$, but suffers from $\phi^n$, where $\phi = \frac{1+\sqrt{5}}{2}$

Mutual Recursion

Mutual recursion occurs when two or more functions apply each other. $f$ applies $g$ and $g$ applies $f$.

```
(define (game state)
    (a-turn state))

(define (a-turn state)
    (cond
        [(a-won? state) 'A-WON]
        [else (b-turn (strategy-a state))]))

(define (b-turn state)
    (cond
        [(b-won? state) 'B-WON]
        [else (a-turn (strategy-b state))]))
```

Generative Recursion

We can use our knowledge from MATH135 to solve for *gcd* in Racket.

```
;; euclid-gcd: Nat Nat -> Nat
(define (euclid-gcd n m)
    (cond
        [(zero? m) n]
        [else (euclid-gcd m (remainder n m))]))
```

The arguments in the recursive call were generated by doing a computation on $m$ and $n$. Thus we have generative recursion. Easy to get wrong and hard to debug.

# 9 Structures

Racket's structures define trivial functions automatically.

*Posn* is a built-in structure with two fields that contain numbers for $x$ and $y$ coordinates. Can make a *Posn* using a constructor function, $make - posn$

```
;; make-posn: Num Num -> Posn

(define my-posn (make-posn 4 3))
```

*Posn* has two selector functions. These produce the field which has the same name of the selector.

```
(posn-x (make-posn 4 3)) => 4
(posn-y (make-posn 4 3)) => 3
```

We have type predicates for this as well. $(posn?\ my - posn) \implies$ true.

Producing a *Posn*

```
;; offset-a-little: Num Num -> Posn
(define (offset-a-little x y)
    (make-posn (+ x 3) (+ y 3)))
```

We can make custom structures.

```
(define struct-name (field0 field1 ... fieldn))

(define-struct inventory (desc price available))
```

This automatically creates several functions (Constructor ($make - inventory$), Predicate, Selectors).

```
(define lentils (make-inventory "dry lentils" 2.49 42))

(check-expect (total-value (make-inventory "rice" 5.50 6)) 33.00)

;; We can extract with

(inventory-desc lentils) => "dry lentils"
(inventory-price lentils) => 2.49
```

The design recipe of custom structures looks like

```
(define-struct inventory (desc price available))
;; an Inventory is a (make-inventory Str Num Nat)
;;      Requires: price >= 0
```

Structure Templates

```
(define (inventory-template item)
    ( ... (inventory-desc item)
    ... (inventory-price item)
    ... (inventory-available item) ... ))
```

Structures can automatically generate significant code as opposed to lists.

We will see quote notation to compact lists.

```
'(1 2 3) => (list 1 2 3)
'(a b c) => (list 'a 'b 'c)
'(1 ("abc" earth) 2) => (list 1 (list "abc" 'earth) 2)
'(1 (+ 2 3)) => (list 1 (list '+ 2 3))
```

# 10 Binary Trees

The expression $((2*6)+(5*2))/(5-3)$ can be represented as a tree. / would be the root node, + and - its child nodes, * and * +'s child nodes etc.

A tree is a set of nodes and edges where an edge connects two distinct nodes. One node must be the root, every node other than the root is a parent or child. A tree must be connected.

Other important terms.

- Leaves: Nodes with no children
- Internal Nodes: Nodes that have children
- Labels: Data attached to node
- Ancestors of node $n$: $n$ itself, the parent of $n$, and the parents of parents of $n$, etc.
- Descendants of $n$: All the node that have $n$ as an ancestor (including $n$).
- Subtree rooted at $n$: All of the descendants of $n$.

A binary tree is a tree with at most two children for each node. Fundamental part of computer science.

Binary tree 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

;; bt-template: BT -> Any
(define (bt-template t))
```

Count the nodes in a tree.

```
;; (count-nodes tree k) counts the number of nodes in the tree that have a key
    equal to k

;; count-nodes: BT Nat -> Nat
(define (count-nodes tree k)
    (cond
        [(empty? tree) 0]
        [(node? tree) (+ (cond
                           [(= k (node-key tree)) 1]
                           [else 0])
                           (count-nodes (node-left tree) k)
                           (count-nodes (node-right tree) k))]))
```

Increment Keys

```
;; increment: BT -> BT
(define (increment tree)
    (cond
        [(empty? tree) empty]
        [(node? tree) (make-node (add1 (node-key tree))
                                 (increment (node-left tree))
                                 (increment (node-right tree)))]))
```

Searching Binary Trees

We will produce true if the key is in the tree, and false otherwise.

If the root node contains the key we are looking for, produce true. Otherwise, recursively search in the left subtree and the right subtree (if either recursive search finds the key, produce true, otherwise false).

```
;; search: Nat BT -> Bool
(define (search-bt k tree)
    (cond
        [(empty? tree) false]
        [(= k (node-key tree)) true]
        [else (or (search-bt k (node-left tree))
                  (search-bt k (node-right tree)))]))
```

We will show the path to a key. Assume for now there are no duplicates in the tree.

```
;; search-bt-path: Nat BT -> (anyof false (listof Sym))
(define (search-bt-path k tree)
    (cond
        [(empty? tree) false]
        [(= k (node-key tree)) '()]
        [(list? (search-bt-path k (node-left tree)))
        (cons 'left (search-bt-path k (node-left tree)))]
        [(list? (search-bt-path k (node-right tree)))
        (cons 'right (search-bt-path k (node-right tree)))]
        [else false]))
```

But this has double calls to $search - bt - path$.

Let's improve it

```
;; search-bt-path-v2: Nat BT -> (anyof false (listof Sym))
(define (search-bt-path-v2 k tree)
    (cond
        [(empty? tree) false]
        [(= k (node-key tree)) '()]
        [else (choose-path-v2 (search-bt-path-v2 k (node-left tree))
                              (search-bt-path-v2 k (node-right tree)))]))
(define (choose-path-v2 left-path right-path)
    (cond
        [(list? left-path) (cons 'left left-path)]
        [(list? right-path) (cons 'right right-path)]
        [else false]))
```

We can make searching more efficient by creating a Binary Search Tree. Placing the keys in a particular order can improve the running time of our search.

```
;; A Binary Search Tree (BST) is one of:
;; empty
;; a Node

(define-struct node (key left right))
;; A Node is a (make-node Nat BST BST)
;; Requires: key > every key in left BST
;;           key < every key in right BST
```

$key > \forall key \in left$

$key < \forall key \in right$.

This property also holds in each subtree.

```
;; Example
(make-node 5
    (make-node 1 empty empty)
    (make-node 7
```

```
        (make-node 6 empty empty)
        (make-node 14 empty empty)))
```

We now save one recursive function application.

```
;; search-bst: Nat BST -> Bool
(define (search-bst n t)
    (cond
        [(empty? t) false]
        [(= n (node-key t)) true]
        [(< n (node-key t)) (search-bst n (node-left t))]
        [(> n (node-key t)) (search-bst n (node-right t))]))
```

Adding to a BST

If our tree is empty, then we create the one node. If our tree already contains the value we want to add, we just produce the tree. Otherwise, our value must go either in the left or right subtree. We only need to make one recursive function application.

We can augment our node to have additional data.

```
(define-struct node (key val left right))
```

An augmented BST can serve as a dictionary.

```
(define-struct node (key val left right))
;; A Binary Search Tree Dictionary (BSTD) is either:
;; empty
;; (make-node Nat Str BSTD BSTD)

(define (search-bst-dict k t)
    (cond
        [(empty? t) false]
        [(= k (node-key t)) (node-val t)]
        [(< k (node-key t)) (serach-bst-dict k (node-left t))]
        [(> k (node-key t)) (serach-bst-dict k (node-right t))]))
```

We can use trees to represent arithmetic.

```
;; A binary artihmetic expression (BinExp) is one of:
;; a Num
;; A BInode

(define-struct binode (op left right))
;; BInode is a (make-binode (anyof '* '+ '/ '-) BinExp BinExp)

(make-binode '/
    (make-binode '+ (make-binode '* 2 6)
                    (make-binode '* 5 2))
    (make-binode '- 5 3))
```

```
;; eval: BinExp -> Num
(define (eval ex)
    (cond
        [(number? ex) ex]
        [(binode? ex) (eval-binode ex)]))

(define (eval-binode node)
    (cond
        [(symbol=? '* (binode-op node))
        (* (eval (binode-left node)) (eval (binode-right node)))]
        [(symbol=? '/ (binode-op node))
        (/ (eval (binode-left node)) (eval (binode-right node)))]
        [(symbol=? '+ (binode-op node))
        (+ (eval (binode-left node)) (eval (binode-right node)))]
```

```
        [(symbol=? '- (binode-op node))
        (- (eval (binode-left node)) (eval (binode-right node)))]
```

We can refactor this to be,

```
(define (eval ex)
    (cond
        [(number? ex) ex]
        [(binode? ex) (eval-binode (binode-op ex)
                                   (eval (binode-left ex))
                                   (eval (binode-right ex)))]))

(define (eval-binode op left-val right-val)
    (cond
        [(symbol=? op '*) (* left-val right-val)]
        [(symbol=? op '/) (/ left-val right-val)]
        [(symbol=? op '+) (+ left-val right-val)]
        [(symbol=? op '-) (- left-val right-val)]
```

# 11 Mutual Recursion

Mutual recursion occurs when two or more functions apply each other ($f$ applies $g$ and $g$ applies $f$).

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

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

Mutual Recursion on a List

```
(define (keep-alternates lst)
    (cond
        [(empty? lst) empty]
        [else (cons (first lst) (drop-alternates (rest lst)))]))

;; (drop-alternates lst) drops the first element of the list and
;; keeps the alternating elements from the rest
(define (drop-alternates lst)
    (cond
        [(empty? lst) empty]
        [else (keep-alternates (rest lst))]))
```

# 12 General Trees

General trees have an arbitrary number of children (subtrees).

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

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

This is mutual recursion.

```
(check-expect (eval (make-opnode '+ (list 1 2 3 4))) 10)

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

;; 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)))]
```

We can use an alternate data definition.

```
;; An alternate arithmetic expression (AltAExp) is one of:
;; a Num
;; (cons (anyof '* '+) (listof AltAExp))
```

We now have the beginnings of a Racket interpreter.

# 13 Local Definitions

Functions can be arbitrarily nested.

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

Heron's formula: $\sqrt{s(s-a)(s-b)(s-c)}$, where $s = (a+b+c)/2$.

*Local* provides a natural way to bring the definition and use together

```
(define (t-area-v4 a b c)
    (local [(define s (/ (+ a b c) 2 ))]
        (sqrt (* s (- s a) (- s b) (-s c)))))
```

A define within a local expression may reuse a name which has already been bound to another value.

We define the informal substitution rule for local to replace every name defined in local with a fresh name.

Benefits of using *local*

- Clarity
- Efficiency
- Encapsulation
- Scope

Clarity: Meaningful names

```
(define (distance p1 p2)
    (local [(define delta-x (- (coord-x p1) (coord-x p2)))
            (define delta-y (- (coord-y p1) (coord-y p2)))] (sqrt (+ (sqr
        delta-x) (sqr delta-y)))))
```

Efficiency: Avoiding Recomputation

Encapsulation: Hiding stuff. More of this in CS246.

*Local* can also work with mutually recursive functions.

```
;; my-even?: Nat -> Bool
(define (my-even? n)
    (local [(define (is-even? n)
                (cond
                    [(= n 0) true]
                    [else (is-odd? (sub1 n))]))
            (define (is-odd? n)
                (cond
                    [(= n 0) false]
                    [else (is-even? (sub1 n))]))]
        (is-even? n)))
```

Example: Insertion Sort

```
(define (isort lon)
    (local [(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)))])))
```

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 here that binding has effect.

Global scope is the scope of top-level definitions.

# 14 First Class Values

Racket is a functional programming language, primarily because Racket's functions are first class values.

Functions can be consumed as function arguments, produced as function results, bound to identifiers, stored in lists and structures.

Examples:

Consider

```
(define (eat-apples lst)
    (cond
        [(empty? lst) empty]
        [(not (symbol=? (first lst) 'apple))
        (cons (first lst) (eat-apples (rest lst)))]
        [else (eat-apples (rest lst))]))

(define (keep-odds lst)
    (cond
        [(empty? lst) empty]
        [(odd? (first lst))
        (cons (first lst) (keep-odds (rest lst)))]
        [else (keep-odds (rest lst))]))
```

We can abstract out the differences.

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

The built-in function filter performs the same actions as $my - filter$.

*Filter* is an example of a higher order function. Higher order functions consume a function or produce a function.

Using my-filter

```
(define (keep-odds lst) (my-filter odd? lst))

(define (not-symbol-apple? item) (not (symbol=? item 'apple)))
(define (eat-apples lst) (my-filter not-symbol-apple? lst))
```

Advantages of functional abstraction

1. Reducing code size
2. Avoiding duplicate code
3. Fix bugs in one place instead of many
4. Improving one functional abstraction improves many applications

The body of *local* can produce such a function as a value.

Here is a small example.

```
(define (make-adder n)
    (local [(define (f m) (+ n m))]
        f))

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

(add2 3) -> 5
(add3 10) -> 13
```

Using *local* gives us a way to create semi-custom functions on the spot to use elsewhere.

We can use this method for *eval* and *apply*.

We can use the contract for a function as its type.

```
;; (lookup-al key al) finds the value in al corresponding to key
;; lookup-al: Sym (listof (list Sym (Num Num -> Num))) ->
;; anyof false (Num Num -> Num))
(define (lookup-al key al)
    (cond
        [(empty? al) false]
        [(symbol=? key (first (first al))) (second (first al))]
        [else (lookup-al key (rest al))]))
```

We will usually have a dependency between the type of the predicate and the type of list.

To express this, we use a type variable, usually X, and use it in different places where the same type is needed.

*Filter* produces a list of the same type that it consumes.

Therefore the contract is

```
;; filter: (X -> Bool) (listof X) -> (listof X)
```

*Filter* is polymorphic or generic.

Many difficulties in using higher order functions can be overcome by careful attention to contracts.

# 15 Functional Abstraction

Abstraction is the process of finding similarities and forgetting unimportant differences.

A function is an example.

Racket provides a mechanism for constructing a nameless function which can then be used as an argument.

```
(local [(define (name-used-once x_1 ... x_n) exp)]
    name-used-once)

;; can also be written as

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

Lambda can be thought of as a make-function.

We can use lambda to replace

```
(define (eat-apples lst)
    (filter (local [(define (not-symbol-apple? item)
                          (not (symbol=? item 'apple)))]
                      not-symbol-apple?)
                    lst))

(define (eat-apples lst)
    (filter (lambda (item) (not (symbol=? item 'apple))) lst))
```

We can also simplify make-adder.

```
(define (make-adder n)
    (local [(define (f m) (+ n m))]
        f))

(define (make-adder n)
    (lambda (m) (+ n m)))
```

Lambda ($\lambda$) is available in Intermediate Student with Lambda. We will see this its important in the last lecture.

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

(define make-adder (lambda (x) (lambda (y) (+ x y))))
((make-adder 3) 4)
(((lambda (X) (lambda (y) (+ x y))) 3) 4)
((lambda (y) (+ 3 y)) 4)
(+ 3 4) -> 7
```

We can also look to abstract the commonality for map.

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

(my-map f (list x_1 x_2 ... x_n))
```

```
;; =
(list (f x_1) (f x_2) ... (f x_n))
```

This function, *my − map* is also a built in higher order function *map*.

We can also abstract from functions that consume lists and produce simple values.

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


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

*foldr* is a built-in function in Intermediate Student with Lambda.

The tracing looks like

```
(foldr f b (list x_1 x_2 ... x_n))
;; =
(f x_1 (f x_2 (... (f x_n b))))

(foldr string-append "2B" '("To" "be" "or" "not"))
;; =
"Tobeornot2B"
```

*foldr* is short for "fold right" and it can be used to implement *map*, filter and other higher order functions.

*foldr* consumes two parameters: one is an element in the list which is an argument to *foldr*, and one is the result of reducing the rest of the list.

The first argument to the function contributes 1 to the count but its actual value is irrelevant.

```
(define (count-symbols lst) (foldr (lambda (x rror) (add1 rror)) 0 lst))

(lambda (x rror) (add1 rror))
;; ignore its first argument

;; Its second argument is the Result of Recursing On the Rest (rror) of the
   list
```

We can use *foldr* to produce cons expressions.

```
(define (negate-list lst)
    (foldr (lambda (x rror) (cons (- x) rror)) empty lst))

;; my-map using foldr

(define (my-map f lst)
    (foldr (lambda (x rror) (cons (f x) rror)) empty lst))
```

Imperative languages tend to provide inadequate support for recursion (usually provide loop constructs).

Higher order functions cover many of the common uses of these looping constructs.

Anything done with the list template can be done using *foldr*, without explicit recursion.

*foldl* is defined in the Intermediate Student language and above.

```
(define (my-fodl combine base lst0)
    (local [(define (foldl/acc lst acc)
```

```
            (cond
                [(empty? lst) acc]
                [else (foldl/acc (rest lst)
                                 (combine (first lst) acc))]))]))

(define (sum-list lon) (my-foldl + 0 lon))
(define (my-reverse lst) (my-foldl cons empty lst))
```

The tracing looks like

```
(foldl f b (list x_1 x_2 ... x_n))
=
(f x_n (f x_n-1 (... (f x_1 b))))

(foldl string-append "2B" '("To" "be" "or" "not")) -> "notorbeTo2B"
```

*foldl* is short for "fold left"

Another built-in higher order function is $build - list$.

This consumes a natural number $n$ and a function $f$, and produces the list

```
(list (f 0) (f 1) ... (f (sub1 n)))
```

Examples:

```
(build-list 4 (lambda (x) x))
;; =
(list 0 1 2 3)

(build-list 4 (lambda (x) (* 2 x)))
;; =
(list 0 2 4 6)
```

We can easily write our own

```
(define (my-build-list n f)
    (local [(define (list-from i)
              (cond
                  [(>= i n) empty]
                  [else (cons (f i) (list-from (add1 i)))]))]))
```

# 16 Generative Recursion

Simple, accumulative, and mutual recursion are ways of deriving code whose form parallels a data definition.

Generative recursion is more general.

It is much harder to come up with such solutions to problems.

Example revisited: GCD

```
(define (euclid-gcd n m)
    (cond
        [(zero? m) n]
        [else (euclid-gcd m (remainder n m))]))
```

This follows from the MATH135 proof of the identity.

An application terminates if it can be reduced to a value in finite time.

For a non-recursive function, it is easy to argue that it terminates.

It is not clear what to do for recursive functions.

Our functions using simple recursion terminate because we are always making recursive applications on smaller instances whose size is bounded below by the base case.

We can thus bound the depth of recursion.

As a result, the evaluation cannot go on forever.

In the case of $euclid - gcd$, our measure of progress is the size of the second argument.

If the first argument is smaller than the second argument, the first recursive application switches them, which makes the second argument smaller.

After that the second argument always gets smaller in the recursive application (since $m > n \mod m$), but it is bounded by below.

Thus any application of $euclid - gcd$ has a depth of recursion bounded by the second argument.

Termination is sometimes hard

```
;; collatz: Nat -> Nat
(define (collatz n)
    (cond
        [(= n 1) 1]
        [(even? n) (collatz (/ n 2))]
        [else (collatz (+ 1 (* 3 n)))]))
```

Hoare's 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 element from the list.

The easiest pivot to select from a list $lon$ is $(firstlon)$.

A function which tests whether another item is less than the pivot is

```
(lambda (x) (< x (first lon)))
```

The first subproblem is then

```
(filter (lambda (x) (< x (first lon))) lon)
```

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

Termination of quicksort follows from the fact that both subproblems have fewer than the original list.

Degenerative quicksort works best when the two recursive function applications are on arguments about the same size.

When one recursive function application is always on an empty list, the pattern of recursion is similar to the worst case of insertion sort (number of steps is roughly proportional to the square of the length of the list).

# 17 Graphs

A directed graph consists of a collection of nodes (also known as vertices) together with a collection of edges.

Binary trees and expression trees were both directed graphs of a special type where an edge represented a parent-child relationship.

Graphs are general data structures that can model many situations.

Given an edge $(v, w)$, we say that $w$ is an out-neighbour of $v$, and $v$ is an in-neighbour of $w$.

A sequence of nodes $v_1, v_2, \ldots, v_k$ is a path of length $k-1$ if $(v_1, v_2), (v_2, v_3), \ldots, (v_{k-1}, v_k)$ are all edges.

If $v_1 = v_k$, this is called a cycle.

Directed graphs without cycles are called DAGs (Directed Acyclic Graphs).

We can represent a node by a symbol and associate with each node a list of its out-neighbours.

This list is called the adjacency list representation.

```
(define g
    '((A (C D E))
        (B (E J))
        (C ())
        (D (F J))
        (E (K))
        (F (K H))
        (H ())
        (J (H))
        (K ())))
    )
```

The data definition looks like

```
;; 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
```

The template for graphs

```
;; graph-template: Graph -> Any
(define (graph-template g)
    (cond
        [(empty? g) ...]
        [(cons? g)
        (... (first (first g))
        ... (listof-node-template
                    (second (first g)))
        ... (graph-template (rest g)) ...)]))
```

We can use this template to write a function that produces the out-neighbours of a node.

```
;; neighbours: Node Graph -> (listof Node)
;; requires: v is a node in g
(define (neighbours v g)
    (cond
        [(empty? g) (error "Node not found")]
        [(symbol=? v (first (first g))) (second (first g))]
        [else (neighbours v (rest g))]))
```

A path in a graph can be represented by an ordered list of the nodes on the path.

We will design a function that consumes a graph plus origin and destination nodes, and produces a path from the origin to the destination if one exists.

Simple recursion doesn't work, must use generative recursion.

Backtracking algorithms try to find a path from an origin to a destination.

If the initial attempt does not work, the algorithm "backtracks" and tries another choice.

Eventually either a path is found, or all possibilities are exhausted.

We need to apply $find-path$ on each of the out-neighbours of a given node.

The *neighbours* function gives us a list of all the out-neighbours associated with that node.

We should write $find-path/list$ which consumes a list of nodes and will apply $find-path$ to each one until it either finds a path to the destination or exhausts the list.

We will create two mutually recursive functions, $find-path$ and $find-path/list$.

```
;; (find-path orig dest g) finds path from orig to dest in g if it exists
;; find-path: Node Node Graph -> (anyof (listof Node) false)
(define (find-path orig dest g)
    (cond
        [(symbol=? orig dest) (list dest)]
        [else (local
            [(define nbrs (neighbours orig g))
            (define ?path (find-path/list nbrs dest g))]
        (cond [(false? ?path) false]
            [else (cons orig ?path)]))]))

;; (find-path/list nbrs dest g) produces path from
;; an element of nbrs to dest in g, if one exists
;; find-path/list: (listof Node) Node Graph -> (anyof (listof Node) false)
(define (find-path/list nbrs dest g)
    (cond
        [(empty? nbrs) false]
        [else (local
            [(define ?path (find-path (first nbrs) dest g))]
            (cond [(false? ?path)
                (find-path/list (rest nbrs) dest g)]
                [else ?path]))]))
```

Backtracking in implicit graphs forms the basis of many artificial intelligence programs, though they generally add heuristics to determine which neighbour to explore first.

$find-path$ always terminates for directed acyclic graphs.

It is possible for the graph to not terminate if there is a cycle in the graph

```
'((A (B))
    (B (C))
    (C (A))
    (D ()))
```

We can use accumulative recursion to solve the problem of $find-path$ not terminating if there are cycles in the graph.

We need a way of remembering what nodes have been visited (along a given path).

Our accumulator will be a list of visited nodes, we must avoid visiting a node twice.

```
;; find-path/list: (listof Node) Node Graph (listof Node)
;; -> (anyof (listof Node) false)
(define (find-path/list nbrs dest g visited)
    (cond
    [(empty? nbrs) false]
```

```
    [(member? (first nbrs) visited)
                (find-path/list (rest nbrs) dest g visited)]
    [else (local [(define ?path (find-path/acc (first nbrs)
                                               dest g visited))]
            (cond [(false? ?path)
                (find-path/list (rest nbrs) dest g visited)]
            [else ?path]))]))

;; find-path/acc: Node Node Graph (listof Node)
;; -> (anyof (listof Node) false)
(define (find-path/acc orig dest g visited)
    (cond
    [(symbol=? orig dest) (list dest)]
    [else (local [(define nbrs (neighbours orig g))
                  (define ?path (find-path/list nbrs dest g
                  (cons orig visited)))]
        (cond [(false? ?path) false]
            [else (cons orig ?path)]))]))

(define (find-path orig dest g)
    (find-path/acc orig dest g '()))
```

Cycling has been solved but this is pretty inefficient. If there is no path from the origin to the destination, $find - path$ will explore every path, and there could be an exponential number of them.

We can add failures and successes.

```
;; find-path/list: (listof Node) Node Graph (listof Node) -> Result
(define (find-path/list nbrs dest g visited)
    (cond
        [(empty? nbrs) (make-failure visited)]
        [(member? (first nbrs) visited)
            (find-path/list (rest nbrs) dest g visited)]
        [else (local [(define result (find-path/acc (first nbrs)
                                                    dest g visited))]
                (cond [(failure? result)
                        (find-path/list (rest nbrs) dest g
                                        (failure-visited result))]
                      [(success?  result) result]))]))

;; find-path/acc: Node Node Graph (listof Node) -> Result
(define (find-path/acc orig dest g visited)
    (cond
        [(symbol=? orig dest) (make-success (list dest))]
        [else (local [(define nbrs (neighbours orig g))
                      (define result (find-path/list nbrs dest g
                      (cons orig visited)))]
                (cond
                    [(failure? result) result]
                    [(success? result)
                           (make-success (cons orig
                                               (success-path result)))]))]))

;; find-path: Node Node Graph -> (anyof (listof Node) false)
(define (find-path orig dest g)
    (local [(define result (find-path/acc orig dest g empty))]
        (cond
            [(success? result) (success-path result)]
            [(failure? result) false])))
```

This runs much faster on diamond graphs.

# 18 Computing History

Charles Babbage (1791-1871) developed mechanical computation for military applications.

Ada Augusta Byron (1815-1852) helped Babbage and wrote articles describing the operation and use of the Analytical Engine

David Hilbert (1862-1943) formalized the axiomatic treatment of Euclidean geometry. Hilbert's 23 problems

The meaning of proof

Axiom: A statement accepted without proof. For example, $\forall n : n + 0 = n$

Proposition: A statement we'd like to prove. For example, "The square of any even number is even"

Formula: A statement expressed with an accepted set of symbols and syntax. For example, $\forall n(\exists k : n = k + k, \exists m : m + m = n * n)$

Proof: A finite sequence of axioms (basic true statements) and accepted derivation rules (e.g. $\phi$ and $\phi \to \sigma$ yield $\sigma$).

Theorem: A mathematical statement $\phi$ together with a proof deriving $\phi$.

Hilbert's questions:

Is mathematics complete? Meaning for any formula $\phi$, if $\phi$ is true, then $\phi$ is provable.

Is mathematics consistent? Meaning for any formula $\phi$, there aren't proofs of both $\phi$ and $\neg\phi$.

Is there a procedure to, given a formula $\phi$, produce a proof of $\phi$, or show there isn't one?

Hilbert believed the answers would be "yes".

Kurt Gödel (1906-1978) answers to Hilbert. Any axiom system powerful enough to describe arithmetic on integers is not complete. If it is consistent, its consistency cannot be proved within the system.

Sketch of his proof.

Define a mapping between logical formulas and numbers.

Use it define mathematical statements saying "This number represents a valid formula", "This number represents a sequence of valid formulae". "This number represents a valid proof", "This number represents a provable formula"

Construct a formula $\phi$ represented by a number $n$ that says "The formula represented by $n$ is not provable". The formula $\phi$ cannot be false, so it must be true but not provable.

What remained of Hilbert's questions.

Is there a procedure which, given a formula $\phi$, either proves $\phi$, shows it false, or correctly concludes $\phi$ is not provable?

The answer to this requires a precise definition of "a procedure", in other words, a formal model of computation.

Alonzo Church (1903-1995) set out to answer "no". He created notation to describe functions on the natural numbers.

Church and Kleene's notation.

The class of all $x$ satisfying a predicate $f : \hat{x}f(x)$. They couldn't do this on typewriters so it became $\lambda x$.

Numbers from nothing (Cantor-style).

$0 \equiv \emptyset$ or $\{\}$

$1 \equiv \{\emptyset\}$

$2 \equiv \{\{\emptyset\}, \emptyset\}$

In general $n$ is represented by the set containing the sets representing $n - 1, n - 2, \ldots, 0$.

$0 \equiv \lambda f.\lambda x.x$ (returns the identify function)

$1 \equiv \lambda f.\lambda x.fx$ (returns the same function)

$2 \equiv \lambda f.\lambda x.f(fx)$ (returns $f$ composed with itself)

The lambda calculus is a general model of computation.

Church proved that there was no computational procedure to tell if two lambda expressions were equivalent.

Alan Turing (1912-1954) defined a different model of computation, and chose a different problem to prove uncomputable. Resulted in a simpler and more influential proof.

Turing showed how to encode a function, $f$, so that it can be placed on the tape along with its data, $x$. He then showed how to write a different function, $u$, so that $(ufx) \equiv (fx)$ (for any $f$). He called $u$ the universal computing machine.

He then assumed that there was a machine that could process such a description and tell whether the coded machine would halt (terminate) or not on its input.

Using this machine, one can define a second machine to act on this information.

The second machine uses the first machine to see if its input represents a coded machine which halts when fed its own description.

If so, the second machine runs forever; otherwise it halts.

Feeding the description of the second machine to itself creates a contradiction: it halts $\iff$ it doesn't halt.

So the first machine cannot exist.

Turing's model bears a closer resemblance to an intuitive idea of real computation.

Turing made further contributions to hardware and software (Turing test).

John von Neumann (1903-1957) is a founding member of the Institute for Advanced Study at Princeton

Grace Murray Hopper (1906-1992) wrote the first compiler.

John Backus (1924-2007) designed FORTRAN. Won the Turing Award in 1978 and proposed a functional programming language for parallel/distributed computation.

John McCarthy (1927-2011) designed and implemented Lisp, taking ideas from the lambda calculus and the theory of recursive functions.