CS 135
Course Notes

University of Waterloo

Nolan Zurek

Functions

INTRO: Values, Expressions, Functions

- Values are numbers or other mathematical objects (for now)
  - Ex. 1, $\frac{5}{2}$, π
  - Racket has unbounded integers: they can be as big as desired
  - Racket can store both exact values (rational numbers, integers) and inexact values (π, $\sqrt{2}$)
    - It is better to use exact values if possible
    - Inexact values are flagged with a "#" in the terminal
- Expressions combine values and operators
  - Ex. $5 + 2$, $\sin 5\pi/6$, $\sqrt{2}$
- Functions generalize a certain sequence of operators and values
  - Ex. $f(a, b) = a^2 + b^2$
  - Functions definitions have three parts
    - Name (f, g, etc.)
    - Parameters (x, y, etc.)
    - Algebraic expression (expression that relates parameters to an output)
- The parameters in the algebraic expression are used as placeholders for future values
- An application of a function supplies arguments for the parameters
- A function is evaluated by substitution: arguments are evaluated before they are passed into functions, which are themselves evaluated (etc) until everything has been evaluated

Canonical Form

- Representing functions in different ways can lead to ambiguity, and different results for expressions that are mathematically equivalent
- So, racket has a canonical form for functions
  - Canonical form: a preferred notation that must be adhered to
  - There are two rules:
    - Functions can only take values as arguments
    - Functions are evaluated and substituted left to right
  - This way, there is only one way/order an expression can be evaluated

Expressions in racket

- In regular math, parentheses are used for function application ($f(x)$) and specifying the order of operations ($x/(x+1)$)
- We can combine these two uses by treating infix operators (operators that sit between values like $+$, $-$) as functions, which they are.
  - Ex. $+(1,4)$ is the same as $1+4$
  - Doing this has the advantage of remove the need to memorize or get everyone to agree to a certain order of operations
- In racket, the syntax for the expression $a+b$ is `(+ a b)`
  - $(a+b*c)/d$ is written as `(/ (+ g (* b c)) d)` in racket
- When translating math expressions, five things need to be taken into account: the order of parameters matters, the order of operations matter, the right function needs to be used, the right variable names need to be used, and negation matters (use `(- x)`)

Functions in racket

```
(define (myFunction x y) (+ x (sqr y)))
```

- Functions in racket have
  - A name for the function (`myFunction`), called the identifier
  - A list of parameters (`a` , `b`)
  - A single body expression (`(+ x (sqr y))`), which defines what the function does
- When a user-defined function is evaluated, the values passed as parameters are substituted into the function, then it is evaluated
- A constant can be defined by creating a function with no parameters and an expression that always evaluates a certain value
  - Ex. `(define k 3)` binds `k` to the value 3
  - A "variable" basically, in a language that dOeSN't hAve vArIaBLeS

Writing Code Efficiently

- Constants should be defined when a certain value needs to be used many places in a function. This saves time as its value will only need to be changed once if its value needs to be updated
- Helper functions are functions that perform often-used subtasks. These often minimize duplicated code, make modifying the code more efficient, and make functions more readable.
  - These usually placed after the function that uses them, unless they define constants

Scope

- If a constant, function, or parameter is defined in multiple places, it may have different values
- Two different scopes: global and function
- In racket, the smallest enclosing scope is the one that has priority
- DrRacket has a tool that can be used to determine the scope of a variable

The design recipe 😫

```
;; (sum-of-squares p1 p2) produces the sum of the squares of p1 and p2

;; Examples:
(check-expect (sum-of-squares 3 4) 25)
(check-expect (sum-of-squares 0 2.5) 6.25)

;; sum-of-squares: Num Num -> Num

(define (sum-of-squares p1 p2)
   (+ (* p1 p1)
      (* p2 p2)))

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

Assignment Header (assignments only)

The assignment header should contain the name of the student, the name of the assignment being submitted, and the date of submission.

```
;; -----------------------------------
;; CS 135 Assignment 0
;; Nolan Zurek
;; September 1 2021
;; -----------------------------------
```

Purpose

Describes what the function actually does

```
;; (sum-of-squares p1 p2) produces the sum of the squares of p1 and p2
```

The purpose should reference the names of all of the parameters of the function in a way that indicates what they do.

Examples

Illustrates the typical use of the function

```
;; Examples:
(check-expect (sum-of-squares 3 4) 25)
(check-expect (sum-of-squares 0 2.5) 6.25)
```

`(check-expect x y)` tests whether x = y, where x is usually a function being tested.

If the function returns inexact values, `(check-within x y z)` is used instead, where z is the tolerance to check within

Contract

Describes what kind of arguments the function takes and what value it returns

```
;; sum-of-squares: Num Num -> Num
```

`Any` can be used if any data type is acceptable

If more specification is required (for example, a parameter must be greater than 0), it needs to be indicated in the contract.

In these cases, the phrase "requires:" should be used, the actual names of the parameters should be referenced (not their positions), and the relationships between them should be expressed in mathematical notation.

```
;; (sum-of-square-roots q1 q2) computes the sum of the square roots of q1 and
q2

;; sum-or-square-roots: Num Num -> Num
;; requires:
;;   q1 >= 0
;;   q2 >= 0
```

It is also possible that a function may output only a certain set of values of a given type. In that case, `anyof` should be used:

```
;; foo: Num -> (anyof Str Bool Num)
```

When representing a list of items, use `(listof X Y Z)` if the list contains any number of elements of the types `X`, `Y`, and `Z`, and list `(list X Y Z)` if the list always contains that number of items in that order.

Data definition and template (optional)

If the program contains a data structure that isn't a `list` or a `ne-list`, its data definition and template must be included in the design recipe.

```
;; A (listof X) is one of:
;; * empty                    ; base definition
;; * (cons X (listof X))      ; recursive definition
```

Definition

Actual implementation of the function in racket

```
(define (sum-of-squares p1 p2)
   (+ (* p1 p1)
      (* p2 p2)))
```

The names of the parameters should accurately describe what they do. If space doesn't permit, they should be explained in the purpose.

Tests

A representative set of function applications and their expected values (edge cases)

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

These are more to make sure the function works in all cases, not the typical use-case. `check-within` should also be used if necessary.

A guideline for tests is that they should make sure all of the code in a program runs. This can be seen in DrRacket: un-run code is highlighted in black. For conditional statements, this means a test for every `cond` solution, as well as for each boundary case.

For lists, lengths of 0 (empty), 1, and more than one should all be tested.

Examples count as tests.

The design recipe should be written in the order it appears: purpose (draft) → examples -> contract → purpose (draft) → definition → tests. Data definitions can appear anywhere between the start of the program and the use of the data type.

Simple Data

Predicates and Booleans

A **predicate** is a function that evaluates to a boolean

```
(= x y)      ; returns true if x = y
(> x y)      ; returns true if x > y
(< x y)      ; returns true if x < y
(>= x y)     ; returns true if x > y or if x = y
(<= x y)     ; returns true if x < y or if x = y

;;for strings
(string=? x y), (string>? x y),  ;etc
```

With the exception of the above, predicates often and with ?

```
(positive? x), (negative? x), (zero? x)
(even? x), (odd? x)
(boolean? x), (string? x), (number? x), (integer? x), (char? x)
```

There is an `(equal? x y)` predicate, but it should only be used if the types of x and y are unknown and may be different. If you know they will be the same type, use the type-specific predicate instead.

Predicates (and anything that evaluates to a boolean) can be combined using **logic gates** / **boolean operators**: `not`, `and`, and `or`.

```
(not x)      ; returns false if x is true and true if x is false
(and x y)    ; returns true if x and y are both true, false otherwise
(or x y)     ; returns true if either x or y are true, false otherwise
```

Short-circuit evaluation occurs when the computer evaluates a part of boolean operator that immediately determines the outcome. Structuring expressions to favor this makes programs more efficient because less code is evaluated.

```
(and false true)    ; computer stops at false
(or true false)     ; computer stops at true
```

Conditional expressions

`Cond` statements are conditional statements that contain a number of predicate-expression (question-answer) pairs (denoted inside `[ ]`. For each predicate, if it evaluates to true, its corresponding expression is run; otherwise, the next predicate is evaluated.

```
(define (ssqw x)                    ; sin-squared window function
  (cond                             ; beginning of conditional expression
    [(< x 0) 0]                     ; if x < 0, expression evaluates to 0
    [(>= x 1) 1]                    ; if x >= 1, expression evaluates to 1
    [(< x 1)                        ; if x = 1
     (sqr (sin (* x pi 0.5)))]))    ; expression evaluates to sin^2(x*pi/2)
```

In this case, any numerical value of x will be correct. However, if there are holes in the conditional statement, all of the predicates may evaluate to false; this causes an error. To avoid this, an `else` **statement** can be written that will evaluate if everything above it evaluates to false.

```
(define (abs n)         ;absolute value of n function
    (cond               ; beginning of conditional expression
        [(> n 0) n]     ; if n > 0, abs(n) = n
        [else -n]))     ; elsewise, abs(n) = -n
```

Because conditional statements short-circuit once an predicate evaluates to true, there is no need to perform the same tests in a later predicate-expression pair.

Conditionals can be nested, but it's considered to be bad style because it is hard to read. Nested conditionals can always be flattened by moving the inner predicate-expression pairs into the outer conditional statement.

Tests for conditional functions should include a test for every case of the function as well as a test for every overlap case. DrRacket highlights code that was not run automatically, so you can know what code hasn't been tested.

Symbols

Symbols are a type of value that are comprised of an immutable sequence of characters, denoted `'name`

```
(define home-city 'Edmonton)
```

They can be compared using `(symbol=? x y)`

```
(symbol=? 'Hello 'Bonjour)    ; false
(symbol=? 'Hola 'Hola)        ; true
```

Symbols are often used for short labels that won't need to be manipulated.

Strings

Strings are sequences of characters denoted by `" "`.

```
(define greeting "good say, sir!")
```

Strings are stored as a sequence of characters, making it a **compound data type** (a symbol is not stored this way). As such, strings have more functions than symbols and are more easily manipulated.

```
(string=? "alpha" "bet")          ; false
(string<? "alpha" "bet")          ; true
(string-append "alpha" "bet")     ; "alphabet"
(string-length "u wot m8")        ; 8
(string-upcase "the licc")        ; "THE LICC"
(substring "Apple" 1 3)           ; "pp"
```

Datatypes Summary

Type	Abbreviation	Examples
Number	Num	1, -3.5, (sqrt 2)
Integer	Int	2, -4

Type	Abbreviation	Examples
Natural Number	Nat	2, 5, 129
Boolean	Bool	true, false
Character	Char	#\A, #\?
String	Str	"yes", "hello"
Symbol	Sym	'CS135, 'male
Any type	Any	-23.4, false, "lol"

Semantics

Modeling

Programs have a precise meaning and effect. A **model** of a programming language describes how a program would behave based on the code. Racket (and functional languages in general) are really easy to model because they have few underlying constructs compared to imperative languages.

Spelling Rules

- Only certain characters and combinations of characters can be used to name identifiers (anything followed by `(define`)
- Identifiers can't contain any of these: `( ) , ; { } [ ] ‘ ’ “ ”.`
- Identifiers can't start with "`'`" followed by a valid identifier

Grammars

- **Syntax**: How we express ideas
- **Semantics**: The ideas that we express
- **Ambiguity**: When a syntactically correct sentence has more than one valid semantically valid meaning
- **Grammars**: Specific rules the enforce syntax
  - Grammars reduce ambiguity in statements
  - Ex. English: {sentence} = {subject} {verb} {object}
  - Ex. Racket: {definition} = (define ({variable} {variable} ...) {expression})

Racket's Semantic Model

Semantic Model: A model that predicts the result of a running program. Racket's semantic model continuously simplifies the program using **substitution**. Specifically, each finds the leftmost sub-expression eligible for rewriting, and rewrites it by the rules we are about to describe. Every substitution step yields a valid program.

Tracing a program: reducing the program step-by-step according to the semantic rules of the language.

Applying built-in functions

Formal rule: `(f v1 ... vn)` ⇒ `v` where `f` is a built-in function and `v` is the value of `f(v1, . . . , vn)`.

```
(+ 3 5)         ; ⇒ 8
(expt 2 10)     ; ⇒ 1024
```

Applying user-defined functions

Formal rule: `(f v1 ... vn) ⇒ exp'` (the values passed into the function are evaluated, then the function is replaced with its expression with those evaluations substituted in)

```
(define (term x y) (* x (sqr y)))

(term (- 3 1) (+ 1 2)) ⇒
(term 2 (+ 1 2)) ⇒
(term 2 3) ⇒
(* 2 (sqr 3)) ⇒
(* 2 9) ⇒
18
```

Applying constants

Formal rule: `id ⇒ val` where `(define id val)` occurs to the left.

The definition of a constant is removed from the program once its value has been substituted in in order to reduce useless code repetition.

```
(define x 3) (define y (+ x 1)) y ⇒
(define y (+ 3 1)) y ⇒
(define y 4) y ⇒
4
```

Substitution of Cond expressions

There are three formal rules:

1. `(cond [false exp] ...) ⇒ (cond ...)`
2. `(cond [true exp] ...) ⇒ exp`
3. `(cond [else exp]) ⇒ exp`

```
(define n 5) (define x 6) (define y 7)

(cond [(even? n) x][(odd? n) y]) ⇒
(cond [(even? 5) x][(odd? n) y]) ⇒
(cond [false x][(odd? n) y]) ⇒
(cond [(odd? n) y]) ⇒
(cond [(odd? 5) y]) ⇒
(cond [true y]) ⇒
y ⇒
7
```

Substitution for `and` and `or`

Substitution rules for `and`

1. `(and false ...) ⇒ false`
2. `(and true ...) ⇒ (and ...)`
3. `(and) ⇒ true`

Substitution rules for `or`

1. `(or true ...) ⇒ true`
2. `(or false ...) ⇒ (or ...)`
3. `(or) ⇒ false`

Errors

Syntax error: A sentence cannot be interpreted using the grammar of the language

```
(10 + 1)
```

Runtime error: An expression cannot be reduced to a value by application our semantic rules.

```
(cond [(> 3 4) x]) ⇒
(cond [false x]) ⇒
(cond ) ⇒
cond: all question results were false
```

Lists

Lists are used when a collection of values need to be stored (ex. a shopping list). The order of the items may or may not matter.

In racket, lists are structured recursively, meaning that a list is defined as a list item plus a smaller list. The "base case" of the list is the empty list, which is a list of 0 items. In racket, this is represented by `empty`.

Lists in racket are defined like this:

```
(define wish-list
    (cons "comics"
        (cons "turtle figure"
            (cons "donkey kong"
                (cons "play-doh set"
                    empty)))))
```

An empty list can also be defined:

```
(define myList empty)
```

List Functions

`cons`: consumes a value and a list, and produces the list with that value appended

```
(cons "after effects" wish-list)
;; list containing "after effects", "turtle figure", etc.
```

`length`: returns the length of the list

```
(length wish-list)     ; 4
```

`first`: returns the first item in the list

```
(first wish-list)      ; "comics"
```

`rest`: returns the list without the first item

```
(rest wish-list)      ; (cons "turtle figure" (cons "donkey kong" ...))
```

`empty?` returns true if the list is empty

```
(empty? wish-list)    ; false
```

`cons?` returns true if the item is another cons statement (a list that contains at least one value)

```
(cons? wish-list)     ; true
```

`list?` returns true if the item is a list (including an empty one)

```
(list? wish-list)     ; true
```

`member?` returns true if an item is inside a list, and false otherwise

```
(member? "turtle figure" wish-list)    ; true
```

`reverse` returns the list in reverse order

```
(reverse wish-list)   ; "playdough set", "donkey kong", ...
```

`reverse` is not to be used on assignments, but can be useful to correct recursion errors

Navigating Lists

Individual items stored in lists can be accessed by a combination of `rest` and `first`.

```
(define clst
    (cons "U2"
        (cons "DaCapo"
            (cons "Waterboys"
```

```
                empty))))

(first clst)                ; "U2"
(first (rest clst))         ; "DaCapo"
(first (rest (rest clst))   ; "Waterboys"
```

Lists in Contracts

When a function accepts or returns a list, it must be denoted in the contract. We will use `(listof Type)` to denote our lists.

```
;; myFunction: (listof Str) -> (listof Nat)
```

Syntax and Semantics

Values

List values can either be `empty` or `(cons v l)`, where `v` is a value and `l` is another list. Representing the value of a list as a nested series of `cond` statements (instead of something like `[1, 2, 3]`) is called **constructor notation**. `v` must be an actual value, not an expression that evaluates to a value.

Substitution Rules

```
(first (cons a b)) ⇒ a, where a and b are values.
```

```
(rest (cons a b)) ⇒ b, where a and b are values.
```

```
(empty? empty) ⇒ true.
```

```
(empty? a) ⇒ false, where a is any Racket value other than empty.
```

```
(cons? (cons a b)) ⇒ true, where a and b are values.
```

```
(cons? a) ⇒ false, where a is any Racket value not created using cons.
```

Data definitions and templates

`cons` can be used to build more than just lists. However, we might build functions expecting the `cons` statements to be organized a certain way. As such, we can specify **data definitions** for different structures of `cons` statements.

Data definition of a List

```
;; A (listof X) is one of:
;; * empty                    ; base definition
;; * (cons X (listof X))      ; recursive definition
```

With this definition, we can show rigorously that any list really is a list.

List processing template

Functions often reflect the structure of their parameters, so we can develop **function templates** to help us write functions for different data definitions, including lists.

Because our recursively defined list has two cases, functions that process lists must also have two matching cases: one for when the list is empty and one for when it isn't.

```
;; listof-X-template: (listof X) → Any
(define (listof-X-template lox)
    (cond [(empty? lox) ...]                    ;; base case
          [(cons? lox) (... (first lox)         ;; something with current item
          (listof-X-template (rest lox)))]))    ;; apply function to rest of
list
```

If the X in `(listof X)` needs processing, a helper function should be written and applied inside of the conditional statement.

If a list needs to output a list, the recursive case should evaluate a `cons` expression.

Examples

Example: `count-items` returns the number of items in the list by adding 1 to the length of the rest of the list.

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

Example: `sum-items` returns the sum of the numbers in the list by adding the first number in the list to the sum of the rest of the numbers in the list.

```
;; sum-items: (listof Num) -> Num
(define (sum-items loc)
    (cond [(empty? loc) 0]
          [else (+ (first loc) (sum-items (rest loc)))]))
```

Example: `negate-list` returns a list with each number replaced by its negative.

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

Recursion

A function is recursive when the body of the function includes a call to that same function. Recursive functions must have at least two cases: one that is equal to a value and terminates immediately (base case) and a call to to the function with some change to the parameter that makes the problem simpler (recursive case).

Simple recursion is when there is only one recursive case.

For lists, this means that a recursive function that processes lists must either terminate with a value or pass a shorter version of the same list into itself.

Recursive functions are similar to the principle of mathematical induction.

Non-empty lists

Sometimes functions only make sense for lists that contain items (ex. `(max ne-list)`). For these cases, we have a new datatype: the non-empty list.

```
;; A (ne-listof X) is:
;; * (cons X (listof X))    ; recursive definition
```

Design recipe refinements

If a new datatype is being used (not including `list` and `ne-list`), its template and data definition must be added to the design recipe someplace before its first instance. For the template, remember that any self-referential data type must have a base case and at least one recursive case.

Strings and lists of characters

In racket, strings are really just lists of characters, even if they have different literals.

Helpful string functions

`string->list` turns a string into a list of characters
`list->string` turns a list of characters into a string

Wrapper functions

Sometimes, data needs to be processed before it can be used in a list (or any other) function. For example, a function might require a list of characters, so anyone wanting to use it on a string have to convert the string themselves.

A wrapper function makes your function easier to use by performing those housekeeping tasks. Wrapper functions always call their "child" function after setting up the proper conditions for it run.

Natural Numbers, Recursively

Formal definition of natural numbers

Logicians use the Peano axioms to formally define the natural numbers. These include

- 0 is a natural number
- For every natural number n, S(0) also a natural number (where S(x) is the successor function that returns the next natural number)

Data definition for natural numbers

This can be translated into racket as a data definition:

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

Racket also has the inverse successor function `(sub1 n)` that moves towards the base case

Because this recursive defenition of natural numbers is similar to our recursive defition of a list, we can write a similar template for it:

```
;; nat-template: Nat -> Any
(define (nat-template n)
(cond [(zero? n) ...]
      [else (... n ... ... (nat-template (sub1 n)) ...)]))
```

Iteration

Racket (and all functional programming languages) don't use loops :(. Instead, they use recursion to iterate through lists.

Countdowns

Countdown: takes the natural numbers `n` and `k` as input and returns a list comprised of `n`, `n-1`, `n-2` ... `k+2`, `k+1`, `k`.

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

If we want our countdown to stop at 0, we should use `(zero? n)` instead of `(= n k)` and remove `k` as a parameter entirely.

Notice that the value of `n` never changes: this is always true of recursive parameters. We say that `n` "goes along for the ride".

Our countdown function is the functional equivalent to something like

```
//define variables, etc
void countdown(int n, int k) {
    for(int i = n; i >= k; i--) {
        myList.add(i);
    }
}
```

Countups

Countup does the opposite of a countdown: it takes the natural numbers `n` and `k` as input and returns a list comprised of `n`, `n+1`, `n+2` ... `k-2`, `k-1`, `k`.

```
;; countup: Nat, Nat -> (listof Nat)
(define (countup n k)
    (cond [(= n k) (cons k empty)]
          [else (cons n (countup-to (add1 n) k))]))
```

If we want our countup to stop at 0, we should use `(zero? n)` instead of `(= n k)` and remove `k` as a parameter entirely.

More Lists

Sorting lists

Helper functions, sometimes recursive ones, are sometimes necessary for list processing. A common example is a helper function that sorts a list.

An easy and approachable way to sort lists recursively is **insertion sort**:

1. Sort the rest of the list
2. Insert the first element in the correct position in the list (using a helper function called `insert`)

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

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

List abbreviations

List abbreviations are available in the language level "Beginning student with list abbreviations".

Defining lists with `list`

```
(cons 1 (cons 2 (cons 3 (cons 4 ... empty))))
```

to be expressed as:

```
(list 1 2 3 4 ...)        ;; no empty
```

Lists defined this way have slightly different use cases to those defined using `cons`: lists constructed with `list` are of fixed size, whereas lists with `cons` are arbitrarily sized. Because of this, lists are preferred for writing test cases.

All of the regular list functions work with `list`. Lists can be "added to" (creates a new list) by using `cons`.

`list` and `cons` should be distinguished between when writing data definitions.

Returning elements

The second element of a list can be accessed using:

```
(second my-list)
```

which is a shortcut for

```
(first (rest my-list))
```

Functions that return specific elements are defined up to `eigth`.

Lists containing lists

Lists can contain any type of element, even another list. This is much more achievable with the `list` notation:

```
;; [[1, 2, 3], [4, 5], 6]
(list (list 1 2 3) (list 4 5) 6)
```

We can also mix-and-match `cons` and `list` to create nested lists.

A list that only contains values (not lists) is called a **flat list**.

Dictionaries

A dictionary contains a number of **keys**, with each one having an associated **value**.

> *Ex*. In an actual dictionary, the keys are words and the values are definitions
> *Ex*. In a stock, the keys are symbols and the values are prices

Many tables with two columns can be expressed as dictionaries.

Dictionary operations

Lookup: Given a key, return the corresponding value
Add: Given a `(key, value)` pair, add it to the dictionary
Remove: Given a key, remove its corresponding `(key, value)` entry

Implementing dictionaries

A simple solution to implementing a dictionary is using an **association list**: a list of lists that contain two values each: the key and the value.

```
;; An association list (AL) is one of:
;; * empty
;; * (cons (list Nat Str) AL)
;;     Requires: each key (Nat) is unique
;; (for an AL with Nat keys)
```

The association list's type is `(listof (listof X Y))`, with the template

```
;; al-template: AL -> Any
(define (al-template alst)
    (cond [(empty? alst) ...]
          [else (... (first (first alst)) ... ; first key
                (second (first alst)) ... ; first value
                (al-template (rest alst)))]))
```

Lookup can be implemented as:

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

Add and remove were left as exercises fill in

Two-dimensional data

Another use of lists is to represent a two-dimensional table. A `r` x `c` is stored in a list with `r` entries, with each one of those having a length of `c`.

When writing functions that produce tables/lists like this, it is often helpful to have a helper function that produces each row, and then a main function that appends the rows together.

Processing two lists at the same time

There are three different cases for functions that consume two lists.

In the simplest case, only simple recursion is performed. An example of a function that processes two lists using only simple recursion is the already-defined `append` function: it only needs to process the second list recursively.

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

In the next case, **processing in lockstep**, two lists of the same length are processed at the same rate. Because `(empty? list1)` if and only if `(empty? list1)`, there are only two combinations that are valid for the possible data. Thus, the lockstep template is

```
(define (lockstep-template lst1 lst2)
    (cond [(empty? lst1) ... ]
          [else (... (first lst1) ... (first lst2) ...
                (lockstep-template (rest lst1) (rest lst2)) ... )]))
```

For example, a formula calculating the dot product of a set of vectors [for vectors $(a_1, a_2, \dots)$ and $(b_1, b_2, \dots)$, $(a_1b_1 + a_2b_2, \dots)$] would process both vectors at the same time and add to the dot product as it goes.

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

The third case is the most complex: different lists being processed at different rates. In this situation, all cases of empty/non-empty have to be checked, leading to four cases in

the template: both lists are empty, one list is empty (x2) and both lists are empty. This leads to the template

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

where expression I doesn't require recursion, expressions II and III may or may not require recursion, and expression IV must have recursion in some form.

An example is a function that determines whether two lists are equal:

```
(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)))]))
```

Patterns of recursion

Simple recursion

In simple recursion, every argument in the recursive function is either unchanged or a recursive call with a simpler input. There can be multiple conditions with a recursive call, as long as there is only one recursive call per condition.

Exponential blowup

Imagine we defined a `(max-list-v2 list)` function like so:

```
(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))]))
```

This function uses `(> )` instead of `(max )` in the recursive call because `(> )` is a less expensive function than `(max )` (this strategy is called in-lining). However, this implementation is much slower than the one that uses `(max )` because `(max-list-v2 )` may be called twice every recursive call (once in the comparison with `<` and once in the `[else]` statement). As such, the number of aclls increases exponentially with each layer of recursion.

Measuring efficiency: big O notation

`max-list-v2` may make up to $2^n - 1$ recursive calls if the list it is called on has n elements. As such, we say `max-list-v2`'s efficiency is proportional to $O(2^n)$.

In contrast, the regular version of `max-list` (as well as other common functions like `length`) may make up to $f(n)$ recursive calls on a list with n elements. As such, we say that it has an efficiency of $O(n)$.

Any function can be classed by efficiency in this way:

"Big-O"	Explanation	Example
$O(1)$	No recursive calls	`add`, `first`

"Big-O"	Explanation	Example
$O(log_2 n)$	List divided in half, work done on half	`binary-search` on balanced tree
$O(n)$	One recursive application per list element	`length`, `max`, etc
$O(nlog_2 n)$	List divided on half, work done on both halves	`merge-sort`
$O(n^2)$	$O(n)$ application on each list item	`insertion-sort`
$O(2^n)$	Two recursive applications per list item	`max-list-v2`
$O(k^n)$	`k` recursive applications per list item	don't even wanna know

The last two (and mostly 3) rows are considered inefficient, and should be avoided (especially the last 2).

Accumulative recursion

When humans find the maximum of the list, we don't usually follow a recursive algorithm: we move through the list iteratively and compare the current value with the largest one we've seen so far.

We can apply this strategy recursively by using an **accumulator**, which is a parameter in a recursive function that keeps track of a value. In this case, the accumulator parameter would be equal to the largest value so far, which we could compare to the rest of the list. If the current max is smaller than the current value being looked at, it is replaced in the recursive call.

```
;; accumulative max function
(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)]))

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

Wrapper functions are almost always used for accumulative recursion because the function we're trying to implement doesn't have an accumulative parameter.

A recursive function is accumulative if all of the recursive function's arguments are either:

1. Unchanged
2. One step closer to the base case
3. A partial answer (accumulator)

The value of the accumulator is always used an at least one of the base cases.

Another example of accumulative recursion is list reversal. In the simple recursion case, the recursive function must pass over the whole list to add an item at the end, which leads to an efficiency of $O(n^2)$. In contrast, the accumulative recursion cases stores the current reversed list as an accumulator and `cons`es the first element in the list with the accumulated reversed list, leading to an efficiency of $O(n)$:

```
;; my-reverse: (listof X) → (listof X)
(define (my-reverse lst) ; wrapper function
    (my-rev/acc lst empty))

(define (my-rev/acc lst acc) ; helper function
    (cond [(empty? lst) acc]
          [else (my-rev/acc (rest lst) (cons (first lst) acc))]))
```

Accumulative recursion is harder to conceptualize and debug than simple recursion, so the latter is preferred if possible.

Generative recursion

In generative recursion, the value passed into the recursive function call is a result of a computation on the previous recursive parameter: a new parameter is *generated*. As such, the parameter may move closer to the base case, but it may also become more complex (unlike simple recursion): for this reason, generative recursion is much harder to debug and conceptualize.

An example of a function that uses generative recursion is `(euclid-gcd n m)` function, which calculates the greatest common denominator shared by the numbers `n` and `m` using the Euclidian method.

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

Generative recursion is powerful, but we will avoid it for now.

Structures

Compound Data

We have already seen compound data in this course in the form of the dictionary (implemented as an association list). Some other examples include

- A complex number $a + bi$
- A binary tree
- A student ID

We could represent this last one as a list containing the label for each attribute, as well as a list with all of the values for those attributes in order

```
(list "James Bond" "Math" (list "CS135" "MATH135" "MATH137"))
```

However, for this to work, we need to remember the order of the attributes. Racket has a better way to store compound data like this.

Structures

Posn

This is a built-in structure (short for "position"). It stores two fields: an x-coordinate denoted `x` and a y-coordinate denoted `y`. A `posn` structure can be created using the constructor function `make-posn`:

```
(make-posn 38 12) ;; -> (make-posn 38 12)
;; (make-posn ... ...) is the literal for posn, much like "" for a string
```

We have selector functions that allow us to access each value from our `posn` structure:

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

Like any other type, `posn` has a type predicate:

```
(posn? value) ;; if value is a posn -> true, else false
```

It also has a template:

```
;; my-posn-template: Posn -> Any
(define (my-posn-template p)
    (...(posn-x p)... ...(posn-y p)...))
```

Self-defined structures

`posn` is an example of a structure built into racket, but we can define our own as well using the special form `define-struct`:

```
(define-struct struc-name field1 field2 ...)
```

Doing this also defines at least three new functions:

1. The constructor, `struct-name`, that lets us create values of this type
2. The type predicate, `(struct-name? value)`, that lets us test for this type
3. Selectors, `(struct-name-field1 value)`, `(struct-name-field2 value)`, etc. that let us access the values of individual fields of the structure

Now, we can create a new value of this type and store it in a constant (or pass it in a function).

We must also write a data definition for the structure that describes the type and requirements of the fields:

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

Self-defined structures follow the following substitution rules:

```
(sname-fname_i (make-sname v_1 ... v_i ... v_n)) -> v_i.
```

```
(sname? (make-sname v_1 ... v_n)) -> true
```

```
(sname? V) -> false for V a value of any other type
```

The template for a user-defined structure is simply all of the fields:

```
(define ... (struct-name-field1) ...
            (struct-name-field2) ...
            (struct-name-field3) ...
            ...)
```

Checked functions

Constructors don't check that their input is the correct type, which can lead to errors. To avoid this, a wrapper function can be written that checks whether the input is valid before making a new structure.

Simulating structures

If we want to, we can "implement" structures ourselves using lists:

1. `(make-struct values...)` creates a list with `values...`
2. `(struct? value)` checks the types (and requirements) of the items in the list
3. Each field function returns a list item from the corresponding index

Why you would want to do this is beyond me; I can only assume this is in the notes because some versions racket don't implement structures.

Mixed data

Functions may consumed mixed data: data of multiple (likely related) types. A particular type of mixed data typically has a data definition.

```
;; type: graduate student
;; type: undergraduate student

;; A student is one of
;; * graduate student
;; * undergraduate student
```

A template for a mixed data type will determine the type of data being inputted (usually with a `cond` block with type predicates), and then apply the corresponding template.

Advantages of structures

- Helps you avoid programming errors (ex. finding the item at the wrong field/list item)
- Help code be more understandable by associating meaningful names with fields
- Automatically generates useful functions (ex. `make-struct`)

Advantages of lists

- Allows you to write functions that work with multiple datatypes
- Can be expressed more compactly than structures

Quote notation

`cons` is a way to define lists that highlights its underlying recursive structure at the expense of usability. `list` is more wieldly, but we can do better: quote notation. Quote notation works by defining a list of `a`, `b`, `c`... as

```
'(a, b, c)
```

`'X` is an abbreviation of `(quote X)`. Here are some more examples:

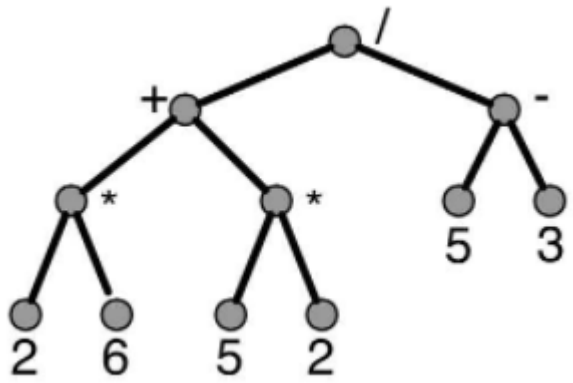
```
'1 -> 1, '"ABC" -> "ABC", 'earth -> 'earth
'(1 2 3) -> (list 1 2 3)
'(a b c) -> (list 'a 'b 'c)
'(1 ("abc" earth) 2) -> (list 1 (list "abc" 'earth) 2)
'(1 (+ 2 3)) -> (list 1 (list '+ 2 3))
'() -> empty
```

Binary Trees

Trees

Storing information in **trees** turns out to be very useful in computing science, both because data often occurs in a tree-like structure and because these structures can be manipulated efficiently.

For example, the expression $\frac{(2*6)+(5*2)}{5-3}$ can be represented as:



A tree is a set of **nodes** and **edges** where an edge connects two distinct nodes. A tree has three requirements:

- One node is the **root** (the starting node)
- Every node that isn't the root is connected to some **parent node** that is closer to (or is) the root. The node is the parent node's **child node**.
- Every node that isn't the root is the child of n, or the child of a child of n, or the child of a child of a child of n, etc. (the tree is connected)

Leaves are nodes without children, while **internal nodes** are nodes that have children (every node must be one of these). The **ancestors** of a node are the list of parent nodes that lead back to the root, while the **descendants** of a node are the list of nodes that has the node as an ancestor. A **subtree rooted at n** is the tree formed of n and its descendants.

Labels can be added to nodes to assign data to them. It is these that make trees computationally useful.

Binary Trees

Binary trees are trees where every parent node has at most two children, and the order of the children doesn't matter.

Data definition:

```
;; A Node is a (make-node Nat BT BT)     ; Nat could be any type
(define-struct node (key left right))    ; key is the label

;; A binary tree (BT) is one of:
;; * empty (nothing to do with lists, just fits requirement well)
;; * Node
```

Binary Tree template

From the data definition, we know a binary tree is a list, a type of compound data (a structure), and a type of mixed data. So, we know we need to use recursion, use each field somehow, and use `cond` to distinguish between both types of data (more accurately, to determine if the tree is empty).

So, our template is:

```
;; bt-template: BT -> Any
(define (bt-template btree)
    (cond [(empty? btree) ...]
          [(node? btree) (... (node-key btree))
                         ... (bt-template (node-left btree))
                         ... (bt-template (node-right btree)))]))
```

Searching binary trees

Searching binary trees for a key has a very simple strategy (it's actually depth-first search, but we don't know that yet):

1. If the root's key is the one we are searching for, return true (or some other info)
2. Otherwise
    1. Search the subtree of the left child
    2. Search the subtree of the right child

Here it is implemented in racket, using the binary tree template

```
;; (search k tree) produces true if k is in tree; false otherwise.
;; 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)))]))
```

It is pretty easy to modify this algorithm so it returns the path that it traveled to find the key.

```
;; 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]))
```

Binary Search Trees

The **binary search tree** is like the binary tree with one additional property (the **ordering property**): The key of a node is larger than every key in the left subtree and smaller than every key in the right subtree (this applies to all internal nodes).

This makes searching is *dramatically* more efficient because we immediately know which subtree a key must be in, and don't need to bother searching the other one. This search algorithm is called **binary search**, and can be performed on ordered lists as well.

Here is an implementation of a binary search on a binary tree in racket:

```
;; (search-bst n t) produces true if n is in t; false otherwise.
;; 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))]))
```

Turning a list into a binary search tree

Since binary trees have structured orders, there is only one place a new key can possibly added into any given tree. We can use binary search to find that location (by searching for the key) and then insert it as appropriate.

With this set of steps, we can write a function that turns a list into a BST by starting with an empty tree and adding each item one by one. Since lists are in the form `cons(k list)`, we add `k` to the BST formed from `list`. Thus, root of the BST is the last item in `list`.

Augmenting Trees and BST Dictionaries

Nodes often (functionally, always) have keys, but we can more values to the node if we wish by adding more fields to the `node` structure. The values can be of any type or in any amount; the only thing that must stay the same is that the keys are unique.

This can be used to implement **BST Dictionaries**, which are much more efficient than association list dictionaries because of the binary search algorithm.

Here is an implementation of search for a BST dictionary in racket:

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

Example: evolutionary tree

Evolutionary trees are created by biologists to illustrate and hypothesize the evolutionary relationships between different species. In an evolutionary tree:

- The leaves are species that currently exist (or existed)
- The internal nodes are hypothesized common ancestors to the leaf notes

- The root node is a single common ancestor
- Each common ancestor "split" into two species at some point, so each internal node has two children (order doesn't matter)
- Each node may be augmented with information about the species (i.e. if it endangered [boolean], how long ago it split [integer], etc)
- The order of the children does not matter

Binary Expression Trees

As illustrated before, any mathematical expression that uses operators that operate on two values ($+$, $-$, $*$, $/$, x^y) can be expressed as a binary tree (although not usually a search tree).

In a binary expression tree:

- Internal nodes have two children (we can express many unary operators as binary operators)
- Leaves have number labels, internal nodes have operator labels
- The order of the children matters and is dictated by the expression

Mutual Recursion

Mutual Recursion occurs when two functions call each other recursively: `f` calls `g` and `g` calls `f`. Functions like this can often be combined into one function, but it is often easier to understand the code if the function is written in two parts.

Mutual Recursion with natural numbers: `is-even?`

We can write two functions `is-even?` and `is-odd?`, both of which take a natural number `n` as input. If `n` is 0, `is-even?` evaluates to true and `is-odd?` evaluates to false; otherwise, it checks if the number below is of the opposite party (the mutual recursion). As such, the two functions are called in an alternating chain until 0 is passed into one of the functions.

```
;; (is-even? n) produces true if n is even and false otherwise

;; Examples:
(check-expect (is-even? 4) true)
(check-expect (is-even? 5) false)

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

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

Of course, these functions can be combined by adding a `cond` block that checks if `n` is 1, and then returns false. However, this can't always be done.

Mutual Recursion on a list: `keep-alternates`

`keep-alternates` keeps every other item in the list. A strategy might be to process the list in groups of two, but this means there must be multiple base cases. Mutual recursion can give us a "better" answer:

```
;; keep-alternates: (listof Any) -> (listof Any)
(define (keep-alternates lst)
    (cond [(empty? lst) empty]
          [else (cons (first lst) (skip-alternates (rest lst)))]))

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

Mutual recursion on a tree: `binexp` and `binode`

When a data type references a second datatype that references the first one, mutual recursion is natural. An example of this relationship is the one between the `binexp` and `binode` templates:

```
;; 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))))
```

General Trees

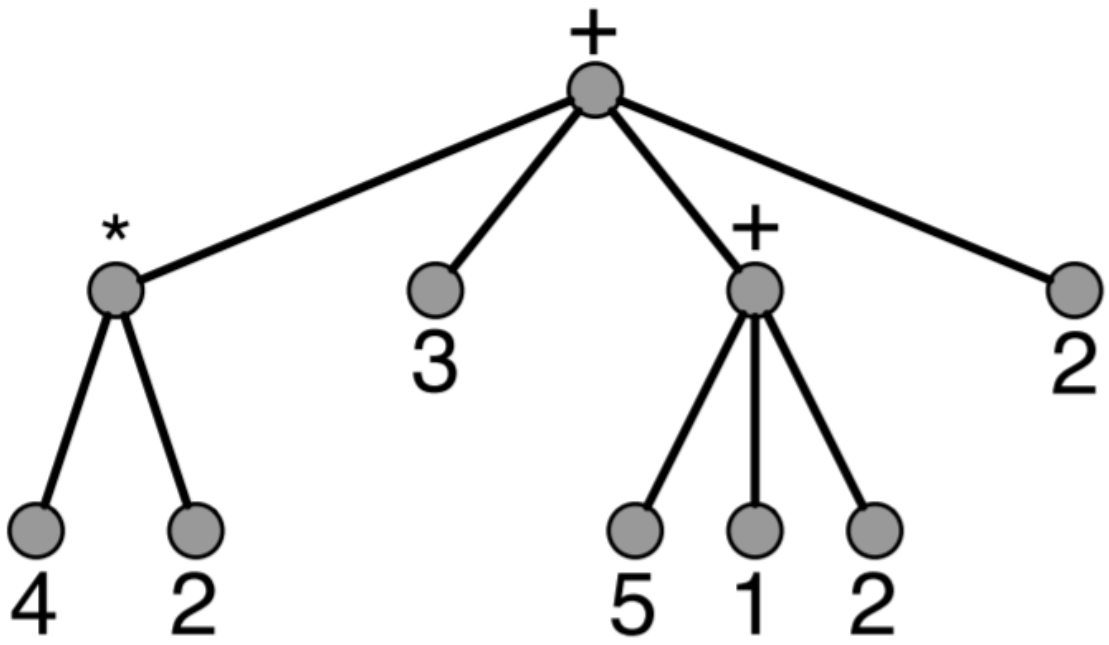
Binary trees are useful in many areas, but their use is limited by the fact that any given node can only have two children or less.

General trees are trees where nodes can have any number of children.

Arithmetic Expressions

As we've seen before, arithmetic operators like `(+ ...)` and `(* ...)` in racket can have any number of arguments. So, while they can't necessairly be represented by binary trees, they can be represented by general trees.

For example, `(+ (* 4 2) 3 (+ 5 1 2) 2)` can be represented as:



For binary arithmetic expressions, we defined each node as having three fields: the `value` (be it a number or an operator), the `right` subtree and the `left` subtree. However, general arithmetic expressions are defined with two fields: `value` (again, either a number or an operator) and `expressions`, a *list* of subtrees.

With this example in mind, we can create a data definition, structure, and template for a general arithmetic tree:

```
;; 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)).
```

This leads of course to mutual recursion.

Developing `Eval`

Knowing what we know about (arithmetic) general trees, we can build a function that evaluates math expressions.

```
;; (eval exp) evaluates the arithmetic expression exp
```

This can be done using a helper function (`apply`) that converts the operators in the tree to real operators.

```
;; 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)))]))
```

Alternate Data Definition

If we want to, we can use a list to create our tree instead of an `OpNode` structure:

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

(sidenote) I don't know why you would prefer to do this, but you can I guess

Mutual Recursion

Having complicated data definitions, especially when it comes to trees, often leads to mutual recursion.

Other uses of General Trees

General trees like this are very good at dealing with hierarchical data. Arithmetic expressions are an extremely common example of this in computing, but plenty of other data is hierarchical as well.

Another example of hierarchical data is a functional programming language (we will use racket as an example). Because racket only consists of function definitions and calls, any racket program can be represented as a tree in the same way that racket arithmetic can. As such, we can use a general tree to build a racket interpreter.

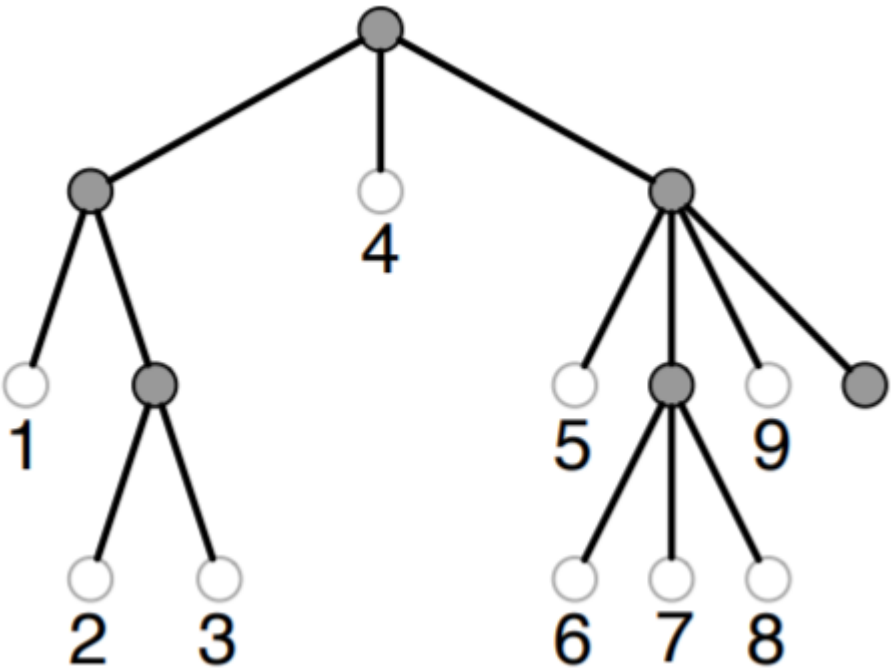
Some other common (and useful) examples of hierarchical data are:

- Organized text (markdown, HTML, etc)
- Webpages (DOM)

Nested Lists

We have discussed normal lists and lists of lists, but this pattern can go arbitrarily deep: any list with a list inside of it can be called a **nested list**.

(Unsurprisingly) Nested lists can be represented as (or really, are) general trees. For example, this is a tree representation of `'((1 (2 3)) 4 (5 (6 7 8) 9 ()))`:



Data Definitions for Nested Lists

A list item might be empty, or a list, or a nested list (which represents any time of list of list of list of...). So, we have:

```
;; A nested list of numbers (Nest-List-Num) is one of:
;; * empty
;; * (cons Nest-List-Num Nest-List-Num)
;; * (cons Num Nest-List-Num)
```

We can also make a template for nested lists that calls itself on every list in the list (meaning all the nested ones):

```
;; nest-lst-template: (nested-listof X) -> Any
(define (nest-lst-template lst)
    (cond [(empty? lst) ...]
          [(list? (first lst))
                  (... (nest-lst-template (first lst))
                   ... (nest-lst-template (rest lst)) ...)]
          [else (... (first lst) ... (nest-lst-template (rest lst)) ...)]))
```

`count-items`

With this template, we can create a function that counts how many items are stored in a nested list (lists don't count as items):

```
;; (count-items nl) counts the number of items in nl.
;; count-items: (nested-listof X) -> Nat
    (define (count-items lst)
        (cond [(empty? lst) 0]
              [(list? (first lst)) (+ (count-items (first lst))
                                      (count-items (rest lst)))]
              [else (+ 1 (count-items (rest lst)))]))
```

Flattening a Nested List

We can also use the template to create a function, `flatten`, that turns a nested listed into a flat list. This is how the racket function `append` works.

```
;; (flatten lst) produces a flat list with all the elements of lst.
;; flatten: (nested-listof X) -> (listof X)
(define (flatten lst)
    (cond [(empty? lst) empty]
          [(list? (first lst)) (append (flatten (first lst))
                                       (flatten (rest lst)))]
          [else (cons (first lst)
                      (flatten (rest lst)))]))
```

Local Definitions and Lexical Scope

For the last modules, we couldn't nest `define` inside of other `define` statements. As such, every constant was accessible everywhere in the program. However, this can be a bit cluttered an inefficient, so racket introduced a solution: `local`.

`local` lets us define constants and functions that can only be accessed within the scope of other functions. This is done by defining a list of constants/functions and the body of the function we're using them for inside the body of a `local` statement:

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

Defining constants this way can make code more readable. Consider implementing Heron's function for the area of a triangle:

$$\sqrt{s(s-a)(s-b)(s-c)} \text{ where } s = \frac{a+b+c}{2}$$

In racket, it is easy to implement this function,

```
(define (t-area-v0 a b c)
    (sqrt (* (/ (+ a b c) 2)
          (- (/ (+ a b c) 2) a)
          (- (/ (+ a b c) 2) b)
          (- (/ (+ a b c) 2) c))))
```

but it is very opaque not recognizable as Heron's function. However, we can use a `local` statement to define `s` as `(/ (+ a b c) 2)` within the scope of our function. Defining it this way, we get

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

this makes the function concise and recognizable without the opacity of defining every instance of `s` and the chunkiness of using helper functions.

Semantics and reusing names

A name defined (bound) inside of a `local` expression may reuse a name from a variable that is defined outside of that expression, just like a parameter might. To ensure that everything works as expected, there is a substitution rule.

The substitution rule works taking the variable in local and assigning a new name (fresh identifier) that hasn't been used anywhere in the program. This is usually done by adding "_1" or "_2", etc to the end of the variable name. This new constant is then "taken" out of the local function and defined inside the enclosing scope. Then, the references to this constant inside of the function are replaced with the new name. This way, the function's behavior doesn't change.

Why Use `local`?

Clarity

If a subexpression is used twice within the same function, it will always have the same value. As such, giving it a name using `local` makes the code much more readable.

It is even sometimes helpful to use a local function when the subexpression is only applied once to enhance readability (mnemonic name).

Efficiency: avoiding extra computation

By passing the value of a function as a local constant, that function isn't evaluated every time the value is referenced. Instead, the value is "saved" as a parameter and "goes along for the ride".

An example of this is `max-list`: the max of the rest of the list is saved as a local constant when passed recursively, so `max-list` isn't called in both the predicate and action part of the `cond` expression.

```
;; max-list-v4: (listof Num) -> Num
;; Requires: lon is nonempty
(define (max-list-v4 lon)
    (cond [(empty? (rest lon)) (first lon)]
          [else
            (local [(define max-rest (max-list-v4 (rest lon)))]
                (cond [(> (first lon) max-rest) (first lon)]
                      [else max-rest])))]))
```

The same thing can be done with `search-bt-path`: the searches of the left and right paths are saved as local constants, so they aren't repeatedly recursively applied.

```
;; search-bt-path-v3: Nat BT -> (anyof false (listof Sym))
(define (search-bt-path-v3 k bt)
    (cond [(empty? bt) false]
          [(= k (node-key bt)) '()]
          [else
            (local [(define left-path (search-bt-path-v3 k (node-left bt)))
                    (define right-path (search-bt-path-v3 k (node-right bt)))
              (cond [(list? left-path) (cons 'left left-path)]
                    [(list? right-path) (cons 'right right-path)]
                    [else false]))])))
```

Even here, we search the tree on both sides. Even though this isn't a $O(n^2)$ nightmare, we can still do better using nested `locals`. In version 4, we check if the left tree contains the key first (using a local definition as before). If it does, we `cons` it; otherwise, we locally define the right path `cons` that (unless it isn't a list, then we return `false`)

```
;; search-bt-path-v4: Nat BT -> (anyof false (listof Sym))
(define (search-bt-path-v4 k bt)
    (cond [(empty? bt) false]
          [(= k (node-key bt)) '()]
          [else
            (local [(define left-path (search-bt-path-v4 k (node-left bt)))]
                (cond [(list? left-path) (cons 'left left-path)]
                      [else (local [(define right-path (search-bt-path-v4
                                                        k (node-right bt)))]
                              (cond [(list? right-path) (cons 'right right-
path)]
                                    [else false]))]))])))
```

Encapsulation: Hiding Stuff

Encapsulation refers to hiding parts of the program from the program as a whole. This is something that structures do, although we can access all the information they encapsulate from the outside. However, local definitions actually *hide* information from the enclosing scope; none of the bindings inside affect the enclosing scope.

Behavior encapsulation are when functions are encapsulated/hidden, meaning that they are inaccessible in the enclosing scope. Using this, we can move helper functions *inside* of their main functions, which further compartmentalizes and organizes the main program. This property is particularly useful when using accumulators.

Here is an accumulative implementation of `sum-list` that uses a `local` helper function.

```
(define (sum-list lon)
    (local [(define (sum-list/acc lst sofar)
                (cond [(empty? lst) sofar]
                      [else (sum-list/acc (rest lst) (+ (first lst) sofar))]))]
    (sum-list/acc lon 0)))    ;; would-be body of main function
```

And an implementation of insertion sort where `insert` is a local function:

```
;; purpose, contract, examples, etc.
(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)))]))]
    ;; Would-be main function
    (cond [(empty? lon) empty] [else (insert (first lon) (isort (rest
lon)))])))
;; tests
```

The above example also illustrates the design recipe for encapsulated functions: a purpose and contract should be included with the encapsulated function, but no examples or tests are necessary. The enclosed function is tested by the tests for the enclosing function, which still has the full design recipe.

Finally, we can use `local` statements to implement mutual recursion. In this case, both mutually recursive functions are included in the main function `my-even`:

```
;; 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)))    ;; the would-be main function
```

Scope: Reusing Parameters

Using `local` definitions removes the need for helper functions to have parameters that "go along for the ride" because these parameters can be accessed by name because the enclosed function is inside of the main function.

Substitution Rule

A the substitution of a local expression has three parts, all of which are executed in a single step.

1. `d_i` is the definition of a constant or function (either in the form `(define x_i exp_i)` or `(define (x_i p_1 ... p_m) exp_i)`). The name, `x_i` is replaced with a new name (fresh identifier, call it `x_i_new`) everywhere in the local expression. This applies to every definition of a constant or function (for all `i` > 0) at once
2. All of the changed definitions (`d_1` ... `d_n`) are "lifted out" to the top level of the program all at once, preserving their ordering
3. The remaining expression is in the form `(local [] bodyexp')`, where `bodyexp'` is the rewritten version of bodyexp with all the definition names replaced. This whole local expression is replaced with `bodyexp'`

Terminology for `local` expressions

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

Functions as Values

First class values

First class values are values that can be

1. *consumed* as function arguments
2. *produced* as function results
3. *bound* to identifiers
4. *stored* in lists and structures

In racket, function are first class values: the can be used in all of these ways (in *intermediate student* or after).

This feature is what separates functional programming languages (like Racket) from languages that aren't primarily functional. For example, without using lambda expressions, (1), (2), and (4) can't be done in Java.

Consuming Functions

In racket, function names can be consumed as parameters and then applied:

```
(define (foo f x y) (f x y))
;; ex. (foo + 2 3) ⇒ (+ 2 3) ⇒ 5
```

To see why this is useful, consider two functions: `remove-vowels`, which removes vowels from a list of characters, and `remove-odds`, which removes odd numbers from a list of numbers. These two functions have the same overall structure; the only difference is the predicated used to determine which items should be removed.

As such, we can define a function that abstracts away this difference by accepting the predicate as a argument. This way, we can choose to filter whatever we want from the list by writing the corresponding predicate.

```
(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))]))
```

Filter

The function we've just written already exists in racket as `filter`. `filter` is a **higher-order function** because either consumes or produces a function. Many similar functions that generalize common types of simple recursion exist in racket as well.

Functional Abstraction

What we did when creating `my-filter` is called **functional abstraction**: creating abstract version of concrete types of functions. Doing this has several 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

Functions producing functions

Because functions are values, functions can create other functions as well, which is enormously useful.

An simple example is the function `make-adder`, which produces a function that adds a specific number to any number:

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

Using a trace, we can see that `(make-adder 3)` produces

```
(define (f_1 m) (+ 3 m))
f_1
```

We can store this function as a value, or use it immediately. For example, `((make-adder 3) 4)` evaluates to `7`. We can even bind this function to an identifier and use it over and

over: `(define add_3 (make-adder 3))`.

Being able to do this using `local` means that functions no longer have to be built-in or explicitly defined by the user.

Storing Functions

We can replace our code for `apply` (from evaluating binary expression trees) with code that actually applies the function stored in the `opnode` instead of having to use a `cond` statement to parse it:

```
(define (my-apply op args)
    (cond [(empty? args) (op )]
          [else (op (eval (first args))
                    (my-apply op (rest args)))]))
```

This has the added advantage of working for a binary tree that has no arguments, as no function is applied (leaving the value unchanged).

However, quote notation makes it really easy to store our tree as a list of lists instead of as a structure: `(eval '(+ 1 (* 3 3 3)))` is extremely concise. However, this gives us the issue of having `+` in symbolic form again. To solve this, we can define a lookup table (association list) where symbols are paired up with their corresponding functions:

```
(define trans-table (list (list '+ +) (list '* *)))
```

Then, we can simply apply this function in the main function by implementing a lookup function:

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

;; (eval ex) evaluates the arithmetic expression ex.
;; eval: AExp -> Num
(define (eval ex)
```

```
    (cond [(number? ex) ex]
          [(cons? ex)
           (my-apply (lookup-al (first ex) trans-table) (rest ex))]))

;; (my-apply op exlist) applies op to the list of arguments.
;; my-apply: ??? (listof AExp) -> Num
(define (my-apply op args)
    (cond [(empty? args) (op )]
          [else (op (eval (first args)) (my-apply op (rest args)))]))
```

This way, the apply function doesn't need to use boilerplate code.

Contracts and Types

Before, contracts consisted of in-built datatypes (`Num`, `Bool`, `Sym`, `(listof ...)` etc.) and user-defined datatypes (`HexColor`, `MarkedSCList`, etc). However, since functions (with their own contracts) can be treated as values, we have to nest contracts. When a function is passed as an argument or created, it is replaced by its own contract in the main contract.

For example, the contract of our `lookup-al` function is:

```
;; lookup-al: Sym (listof (list Sym (Num Num → Num))) →
;;            (anyof false (Num Num → Num))
```

Since it takes a `Sym`, an association list between `Sym`s and binary arithmetic functions (whose contracts are `Num Num -> Num`) and either returns `false` or a binary arithmetic function.

Parametric Types

Sometimes, although a higher order function and accept any type of datatype, its the datatypes of its output depend on the functions it accepts. For example, the output of `filter` can produce any type of list, but the predicate provided determines the kind of list that gets outputted (i.e. there is a dependence between the type of predicate and type of list).

To express this, we use a **type variable**, often `X` to denote a hypothetical type that could be anything, but must remain constant (more than one can be used if required). Using a

type variable denotes a relationship between members of the contract. As such, the contract of `filter` is:

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

Functions that can work on many types of values this way are called **polymorphic** or **generic**.

Simulating Structures

We can use the ability to produce and bind functions to implement structures without using lists. Consider a structure that stores a point:

```
(define-struct make-point (x y))
```

This can be simulated with the function

```
(define (mk-point x y)
    (local [(define (symbol-to-value s)
                (cond [(symbol=? s 'x) x]
                      [(symbol=? s 'y) y]))]
    symbol-to-value))
```

This stores a point "structure" as a function, which can be bound to a name using `define` if desired. When points as passed into the "structure", they are passed into the local definition, which will get "lifted out" when the program is evaluated.

As such, if we define a function like `(define p1 (mk-point 3 4))`, it will become `(define p1 symbol_to_value_1)` (because `symbol_to_value` was "lifted out"). `(p1 'x)` evaluates to `(symbol_to_value_1 'x')`, which returns 3 (because we set the "structure" this way). To make it easier to access, we can write some accessor functions:

```
(define (point-x p) (p 'x))
(define (point-y p) (p 'y))
```

So now, `(point-x p1)` evaluates to 3 and `(point-y p1)` evaluates to 4. We have implemented structures (the type predicate can presumably also be implemented).

Functional Abstraction

Abstraction is the process of generalizing sets of similar things. This is done by having functions with parameters (on a value level) and partly by writing function templates (on a functional level). Higher order functions provide a way to fully abstract away small differences in functions by letting functions be parameters.

Anonymous Functions

An anonymous function is a function that has no name (or rather, that doesn't get called by name outside of its local definition).

```
(define (make-adder n)
    (local [(define (f m) (+ n m))] f))
        (make-adder 3)
```

In this example, `f` is only used when creating other functions; it does not need a meaningful name. Because defining it with one adds unnecessary complexity by having useless code, there is a special form for producing anonymous functions to be used as parameters: **lambda expressions**.

Lamda Expressions

A lambda function is a function that makes functions. It is in the form

```
(lambda (x y) (+ x (* x y)))
```

Here, `(x y)` denotes the two parameters of our function, `x` and `y`, and `(+ x (* x y))` is the expression that relates these variables. This code is the unnamed equivalent to:

```
(define (f x y)
    (+ x (* x y)))
```

The lambda version of `make-adder` is:

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

Because lambda functions are values (in fact, they are designed specifically to be used as values), they can be applied in the same line as their definition:

```
((lambda (x y) (+ x (* x y))) 3 6)    ;; -> 21
```

Accordingly, the substitution rule is simply replacing the parameters in the function with the given parameters in the function application, just like a regular function.

```
((lambda (x_1 ... x_n) exp) v_1 ... v_n) -> exp'
```

Lambda and Function Definitions

In the internals of racket using `define` does one thing: binds a value to a name (of course, that value can be a function because racket is a functional programming language). The actual *function* being created is a lambda expression, which is then bound to the specified value.

```
(define (interest-earned amount) (* interest-rate amount))
;; is actually
(define interest-earned (lambda (amount) (* interest-rate amount)))
```

As such, our semantic rule for functions must also include the step that exposes the lambda expression. However, this can make traces much more complicated without telling us anything else about how the program runs, so well will skip this step when tracing functions defined normally (not with a lambda expression).

Transforming Strings

Consider a function `transform` that takes a string and turns it into another based on a set of rules we specify. Before lambda expressions, we would need to hard-code these rules into the `tranform` function, meaning we couldn't apply rules flexibly. Moreover, if some rules overlapped, using a cond statement would only let one be applied at a time.

For example, if had rules to capitalize each letter in the string and switch each letter in the string with the next one alphabetically, "a" could tranform into "A" or "b" (depending

on the order of our rules), but never "B".

Both of these problems can be solved using lambda functions.

We could make a list of question-answer pairs, where the question is a predicate (that accepts a character) and the answer is a Char → Char function. The first question-answer pair in the list would be applied to the string character by character, then the second pair, etc.

Map

`Map` is a function that takes a list and a function, and returns the list with the function having been applied to each element. It can be fairly easily implemented by using a function as a first-class value:

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

`Map` can be used to considerable shorten definitions of functions that traverse through lists and modify each item, something that has come up often in CS 135 (consider `negate-list` and `compute-taxes`). For example, `negate-list` can be implemented as:

```
(define (negate-list lst) (my-map - lst))
```

Foldr

Consider another set of functions: those that process a list and produce a single value. Many of these functions follow a similar structure: A base case for when the function is empty and some operation that links the current list item with a recursive call on the rest of the list. This is, of course, our list template. An example of a function that applies it is `sum-of-numbers`, which adds all the numbers in a list:

```
(define (sum-of-numbers lst)
    (cond [(empty? lst) 0]                          ;; base case (0)
          [else (+ (first lst)                      ;; operation (+)
                   (sum-of-numbers (rest lst)))]))  ;; recursive application
```

This can be generalized by the function `foldr`, which takes a function that combines values (operator), the base case, and the list to apply it to. Here is its implementation

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

And `sum-of-numbers` implemented using `foldr`

```
(define sum-of-numbers (foldr + 0 lst))
```

"Foldr" is short for "fold right", because `foldr` can be thought of as folding a list using the provided combining function.

`Foldr`'s contract is

```
;; foldr: (X X -> X) X (listof X) -> X
```

It is possible to use `folder` in such a way that the value of the first list item doesn't matter. In this case, it doesn't need to be included in the combination function. An example of this is `count-items`: because it only counts items, the value of those items doesn't matter and doesn't need to be included in the function.

Producing lists with `foldr`

Lists can also be produced with folder by including a call to `cons` in the combination function. For example, we can define `negate-list` using `foldr` like so:

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

In this case, the lambda expression `cons`es the negated result to the recursive application.

Implementing `map` with `foldr`

Since `map` is a simply recursive list → list function, we can implement it using `foldr`. Specifically, it applies a function to the first item in the list, then `cons`es it with the

function mapped to the rest of the list.

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

Using `foldr`

Although `foldr` effectively replaces the list template, it is not obsolete. Writing a function explicitly using the list template has the advantage of being readable and human-understandable, unlike the much more opaque `foldr`. `foldr` condenses code, but at the expense of readability.

`foldl`: Generalizing Accumulative Recursion

Just as `foldr` starts from the right of a list and moves left, `foldl` starts from the left of a list and moves right using accumulative recursion. The first item of the list and the accumulated value are combined using the combine function, then passed as the accumulator into the recursive call.

`foldl` can be implemented like so:

```
(define (my-foldl combine base lst0)
    (local
        [(define (foldl/acc lst acc)
            (cond [(empty? lst) acc]
                  [else (foldl/acc (rest lst) (combine (first lst) acc))]))]
        (foldl/acc lst0 base)))
```

Here is `sum-list` and `my-reverse` implemented by using `foldl`. Notice that `sum-list` has the same parameters as its `foldr` implementation; this shows how it can be easily implemented using both simple and accumulative recursion.

```
(define (sum-list lon) (my-foldl + 0 lon))
```

```
(define (my-reverse lst) (my-foldl cons empty lst))
```

`build-list`

`build-list` is another useful higher-order function that takes a natural number `n` and a function `f`, and produces the list

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

We can easily implement it:

```
(define (my-build-list n f)
    (local
        [(define (list-from i)
            (cond [(>= i n) empty]
                  [else (cons (f i) (list-from (add1 i)))]))]
        (list-from 0)))
```

Generative Recursion

Simple and accumulative recursion, which we have been using so far, is a way of deriving code whose form parallels a data definition. Generative recursion is more general: the recursive cases are generated based on the problem to be solved. This means that the solutions (and cases) for these problems require some background knowledge of the problem attempting to be solved.

An example is the calculation of $gcd(a, b)$ using the Euclidian algorithm:

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

Coming up with this solution requires outside knowledge, namely that $gcd(a, b) = gcd(b, r)$.

Termination

A function **terminates** when it can be reduced to a value in finite time. It is easy to argue that a non-recursive function terminates because it can be shown that all functions called inside of it terminate. However, showing that all recursive functions terminate is harder.

Termination of simple recursion

Simply recursive functions always make recursive applications on smaller (or less complex) values, which are bounded on the bottom by a base case (for example, 0 or a `empty`). As a result, we can show that the **depth of recursion** is bounded below, and thus the function evaluation can't go on forever.

Many list functions have a maximal depth of recursion equal to the length of the list.

Termination of `euclid-gcd`

Although there is no way to prove that generatively recursive functions terminate in the general case, we can use outside knowledge to show that specific generatively recursive functions do.

In the case of `euclid-gcd`, we can show that the second argument always gets smaller with each application, and is bounded below by the base case of 0. As such, its depth of recursion is bounded by the size of the second argument, even though it gets reduced much faster in practice.

An example of a function we can't show to terminate is the collatz function (namesake of the collatz conjecture).

Quicksort

Quicksort is a recursive, divide-and-conquer sorting algorithm that follows the following steps when sorting a list of numbers:

1. Set the first number in the list as the "pivot"
2. Create three lists: the one with all the elements in the list smaller than or equal to the pivot, all the greater than or equal to elements, and a list containing just the pivot
3. Quicksort the first two lists
4. Append into one list in the order {less than} {pivot} {greater than}

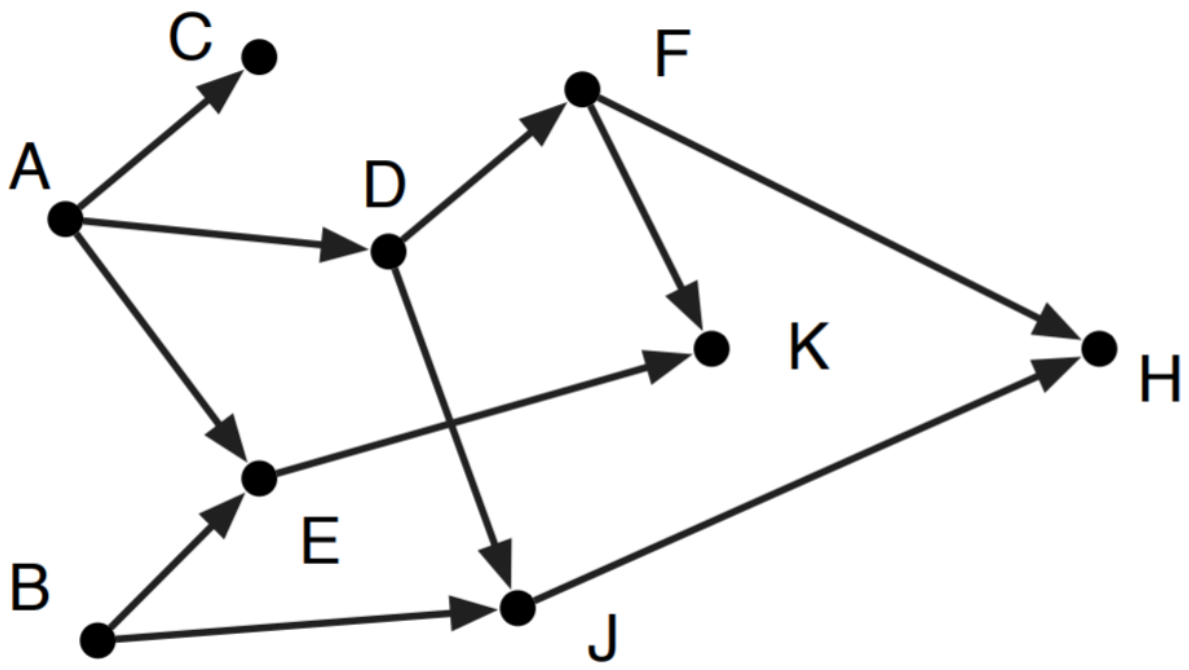
Quicksort usually has an efficiency of $O(n * log(n))$. However, if a list is already in normal or reverse order, quicksort will have an efficiency approaching $O(n^2)$.

Quicksort termination

Since the total number of items in the sublists is always smaller than the number of items in the main list (because the pivot is not included), we know quicksort must terminate. Its depth of recursion is bounded above by the number of items in the list.

Directed Graphs

A **directed graph** consists of a collection of **nodes** (also called **vertices**) connected by **edges**. An edge is an ordered pair of nodes, which we can represent by an arrow from one node to another.



It turns out that trees are a type of graph; specifically when an edge denotes a parent-child relationship. Graphs are the generalization of trees: any two nodes can be connected.

An edge between nodes A and B is denoted (A, B), which is graphically denoted as $A \to B$. Here, B is an **out-neighbor** of A, and A is an **in-neighbor** of B.

A sequence of nodes A, B, C, D... is a **path** or **route** if (A, B) and (B, C) and (C, D), etc. are all edges. In this case, the path has a length equal to the number of edges. If a path comes back upon the starting element, it is called a **cycle**; directed graphs without cycles are called **directed acyclic graphs** (**DAG**s).

Representing Graphs

We can represent a graph by associating each node with a symbol, and storing a list of each node's out-neighbors within it. This representation is called an **adjacency list**, and it is quite similar to how we stored trees.

Data definitions for Node and Graph

```
;; 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 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)) ...)]))
```

This leaves space for `list-of-nodes-template`, which processes lists of nodes (likely using mutual recursion).

Finding neighbors

We can use the template to create a function that returns the neighbors of a given node in a given graph. This is done by recursing through the adjacency list normally and checking whether each node is the given node (and returning the list if neighbors if it is).

```
;; 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))]))
```

Finding Paths

A path on a graph can be represented as an ordered list of nodes that form the path. We must use generative recursion to solve this problem (simple recursion will not work). If a path between two nodes exists, either:

1. The starting and target node are the same
2. There is a path between an out-neighbor of the starting node and the target node

This creates a sub-problem to solve, namely, the sub-problem of finding the path between the each of the starting node's out-neighbors and the target node.

If 1) is not satisfied by this step, the algorithm backtracks by applying these steps to each of the out-neighbors. This same process keeps happening until either the target node is found or all of the possibilities are exhausted.

(This process is "backtracking" because if the algorithm fails to find a node from node to the target, it "backtracks" and tries the next neighbor).

This suggests a `find-path` would be mutually recursive with a `find-path-list` function that applies find-path to all the out-neighbors, just like we saw with general trees.

```
;; (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

Sometimes, it is not computationally efficient to store an entire graph that represents a given situation (for example, a decision tree for all possible chess games). However, if a list of neighbors can be generated simply from knowing the value of a node (for example, in a chess game, the possible moves that turn), we can still use backtracking to search the graph.

This "blind backtracking" is the basis for many artificial intelligence programs.

Termination of `find-path`

For directed acyclic graphs, we know that any given iteration of `find-path` will never search the origin (or otherwise, there would be a cycle somewhere). Therefore, the each sub-problem must be smaller, so we know that `find-path` will always terminate in a DAG.

However, if there is a cycle in the graph, `find-path` will *never* terminate because it will keep searching around and around the graph.

To avoid this, we can add an accumulator that keeps track of which nodes have been visited already. Then, we will add a check to `find-path-list` that checks whether the node we are currently inspecting has already been inspected.

Here is the new code for `find-path-list`:

```
;; 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) ;; new wrapper function
    (find-path/acc orig dest g '()))
```

In this graph, the accumulator limits the depth of recursion of the function to the number of nodes, so we know that the program must terminate.

Efficiency

We now have a `find-path` that even works on graphs with cycles, but it still has one problem: inefficiency. It is possible that this algorithm will explore every possible path from the origin in a graph, which grows exponentially with each new node and edge. Indeed, even relatively small graphs may have tens of thousands of possible paths to search.

We can solve this by passing the results of a failed computation on to the next computation. That way, the same computation won't be performed twice. To do this, we will return the failed computation instead of false when a path isn't found. However, we now need to distinguish between successful and unsuccessful computations; we will define two new structures to do this:

```
(define-struct success (path))
;; A Success is a (make-success (listof Node))

(define-struct failure (visited))
;; A Failure is a (make-failure (listof Node))
```

```
;; A Result is (anyof Success Failure)
```

This way, we can use the auto-generated type predicates to tell whether the returned path is a successful one or not.

Final Implementation of `find-list`

```
;; 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])))
```