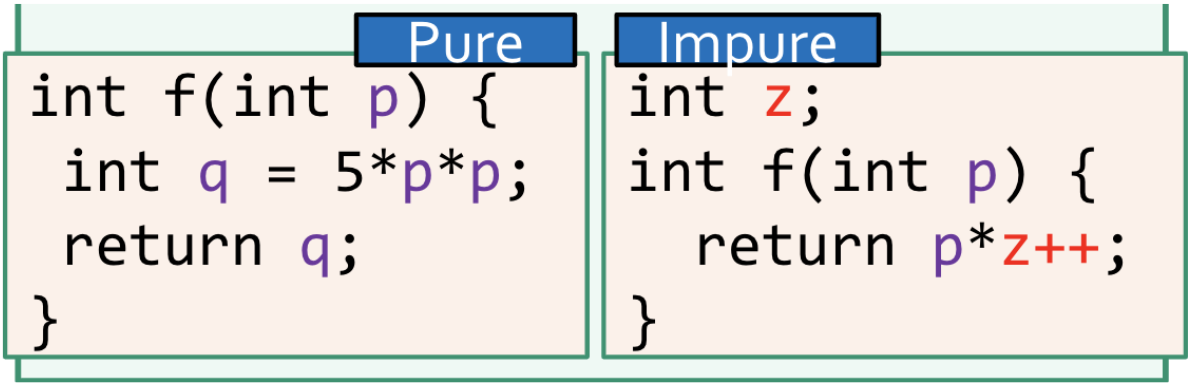
2 - Haskell

Functional Programming

- every function must take an argument
- every function must return a value
- calling a function will always output the same value given the same input
- functions are just like any data and can be stored in variables and passed as args
- functions are pure and have no side effects
- all variables are immutable and can never be modified

Pure Functions

- also called referentially transparent function
- given an input x, the function always returns the same output y
- it computes its output exclusively on input parameters Pros
- make it easier to reason, debug, test
- enable parallelism and compiler optimization
- help us formally analyze and prove
- our programs



Imperative vs Functional

Imperative	Functional
Algorithmic – series of statements and variable changes.	Transformation through function calls.
Changes in variable state are required.	All "variables" are constants - no changes are allowed!
Sequences of statements, loops and function calls.	Function calls and recursion – no loops, no statements!
Multi-threading is tricky and prone to bugs	Multi-threading is easy!
The order code executes is important.	The order of execution is of low importance!

Parallelism

- order of execution is low importance
- imperative functions require function calls to occur in order because of external effects of non pure functions
- FP can call each function using lazy evaluation (only evaluates a line if it needs to)

```
somefunc arg =
  if z == 7 then 0
            else y
  where
    y = func1 arg
    z = func2 arg
```

Haskell Utility Functions

`init :: [a] -> [a]`

- returns all but the last value of a list
- e.g. `init [1,2,3]` returns `[1,2]`

elem a::data b::list

- check if a::data is an element of b::list

fromIntegral a::Int

- converts a::Int to Double

reverse a::list

- reverses the list

any_type or any lowercase type signature

- defines any type of data

error a::String

- return an error

quicksort

```
-- Quicksort in eight lines!
qsort lst
 | lst == [] = []
 | otherwise = (qsort less_eq) ++ [pivot] ++ (qsort greater)
 where
    pivot = head lst
    rest_lst = tail lst
    less_eq = [a | a <- rest_lst, a <= pivot]
    greater = [a | a <- rest_lst, a > pivot]
```

show a::any

- show the actual value e.g. show (tail a) gives `[5]`

map f::func l::list

- returns mapped list given a function f

backticks on funcs

- allows u to use funs as infix e.g.

```
max a b
elem a b
-- is the same as
a `max` b
a `elem` b
```

Haskell

General

- Haskell is one of few purely functional languages
- follows lambda calculus - a theory that each function solely outputs the function definition with no outside effects
- no vars, sequences of statements, loops

Example Code

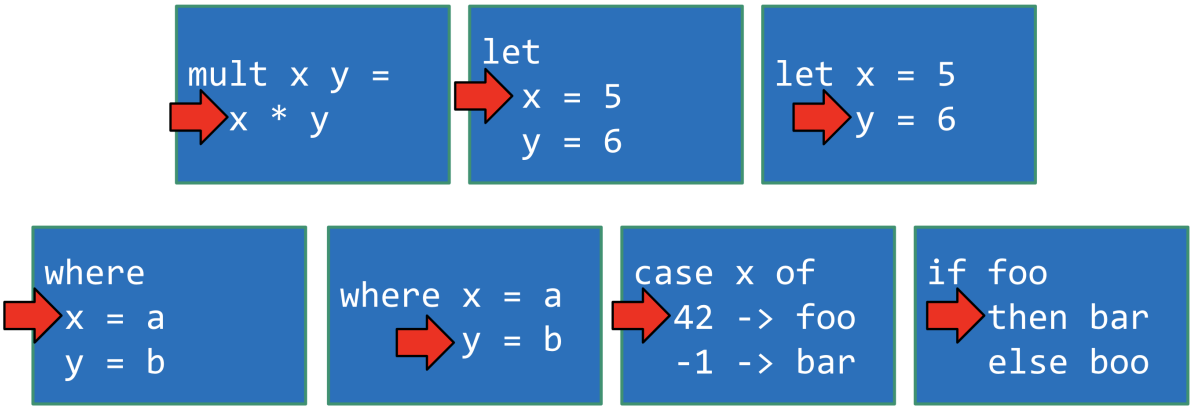
```
square x = x*x
hypot a b = sqrt (square a + square b)

Prelude> load hypot
Prelude>hypot 1 2
```

- Prelude loads standard library, etc.
- Haskell uses REPL paradigm like Python - Read Execute Print Loop

Indentation

- code part of an expression should be indented further than the declaration of the first line

- align the spacing for all items of a group



Data types

- statically typed - all var types determined at compile time → no changing types
- has type inference
- ints, big-ints (any number of digits), doubles, bools, chars

```
googol = 10^100 :: Integer
```

- the `:: Integer` is manually typing big-int, but Haskell will figure out its type
- unary negative needs parentheses e.g. `(-1)`
- logical ops allowed `&&` `||`

Composite data types

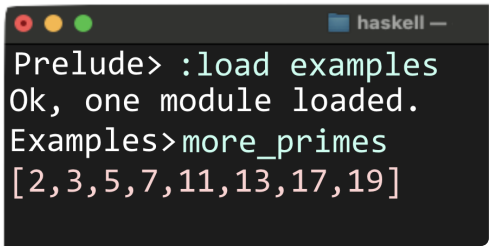
- tuple - collection of values, could be diff types
- lists - must be of same type
- string

Constructing Lists

Concatenation and Consing

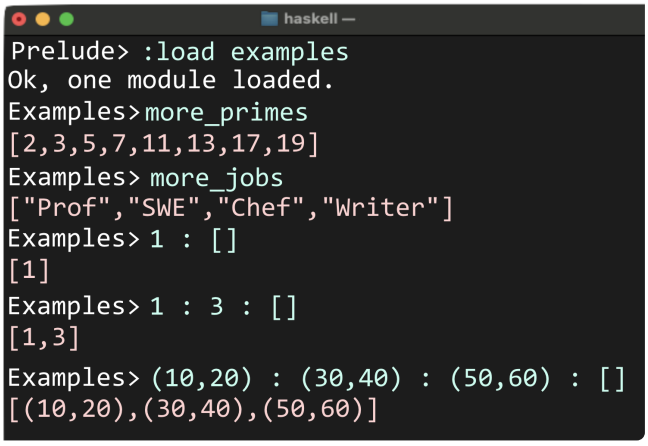
examples.hs

```
-- A List of primes
primes = [2,3,5,7,11,13]

-- A bigger List of primes
more_primes = primes ++ [17,19]
```



examples.hs

```
-- A List of primes
primes = [2,3,5,7,11,13]

-- A bigger List of primes
more_primes = primes ++ [17,19]

-- A List of jobs
jobs = ["SWE","Chef","Writer"]

-- A bigger list of jobs
more_jobs = "Prof" : jobs
```



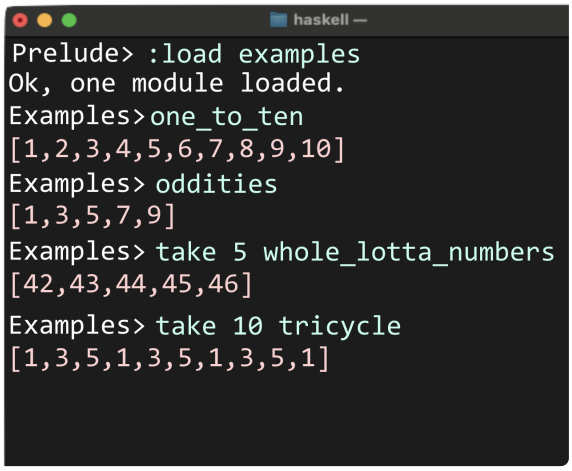
- cons operator `:` only prepends ONE item to the front of a list
- `++` requires 2 existing lists and concatenates 2 lists

Ranges

- will read patterns as subtractions e.g., `[1,3..10]` will do 3-1 then print until 10
- infinite lists and cycles only generate when required e.g., `[42, ...]`

examples.hs

```haskell
-- All #s between 1 and 10, inclusive
one_to_ten = [1..10]

-- Odd #s between 1 and 10
oddities = [1,3..10]

-- An infinite list from 42 onward!
whole_lotta_numbers = [42..]

-- An infinite cycle of
1,3,5,1,3,5,...
tricycle = cycle [1,3,5]
```



Tuples

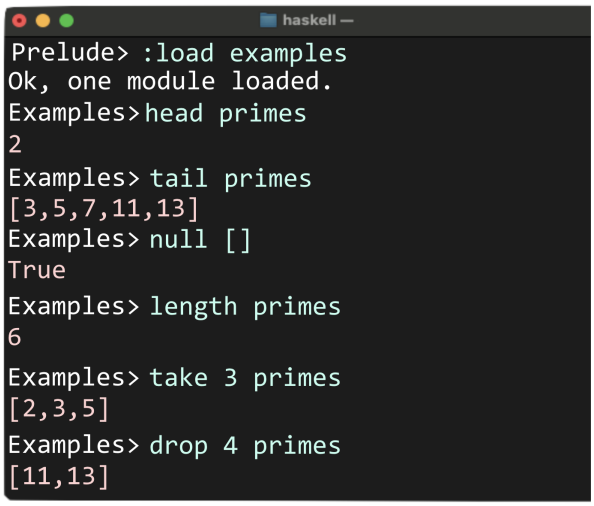
```haskell
grade :: (String,Int)
grade = ("me",4)

> fst grade -- fst only works on tuples of 2 values
> "me"
> snd grade
> 4
```

Lists

- can hold zero or more of same type
- implemented using linked lists i.e. O(n) to access nth item

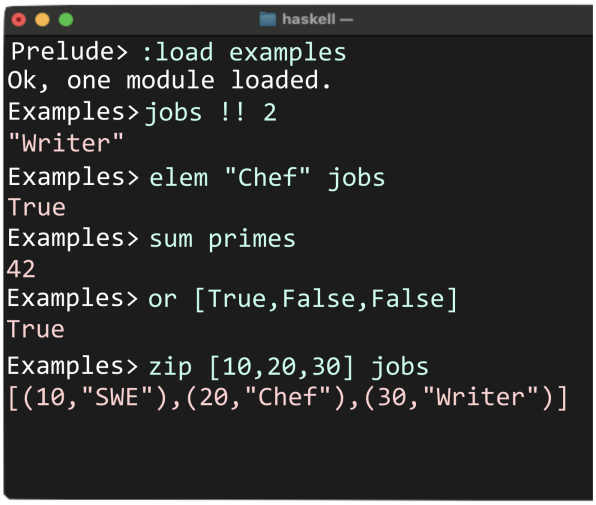
examples.hs

```
-- A List of primes
primes :: [Int]
primes = [2,3,5,7,11,13]

-- A List of jobs
jobs = ["SWE","Chef","Writer"]

-- A List of Lists
lol = [[1,2,3],[4,5],[6,7,8,9]]

-- A list of tuples
lot = [("foo",1),("bar",2),("boo",3)]

-- An empty list
mt = []
```



examples.hs

```
-- A List of primes
primes :: [Int]
primes = [2,3,5,7,11,13]

-- A List of jobs
jobs = ["SWE","Chef","Writer"]

-- A List of Lists
lol = [[1,2,3],[4,5],[6,7,8,9]]

-- A list of tuples
lot = [("foo",1),("bar",2),("boo",3)]

-- An empty list
mt = []
```

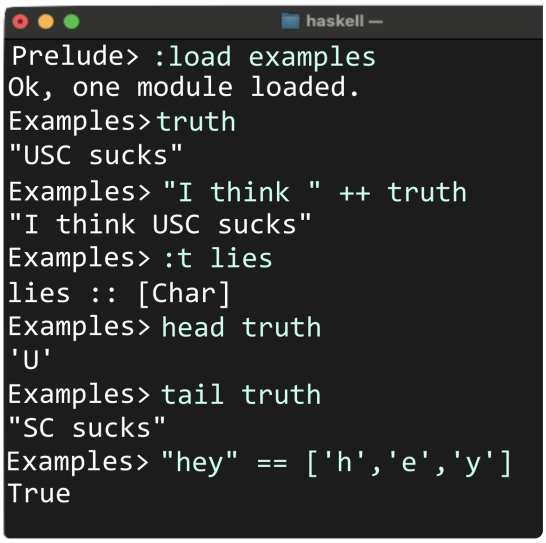


Strings

- a string is equivalent to a list of chars e.g. `String == [Char]`

examples.hs

```
-- Defining a String w/explicit type
truth :: [Char]
truth = "USC sucks"

-- Defining a string w/o a type
lies = "USC kids are smart"
```



Comprehension

- used to generate arbitrarily complex lists of items with declarative syntax
- create a new list based on one or more existing lists

```
output_list = [f x | x ← input_list,(guard1 x),(guard2 x)]

square1 = [x^2 | x ← input_list]
square2 = [x^2 | x ← input_list, x>5, x<20]
```

```
out = [(f x y) | x←in1, y←in2, (guard1 x), (guard2 y)]

all_prod = [x*y | x←l1, y←l2, even x, odd y, x*y < 15]
```

Definition

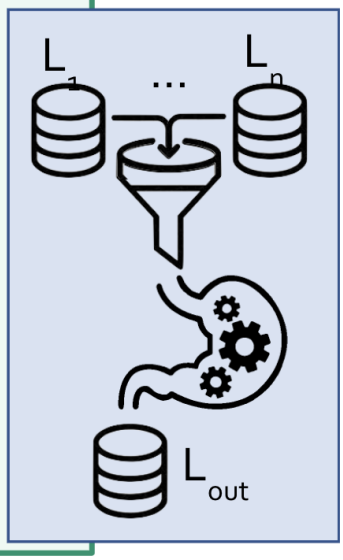
Definition

Comprehension

A list comprehension is a F.P. construct that lets you easily create a new list based on one or more existing lists.

With a list comprehension, you specify the following inputs:

1. One or more input lists that you want to draw from (these are called "generators")
2. A set of filters on the input lists (these are called "guards")
3. A transformation applied to the inputs before items are added to the output list.



Dependent Generators

- generators that generate from the values of another generator

```
-- Generate right triangles
right_triangle_tuples = [ (a,b,c) |
   a <- [1..10], b <- [a..10], c <- [b..10],
   a^2+b^2==c^2 ]
```

Functions

What's in a function

1. type signature: inputs and return value
2. function name and parameters
3. expression that defines behavior

```
string insult(string name, string smell) {
  string s = name + " smells like " +
             smell + " and doesn't floss.";
  return s;
}
```

```
insult :: String -> String -> String
insult name smell =
 name ++ " smells like " ++
 smell ++ " and doesn't floss."

main::IO()
main = do
  putStr "What is your name? "
  name <- getLine
  putStrLn (insult name "unwashed dog")
```

Left Associative

```
f :: Int -> Int
f x = x^2

g :: Int -> Int
g x = 3*x


-- We want to compute f(g(x))
compute_f_of_g x =
  f (g x)

-- We want to compute f(x*10)
compute_f_of_x_times_ten x =
  f (x * 10)
```

Haskell evaluates function calls from left to right.

As such, its function application is called "left-associative."

In addition, Haskell always calls functions before evaluating operators.

Mathematics	Haskell
f(x)	f x
f(x,y)	f x y
f(g(x))	f (g x)
f(x,g(y))	f x (g y)
f(x)g(y)	f x * g y

Local Bindings

- `let` and `where` locally associate/bind variable names as "temporary" vars
- beware of global variable shadowing – need to check the behavior

Let

```
get_nerd_status gpa study_hrs =
    let
        gpa_part = 1 / (4.01 - gpa)
        study_part = study_hrs * 10
        nerd_score = gpa_part + study_part
    in
        if nerd_score > 100 then
            "You are super nerdy!"
        else "You're a nerd poser."
```

Where

```
get_nerd_status gpa study_hrs =
  if nerd_score > 100
     then "You are super nerdy!"
     else "You're a nerd poser."
  where
      gpa_part = 1 / (4.01 - gpa)
      study_part = study_hrs * 10
      nerd_score = gpa_part + study_part
```

Nested Functions

```
whats_the_behavior_of name =
 if name == "Carey"
    then behaves_like name "twelve year-old"
    else behaves_like name "grown-up"
 where
   behaves_like n what =
      n ++ " behaves like a " ++ what ++ "!"
```

- Nested Functions have visibility of all outside scope, so they can "see" the immediate outer scope variables

- this allows us to simplify the previous to:

nestfunc.hs

```
-- Function to describe someone's behavior
whats_the_behavior_of name =
 if name == "Carey"
    then behaves_like name "twelve year-old"
    else behaves_like name "grown-up"
 where
   behaves_like n what =
    name ++ " behaves like a " ++ what ++ "!"
```

Conditionals

- every `if` requires an `else` - there are no void functions in Haskell
- else-if is implemented using nested conditionals

```
if <expression> then <expression>
                else <expression>
```

Guards

- basically, syntactic sugar for conditionals

```
if <condition> then <do-this>
```

```
| <condition> = <do-this>
```

In Haskell, we can define a function as a series of one or more guards.

```
somefunc param1 param2
  | <if-x-is-true> = <run-this>
  | <else-if-y-is-true> = <run-that>
  | <else-if-z-is-true> = <run-other>
  | otherwise = <run-this-otherwise>
```

Pattern Matching

- again syntactic sugar
- specifying specific outputs for each unique input (can leave generalized if needed)
- the patterns match IN ORDER - runs the first matched function implementation
- you can use `_` to mean "any value" or "not required"

```
course_critic  _    "cs131" =
  "I hope you like coding in 20 languages!"
```

critic.hs

```
course_critic :: String -> String -> String
course_critic "Carey" "CS131" =
  "Get ready for pop-tarts and Haskell!"
course_critic "Eggert" course =
  course ++ " is gonna be difficult."
course_critic prof "cs131" =
  "I hope you like coding in 20 languages!"
course_critic prof course =
  course ++ " with " ++ prof ++ " is fun!"
```

Tuple Pattern Matching

```
-- Original way
exp :: (Int,Int) -> Int
exp t = (fst t) ^ (snd t)

-- Version #2: With pattern matching
expv2 :: (Int,Int) -> Int
expv2 (base,exponent) = base ^ exponent

-- Version #3: With simple and complex matching
expv3 :: (Int,Int) -> Int
expv3 (base,0) = 1
expv3 (base,exponent) = base ^ exponent
```

```
-- Extract the first value of a tuple
extract_1st :: (type1,type2,type3) -> type1
extract_1st (a,_,_) = a

-- Extract the second value of a tuple
extract_2nd :: (type1,type2,type3) -> type2
extract_2nd (_,b,_) = b

-- Extract the third value of a tuple
extract_3rd :: (type1,type2,type3) -> type3
extract_3rd (_,_,c) = c
```

List Pattern Matching

- List patterns identified by parentheses
- elements separated by colons, last elem must be rest of the list
- the pattern is: `(first:rest)` where `first::any` and `rest::list`
- rest is always a list
- the patterns are checked in order

Structure

```haskell
-- Extract the first item from a list
get_first_item (first:rest) =
   "The first item is " ++ show(first)

-- Extract all but the first item of a list
get_rest_items (first:rest) =
   "The last n-1 items are " ++ show(rest)

-- Extract the second item from a list
get_second_item (first:second:rest) =
   "The second item is " ++ show(second)
```

Examples

favs.hs

```haskell
favorites :: [String] -> String
favorites [] = "You have no favorites."
favorites (x:[]) = "Your favorite is " ++ x
favorites (x:y:[]) = "You have two favorites: "
 ++ x ++ " and " ++ y
favorites ("chocolate":xs) =
  "You have many favs, but chocolate is #1!"
favorites (x:xs)
 | "strawberry" `elem` xs =
  "How could you like strawberry? Yick."
 | x /= "chocolate" = "You have bad taste."
 | otherwise = "You have good taste."
```

```
haskell —
Prelude> :load favs
Ok, one module loaded.
Favs>favorites []
"You have no favorites."
Favs>favorites ["chocolate"]
"Your favorite is chocolate"
Favs>favorites ["mint","earwax"]
"You have two favorites: mint and earwax"
Favs>favorites ["chocolate","egg","dirt"]
"You have many favs, but chocolate is #1!"
Favs>favorites ["tea","coffee","carrot"]
"You have at least three favorites!"
```

Guards vs pattern matching

The Old Way: With Guards

sm0l.hs

```
-- Find sm0l-est item w/o pattern matching
sm0lest lst
 | lst == [] = error "empty list"
 | length lst == 1 = first
 | otherwise = min first (sm0lest rest)
 where
   first = head lst
   rest = tail lst
```

The New Way: With Pattern Matching

```
-- Find sm0l-est item with pattern matching

sm0lest [] = error "empty list"
sm0lest (x:[]) = x
sm0lest (x:xs) = min x (sm0lest xs)
```

rev.hs

```
-- Reverse a list w/o pattern matching
rev lst
 | lst == [] = []
 | otherwise = (rev rest) ++ [first]
 where
  first = head lst
  rest  = tail lst
```

```
-- Reverse a list with pattern matching

rev [] = []
rev (x:xs) = (rev xs) ++ [x]
```

Cheat Sheet

Pattern	Example	Argument	Succeeds?	Bindings
7	func 7 = "Lucky 7"	7	Yep	N/A
x	func x = x + 5	7	Yep	x ← 7
(x:xs)	func (x:xs) = ...	[10,20,30]	Yep	x ← 10, xs ← [20,30]
(x:xs)	func (x:xs) = ...	[[10,20,30]]	Yep	x ← [10,20,30], xs ← []
(x:xs)	func (x:xs) = ...	["UCLA"]	Yep	x ← "UCLA", xs ← []
(x:xs)	func (x:xs) = ...	"UCLA"	Yep	x ← 'U', xs ← "CLA"
(7:xs)	func (7:xs) = ...	[7,8,9]	Yep	xs ← [8,9]
(7:xs)	func (7:xs) = ...	[8,9,10]	Nope – doesn't match	N/A
(_:xs)	func (_:xs) = ...	[8,9,10]	Yep	xs ← [9,10]
(x:_:xs)	func (x:_:xs) = ...	[8,9,10]	Yep	x ← 8, xs ← [10]
[]	func [] = "Empty list"	[]	Yep	N/A
[]	func [] = "Empty list"	[42]	Nope	N/A
[x]	func [x] = ...	[100]	Yep	x ← 100
[x]	func [x] = ...	[100,200]	Nope	N/A
[42,x]	func [42,x] = ...	[42,43]	Yep	x ← 43

Higher Order Functions

First Class Functions

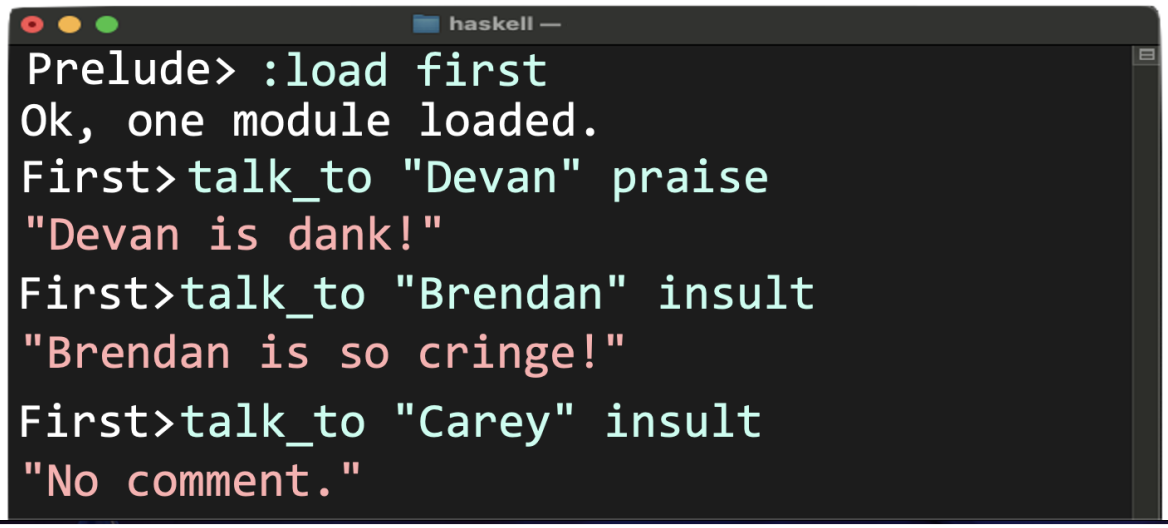
- when a function is treated like any other data/vars
- stored in vars, used as args, returns values, stored in data structure

- higher order functions' type definition must be wrapped in parentheses

Higher Order Functions

first.hs

```
insult :: String -> String
insult name = name ++ " is so cringe!"

praise :: String -> String
praise name = name ++ " is dank!"

talk_to :: String -> (String -> String) -> String
talk_to name talk_func
  | name == "Carey" = "No comment."
  | otherwise = (talk_func name)
```



Returning Functions

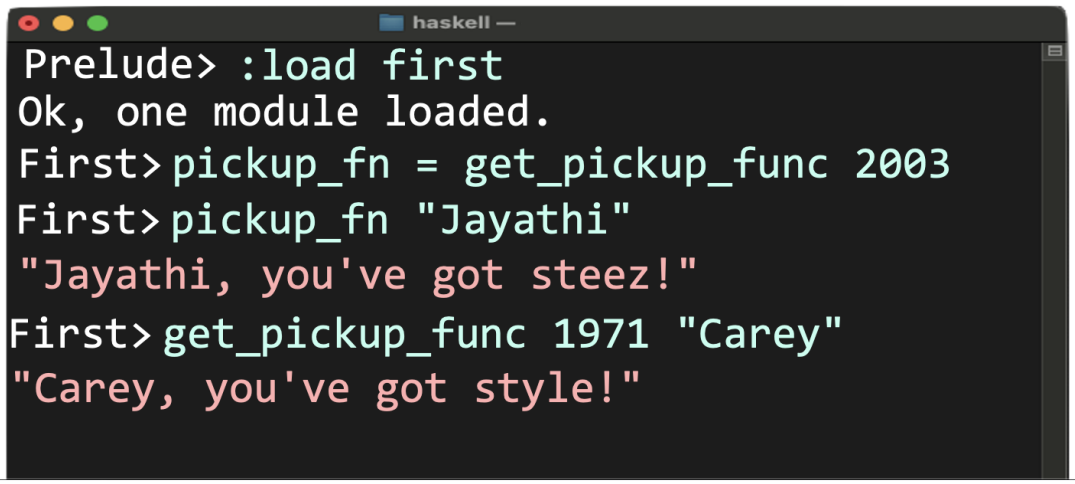
- you can save returned functions to variables → example w/ "jayathi"

- you can also overlload higher ordered returned functions passing the second parameter in the same line
→ look at second example w/ "carey"
- Because Haskell is left associative, it does the greedy choice and does `get_pickup_func carey`. Then that is now a function and passes `"carey"` as a param

first.hs

```
get_pickup_func :: Int -> (String -> String)
get_pickup_func born
 | born >= 1997 && born <= 2012 = pickup_genz
 | otherwise = pickup_other
 where
  pickup_genz name = name ++ ", you've got steez!"
  pickup_other name = name ++ ", you've got style!"
```

`pickup_fn` `name ++ ", you've got steez!"`



Fundamental Higher Order Funcs

Mappers

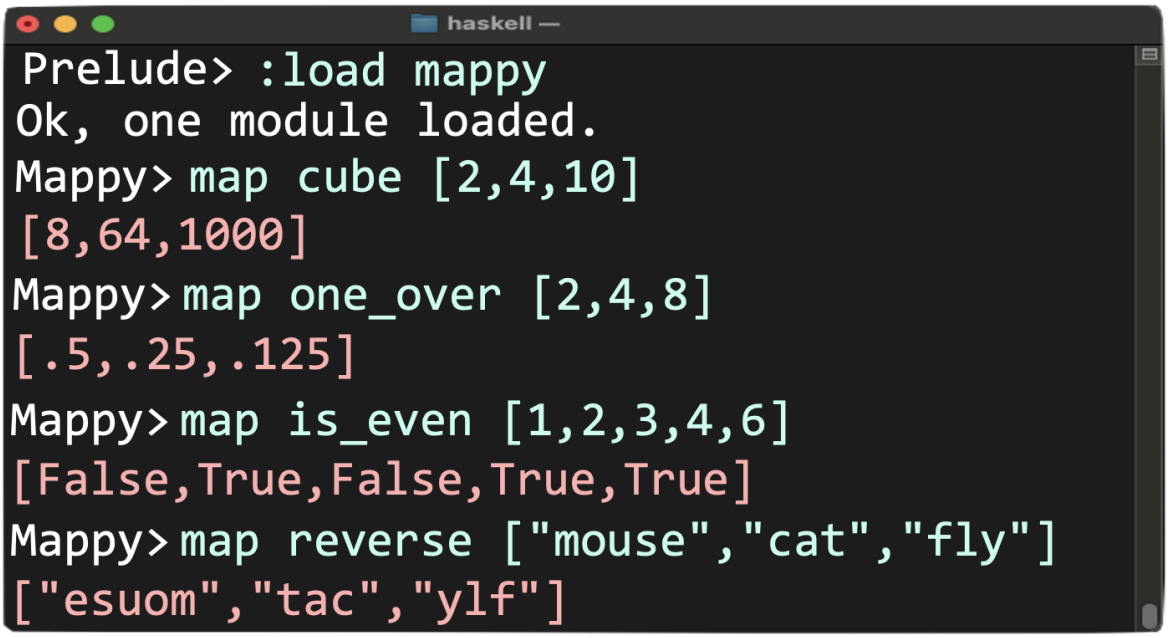
- one-to-one map of a list of values to another list using a **transform function**

- Haskell has built-in `map::(a→b) → [a] → [b]` s.t. `map f::func l::list`
- returns a new list (not in place because all data in Haskell is immutable)

Example

mappy.hs

```
cube :: Double -> Double
cube x = x^3

one_over  :: Double -> Double
one_over x = 1/x

is_even :: Int -> Bool
is_even x = x `mod` 2 == 0
```



How it works

map.hs

```
map :: (a -> b) -> [a] -> [b]
map func [] = []
map func (x:xs) =
  (func x) : map func xs
```

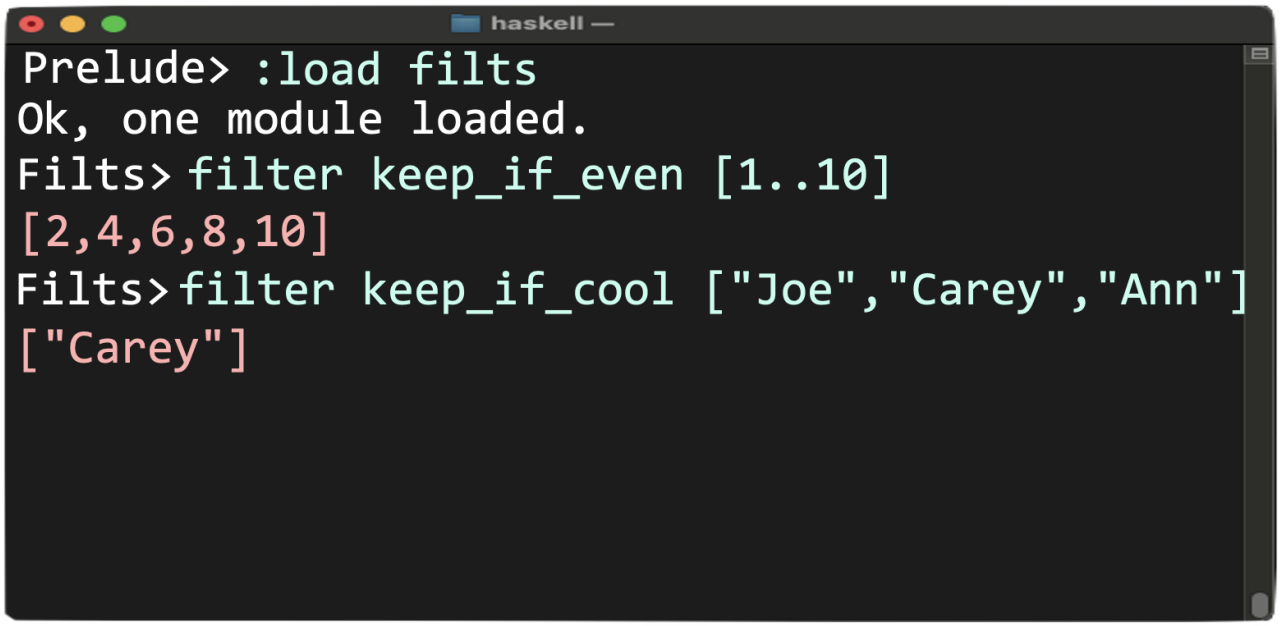
Filters

- filters out items from a list using a **predicate function**
- built-in `filter::(a→Bool) → [a] → [a]` s.t. `filter f::func l::list`
- takes a function that returns a boolean

Example

filts.hs

```
keep_if_even :: Int -> Bool
keep_if_even val = val `mod` 2 == 0

keep_if_cool  :: String -> Bool
keep_if_cool s =
 s `elem` ["Carey","Boelter"]
```



How it works

filter.hs

```
filter :: (a -> Bool) -> [a] -> [a]
filter predicate [] = []
filter predicate (x:xs)
 | (predicate x) = x : (filter predicate xs)
 | otherwise = filter predicate xs
```

Reducers

- operates on a list and collapses to a single value
- takes a function to process each element, an initial accumulator value, list to operate on
- has 2 built-in: `foldl` and `foldr`

Foldl

- Pseudocode

```
foldl(f, initial_accum, lst):
  accum = initial_accum
  for each x in lst:
    accum = f(accum, x)
  return accum
```

```
// Example f() functions:

int f1(int accum, int x)
  { return accum + x; }

string f2(string accum, string x)
  { return x + accum; }
```

- folds left associative - i.e. in order
- internals

```
foldl f accum [] = accum
foldl f accum (x:xs) =
  foldl f new_accum xs
 where new_accum = (f accum x)
```

Foldr

- pseudocode

```
foldr(f, initial_accum, lst):
  accum = initial_accum
  for each x in lst from back to front:
    accum = f(x, accum)
  return accum
```

```
// Example f() functions:
int f1(int x, int accum)
  { return accum + x; }

string f2(string x, string accum)
  { return x + accum; }
```

- folds right associative - reverse
- internals

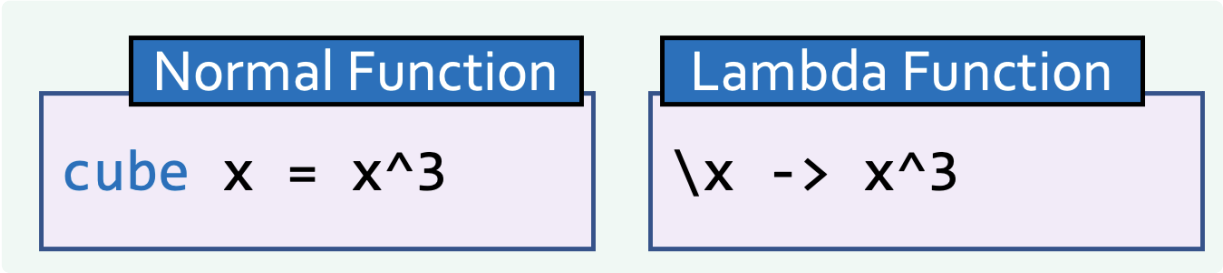
```
foldr f accum [] = accum
foldr f accum (x:xs) =
  f x (foldr f accum xs)
```

Advanced FP Topics

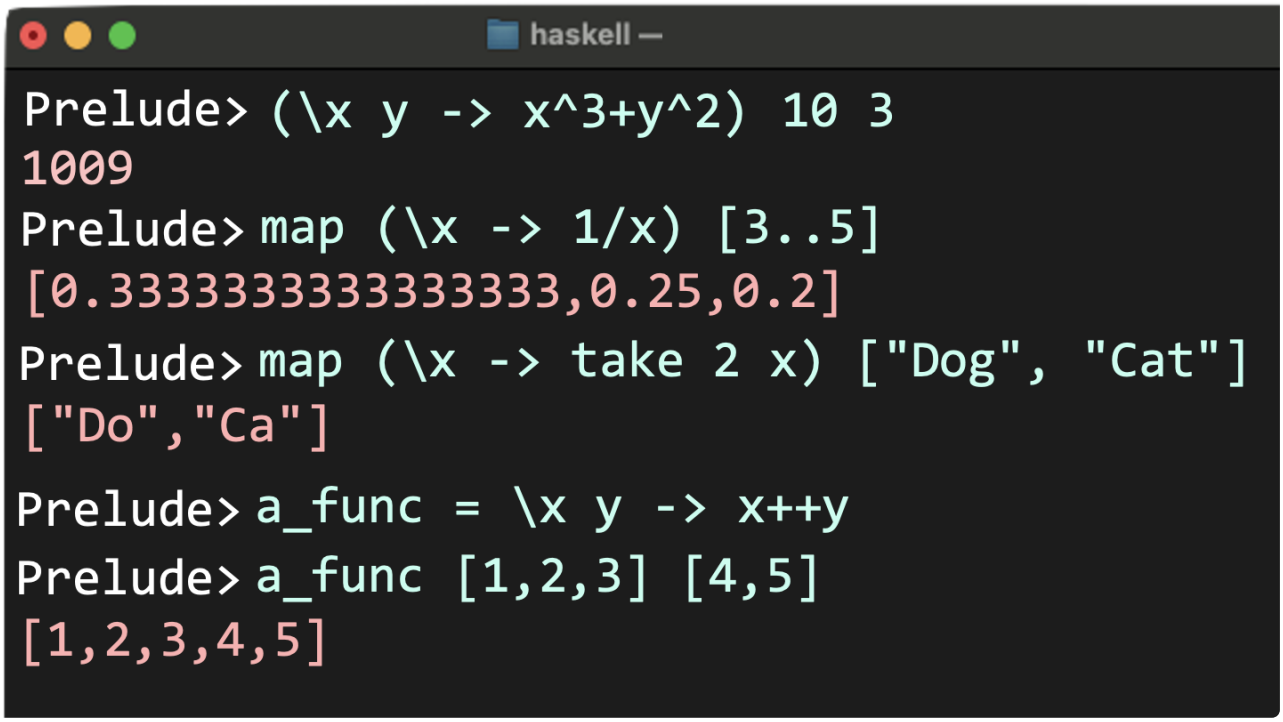
Lambda functions

- a nameless function - used to make readability better

- especially used in higher-order funcs



Examples



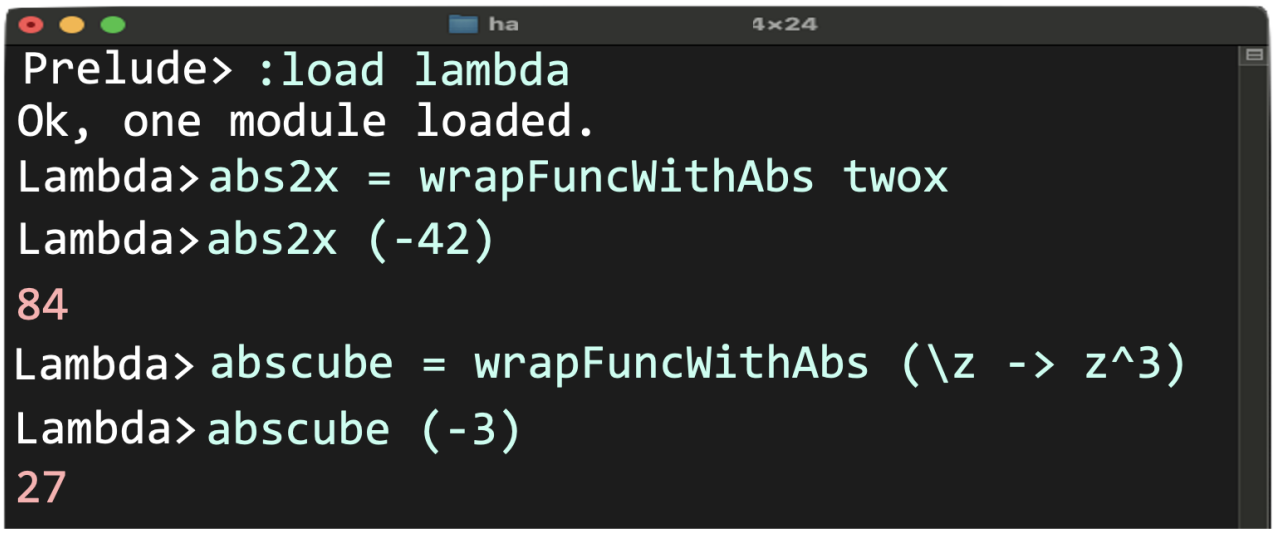
Saving lambdas as functions

lambda.hs

```
wrapFuncWithAbs func =
    (\x -> abs (func   x))

cubed x = x^3
twox x = 2*x
```

abs2x `\x -> abs (twox   x)`

abscube `\x -> abs ((\z -> z^3) x)`



Closures

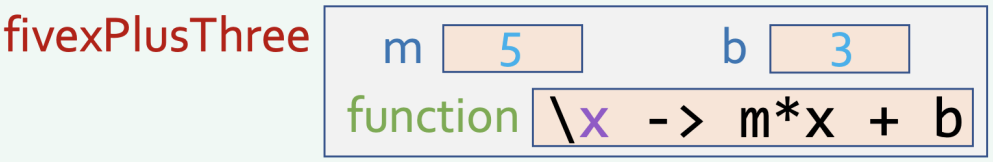
- a function that uses a lambda that captures values

Definition

Closure

A closure is a combination of two things:

1. A function of zero or more arguments that we wish to run at some point in the future
2. A list of "free" variables and their values that were captured at the time the closure was created

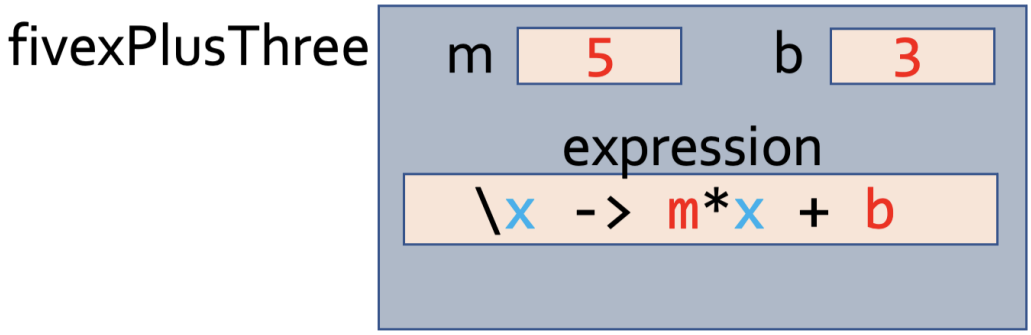


When a closure later runs, it uses the values of the free variables captured at the time it was created*.

- captured values are values that are not parameters of the lambda but are enclosed within the lambda - also called "free variables"
- in the following, `twoxPlusOne` and `fivexPlusThree` are closures that are functions that capture 2,1 or 5,3, respectively when called with a value x

lambda.hs

```
slopeIntercept m b = (\x -> m*x + b)


twoxPlusOne = slopeIntercept 2 1

fivexPlusThree = slopeIntercept 5 3
```



Partial Function Application

- a way to decrease the number of params of a function by 1 to create a new function that returns a function that accepts the last arg

Definition

Partial Function Application

PFA is an operation where we define a new function g by combining:

1. An existing function f that takes two or more arguments, with
2. Default values for one or more of those arguments.

The new function g is a specialization of f, with hard-coded values for some of f's parameters.

Once defined, we can then call g with those arguments that have not yet been hard-coded.

With PFA, we call a function with less than the full number of arguments:

$$f(x, y, z) = x + y + z$$

$$f' = f(10, 20)$$

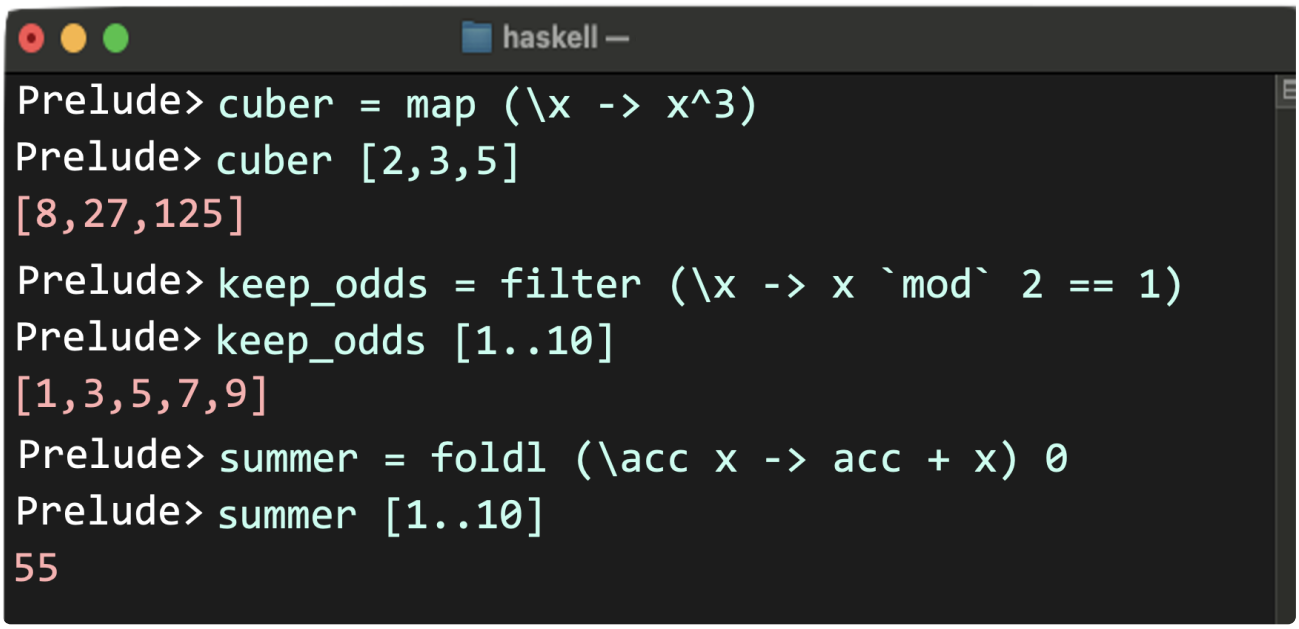
The function returns a new version of itself that takes just the missing parameters. The new function has the earlier parameters replaced by the passed-in values:

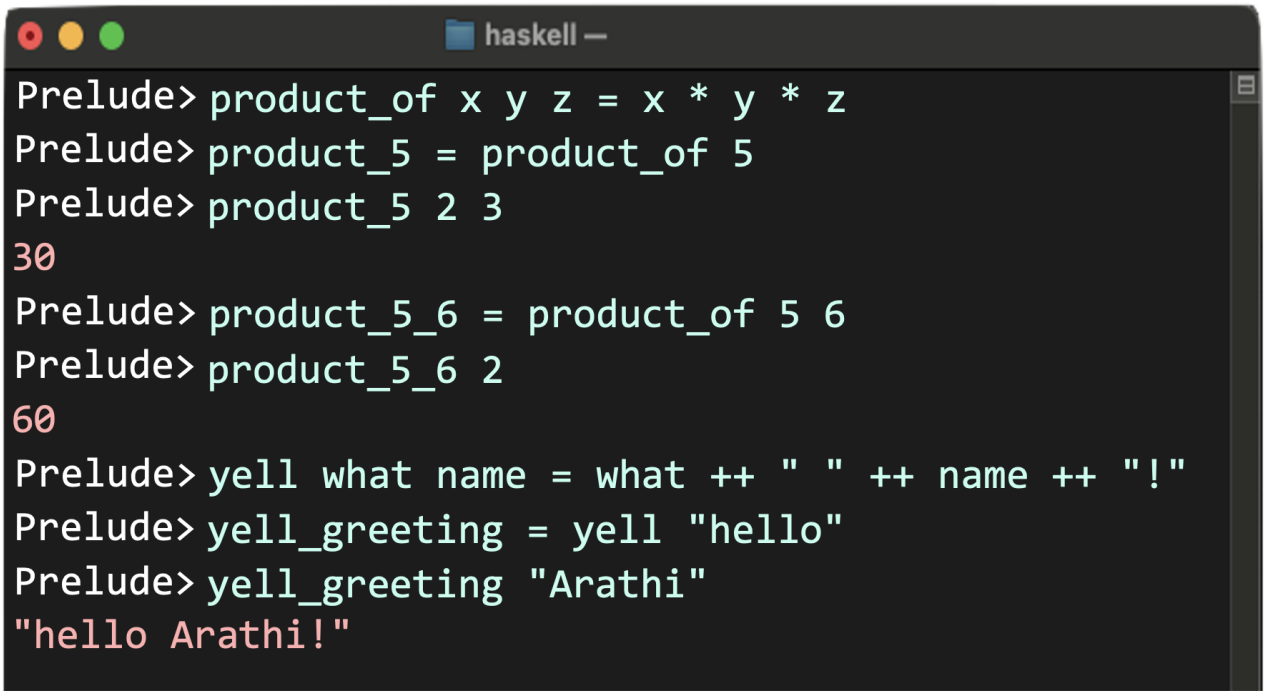
$$f'(z) = 10 + 20 + z$$

We can later call our new function, passing in just the remaining argument(s) to get a result.

$$result = f'(30) \Rightarrow 60$$

Examples





Currying

- a way to trasnform a multi-arg func into a series of funcs that take only 1 arg

Currying

Definition

Currying is the concept that you can represent any function that takes multiple arguments by another that takes a single argument and returns a new function that takes the next argument, etc.

$y = f(a,b,c)$ → $y = ((f_c\ a)\ b)\ c$

Each function takes one argument and returns another function as its result (except for the last function which returns the result).

```
function f(x, y, z) { return x + y + z; }
```

```
function f(x) {
  function g(y) {
    function h(z) {
      return x + y + z;
    }

    return h;
  }

  return g;
}
```

Type Defs

So do you remember those unusual Haskell function type signatures?

```
mult3 :: Double -> Double -> Double -> Double
mult3 x y z = x * y * z
```

Well, here's what they should really look like given that all functions are curried:

```
mult3 :: Double -> (Double -> (Double -> Double))
mult3 x y z = x * y * z
```

Algebraic Data Types

Definition

Definition

Algebraic Data Types

"Algebraic Data Type" is the term we use in functional programming to describe a user-defined data type that can have multiple fields.

The closest thing to algebraic data types would be C++ structs and enumerated types.

We use algebraic data types to create complex data structures like trees.

Structure

- data is used to specify ADTs all data and variants must be Capitalized
- the pipe separates variants of the data type
- you can construct data types of data types
  - a meal has variants: Breakfast, Lunch, Dinner, Fasting

- each of these is made up of data types of ADTs

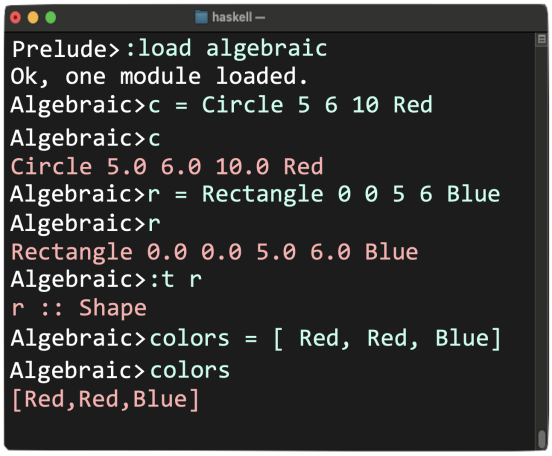
```
data Drink = Water | Coke | Sprite | Redbull
data Veggie = Broccoli | Lettuce | Tomato
data Protein = Eggs | Beef | Chicken | Beans

data Meal =
 Breakfast Drink Protein |
 Lunch Drink Protein Veggie |
 Dinner Drink Protein Protein Veggie |
 Fasting

careys_meal = Breakfast Redbull Eggs
pauls_meal = Lunch Water Chicken Broccoli
davids_meal = Fasting
```

Examples

- use `deriving Show` to show variable values

algebraic.hs

```
data Color = Red | Green | Blue
             deriving Show

data Shape =
  Circle
   Float Float -- x, y
   Float       -- radius
   Color |
  Rectangle
   Float Float -- x1, y1
   Float Float -- x2, y2
   Color
  deriving Show
```



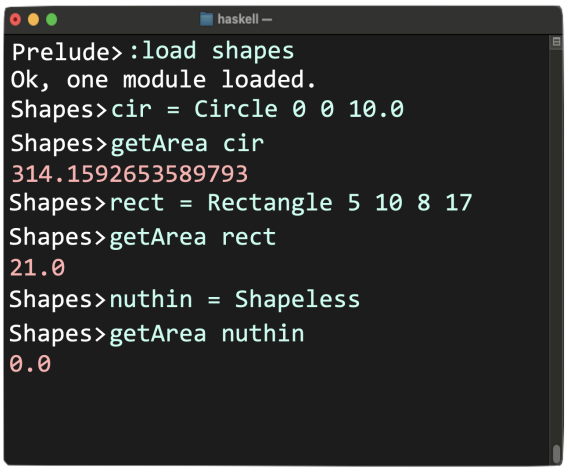
Pattern Matching

shapes.hs

```
data Shape =
   Shapeless
 | Circle
    -- x      y    rad
    Double Double Double
 | Rectangle
    -- x1    y1     x2     y2
    Double Double Double Double
 deriving Show

getArea :: Shape -> Double
getArea Shapeless = 0
getArea (Circle _ _ r) = pi * r^2
getArea (Rectangle x1 y1 x2 y2) =
  (abs (x2-x1)) * (abs (y2-y1))
```

In functional languages, we use pattern matching to process algebraic variables.

Let's see an example with our shapes.



Data Structures

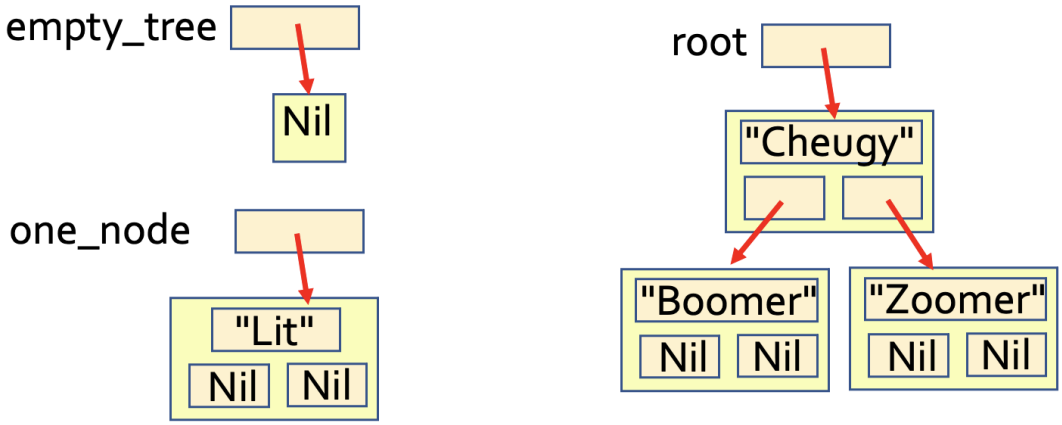
- immutability makes debugging, parallelization, and garbage collection easier
- BUT, some data structures are inefficient

Trees

- construct using ADTs - see HW2
- searching

bst.hs

```
data Tree =
  Nil |
  Node String Tree Tree

search Nil val = False
search (Node curval left right) val
 | val == curval = True
 | val < curval = search left val
 | otherwise = search right val
```

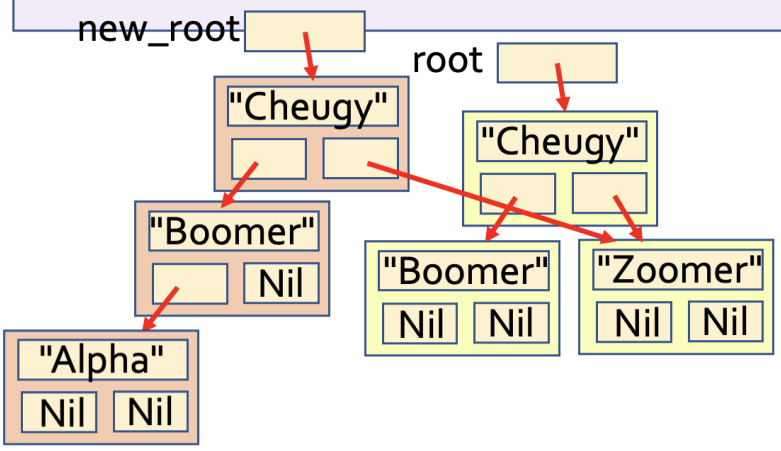


- insertion - only change the path to root

$$O(\log n)$$

bst.hs

```
data Tree =
  Nil |
  Node String Tree Tree
```



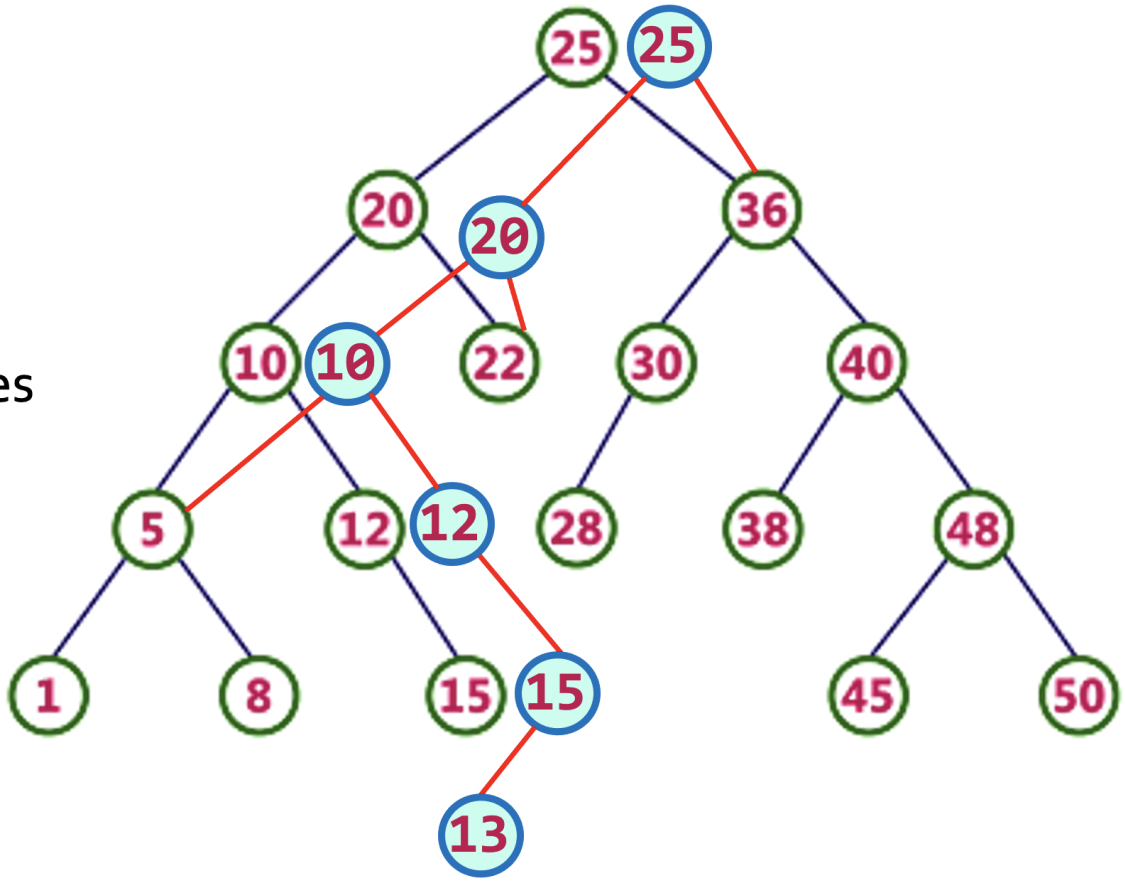
-

```
insert(node, v) // Returns a new tree
 if isEmpty(node) then
   return new Node(v,Nil,Nil)

 orig_l_subtree = node->left
 orig_r_subtree = node->right

 If v == node->val then
   return new Node(node->val, orig_l_subtree,
                              orig_r_subtree)
 Else if v < node->val then
   new_l_subtree = insert(orig_l_subtree , v)
   return new Node(node->val, new_l_subtree,
                              orig_r_subtree)
 Else if v > node->val then
   new_r_subtree = insert(orig_r_subtree , v)
   return new Node(node->val, orig_l_subtree,
                              new_r_subtree)
```

-

```
insert Nil v =
  Node v Nil Nil


insert (Node val left right) v

  | v == val =
      Node val left right

  | v < val =
      Node val (insert left v) right

  | v > val =
      Node val left (insert right v)
```

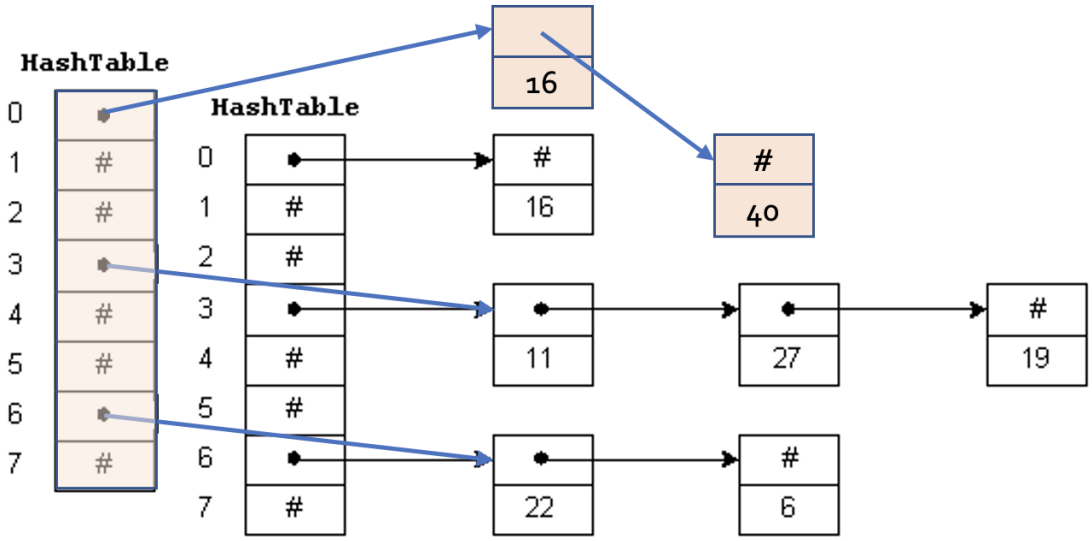
es



-
- garbage collection deletes unlinked old nodes

Hash Tables

- requires O(buckets) for each change because you need to reconstruct the entire hash array



Linked List

- same as imperative
- O(1) at the front, O(n) at the end, somewhere in between for the middle

Functional Programming... In Summary

Every function must take an argument.	Every function must return a value.	Functions are "pure" and have no side effects
Calling the same function with the same input always returns the same result.	All variables are "immutable" and can never be modified!	Functions are just like any other data and can be stored in variables and passed as arguments.

While there are no purely functional languages in mainstream use, the paradigm has influenced every major language!