Haskell in 3 Hours

check out:

- Teach Yourself Programming in Ten Years
  `http://norvig.com/21-days.html`

check out:

- Teach Yourself Programming in Ten Years
  `http://norvig.com/21-days.html`
- Haskell Standard Libraries
  `http://haskell.org/ghc/docs/latest/html/libraries`

check out:

- Teach Yourself Programming in Ten Years
  `http://norvig.com/21-days.html`
- Haskell Standard Libraries
  `http://haskell.org/ghc/docs/latest/html/libraries`
- Hoogλe

  `http://haskell.org/hoogle`

check out:

- Teach Yourself Programming in Ten Years
  `http://norvig.com/21-days.html`
- Haskell Standard Libraries
  `http://haskell.org/ghc/docs/latest/html/libraries`
- Hoogλe
  `http://haskell.org/hoogle`
- **Learn You a Haskell for Great Good!**
  `http://learnyouahaskell.com`

check out:

- Teach Yourself Programming in Ten Years
  `http://norvig.com/21-days.html`
- Haskell Standard Libraries
  `http://haskell.org/ghc/docs/latest/html/libraries`
- Hoogλe
  `http://haskell.org/hoogle`
- **Learn You a Haskell for Great Good!**
  `http://learnyouahaskell.com`

for the fearless:

- Haskell98 Report
  `http://haskell.org/onlinereport`

you will see:

today

interactive evaluator, expressions, lists, functions, recursion, types, polymorphism, overloading, pattern matching, local bindings, currying

next week

first-class functions: `map`, `filter`, `fold`, ..., application, composition, datatypes, records, class declarations, input/output, `do`-notation, files

purely functional programming

imperative programs

- tell computer what to do
- execute sequence of tasks
- change state: `a := 5; ...; a := 42`

purely functional programming

imperative programs

- tell computer what to do
- execute sequence of tasks
- change state: `a := 5; ...; a := 42`

purely functional programs

- tell computer what stuff is
- `factorial n = product [1..n]`
- no side-effects
- referential transparency
- simple reasoning or even proofs

laziness

- calculate things only when needed
- program = series of **data transformations**

```
doubleMe(doubleMe(doubleMe([1,2,3,4,5,6,7,8])))
```

- computed in one pass

static types

- **compiler** knows types of everything
- many errors caught at compile time: `17 + "4"`
- type inference $\rightsquigarrow$ few annotations necessary

interactive evaluator

- invoked by typing `ghci` in terminal
- define function, for example, in `myfunctions.hs`
- load by typing `:l myfunctions`
- reload by typing `:r`

```
# ghci
GHCi, version 6.10.3: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer ... linking ... done.
Loading package base ... linking ... done.
Prelude> :set prompt "ghci> "
ghci>
```

arithmetic

```
ghci> 2 + 15
17
ghci> 49 * 100
4900
ghci> 1892 - 1472
420
ghci> 5 / 2
2.5
```

implicit precedences

```
ghci> (50 * 100) - 4999
1
ghci> 50 * 100 - 4999
1
ghci> 50 * (100 - 4999)
-244950
```

negating numbers

`5 * (-3)`, not `5 * -3`

booleans and equality

```
ghci> True && False
False
ghci> True && True
True
ghci> False || True
True
ghci> not False
True
ghci> not (True && True)
False
```

```
ghci> 5 == 5
True
ghci> 1 == 0
False
ghci> 5 /= 5
False
ghci> 5 /= 4
True
ghci> "hello" == "hello"
True
```

type safety

```
ghci> 5 + 4.0
9.0
ghci> 5 + "four"
<interactive>:1:0:
    No instance for (Num [Char])
      arising from a use of ‘+’ at <interactive>:1:0-9
    Possible fix: add an instance declaration for (Num [Char])
    In the expression: 5 + "four"
    In the definition of ‘it’: it = 5 + "four"
```

functions

```
ghci> succ 8
9
ghci> min 9 10
9
ghci> min 3.4 3.2
3.2
ghci> max 100 101
101
ghci> succ 9 + max 5 4 + 1
16
ghci> (succ 9)+(max 5 4)+1
16
```

```
ghci> succ 9 * 10
100
ghci> succ (9 * 10)
91
ghci> div 92 10
9
ghci> 92 ‘div‘ 10
9
ghci> succ (succ 7)
9
ghci> succ(succ(7))
9
```

own functions

```
doubleMe x = x + x

doubleUs x y = x*2 + y*2

doubleUs x y = doubleMe x + doubleMe y

doubleSmallNumber x = if x > 100 then x else x*2

doubleSmallNumber’ x = (if x > 100 then x else x*2) + 1
```

lists

```
ghci> let numbers = [4,8,15,16,23,42]

ghci> numbers
[4,8,15,16,23,42]

ghci> [1,2,3,4] ++ [9,10,11,12]
[1,2,3,4,9,10,11,12]

ghci> "hello" ++ " " ++ "world"
"hello world"

ghci> ['w','o'] ++ ['o','t']
"woot"
```

lists

```
ghci> 'A':" SMALL CAT"
"A SMALL CAT"

ghci> 5:[1,2,3,4,5]
[5,1,2,3,4,5]

ghci> "Steve Buscemi" !! 6
'B'

ghci> [9.4,33.2,96.2,11.2,23.25] !! 1
33.2
```

lists

```
ghci> [3,2,1] > [2,1,0]
True
ghci> [3,2,1] > [2,10,100]
True
ghci> [3,4,2] > [3,4]
True
ghci> [3,4,2] > [2,4]
True
ghci> [3,4,2] == [3,4,2]
True
```

```
ghci> head [5,4,3,2,1]
5
ghci> tail [5,4,3,2,1]
[4,3,2,1]
ghci> last [5,4,3,2,1]
1
ghci> init [5,4,3,2,1]
[5,4,3,2]
ghci> head []
*** Exception ...
```

lists

```
ghci> length [5,4,3,2,1]
5
ghci> null [1,2,3]
False
ghci> null []
True
ghci> reverse [5,4,3,2,1]
[1,2,3,4,5]
```

```
ghci> take 3 [5,4,3,2,1]
[5,4,3]
ghci> take 1 [3,9,3]
[3]
ghci> take 5 [1,2]
[1,2]
ghci> take 0 [6,6,6]
[]
```

lists

```
ghci> minimum [8,4,2,1,5,6]
1
ghci> maximum [1,9,2,3,4]
9
ghci> sum [5,2,1,6,3,2,5,7]
31
ghci> product [6,2,1,2]
24
ghci> 4 ‘elem‘ [3,4,5,6]
True
ghci> 10 ‘elem‘ [3,4,5,6]
False
```

ranges

```
ghci> [1..20]
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]

ghci> ['a'..'z']
"abcdefghijklmnopqrstuvwxyz"

ghci> ['K'..'Z']
"KLMNOPQRSTUVWXYZ"
```

ranges

```
ghci> [2,4..20]
[2,4,6,8,10,12,14,16,18,20]

ghci> [3,6..20]
[3,6,9,12,15,18]

ghci> [1,2,4,8,16..100]
<interactive>:1:11: parse error on input '..'

ghci> [0.1, 0.3 .. 1]
[0.1,0.3,0.5,0.7,0.8999999999999999,1.0999999999999999]
```

infinite lists

```
ghci> take 10 (cycle [1,2,3])
[1,2,3,1,2,3,1,2,3,1]

ghci> take 12 (cycle "LOL ")
"LOL LOL LOL "

ghci> take 10 (repeat 5)
[5,5,5,5,5,5,5,5,5,5]

ghci> replicate 10 5
[5,5,5,5,5,5,5,5,5,5]
```

list comprehensions

```
ghci> [x*2 | x <- [1..10]]
[2,4,6,8,10,12,14,16,18,20]

ghci> [x*2 | x <- [1..10], x*2 >= 12]
[12,14,16,18,20]

ghci> [ x | x <- [50..100], x ‘mod‘ 7 == 3]
[52,59,66,73,80,87,94]

ghci> [ x | x <- [10..20], x /= 13, x /= 15, x /= 19]
[10,11,12,14,16,17,18,20]

ghci> [ x*y | x <- [2,5,10], y <- [8,10,11]]
[16,20,22,40,50,55,80,100,110]
```

tuples

```
ghci> [(1,2),(8,11,5)]
<interactive>:1:7:
    Couldn't match expected type `(t, t1)'
           against inferred type `(t2, t3, t4)'
    In the expression: (8, 11, 5)
    In the expression: [(1, 2), (8, 11, 5)]
    In the definition of `it': it = [(1, 2), (8, 11, 5)]

ghci> fst (8,11)
8
ghci> fst ("Wow", False)
"Wow"
ghci> snd (8,11)
11
ghci> snd ("Wow", False)
False
```

tuples

```
ghci> zip [1,2,3,4,5] [5,5,5,5,5]
[(1,5),(2,5),(3,5),(4,5),(5,5)]

ghci> zip [1 .. 5] ["one", "two", "three", "four", "five"]
[(1,"one"),(2,"two"),(3,"three"),(4,"four"),(5,"five")]

ghci> zip [5,3,2,6,2,7,2,5,4,6,6] ["im","a","turtle"]
[(5,"im"),(3,"a"),(2,"turtle")]

ghci> zip [1..] ["apple", "orange", "cherry", "mango"]
[(1,"apple"),(2,"orange"),(3,"cherry"),(4,"mango")]
```

types

```
ghci> :t 'a'
'a' :: Char

ghci> :t True
True :: Bool

ghci> :t "HELLO!"
"HELLO!" :: [Char]

ghci> :t (True, 'a')
(True, 'a') :: (Bool, Char)

ghci> :t 4 == 5
4 == 5 :: Bool
```

integers

```
addThree :: Int -> Int -> Int -> Int
addThree x y z = x + y + z

factorial :: Integer -> Integer
factorial n = product [1..n]

ghci> factorial 50
30414093201713378043612608166064768844377641568960512000000000
```

floating point numbers

```
circumference :: Float -> Float
circumference r = 2 * pi * r

ghci> circumference 4.0
25.132742

circumference' :: Double -> Double
circumference' r = 2 * pi * r

ghci> circumference' 4.0
25.132741228718345
```

type variables

```
ghci> :t head
head :: [a] -> a

ghci> :t fst
fst :: (a, b) -> a
```

overloading

```
ghci> :t (==)
(==) :: (Eq a) => a -> a -> Bool

ghci> 5 == 5
True
ghci> 5 /= 5
False
ghci> 'a' == 'a'
True
ghci> "Ho Ho" == "Ho Ho"
True
ghci> 3.432 == 3.432
True
```

comparison

```
ghci> :t (>)
(>) :: (Ord a) => a -> a -> Bool

ghci> "Abrakadabra" < "Zebra"
True
ghci> "Abrakadabra" `compare` "Zebra"
LT
ghci> 5 >= 2
True
ghci> 5 `compare` 3
GT
```

string conversion

```
ghci> show 3
"3"
ghci> show 5.334
"5.334"
ghci> show True
"True"

ghci> read "True" || False
True
ghci> read "8.2" + 3.8
12.0
ghci> read "5" - 2
3
ghci> read "[1,2,3,4]" ++ [3]
[1,2,3,4,3]
```

string conversion

```
ghci> read "4"
<interactive>:1:0:
    Ambiguous type variable ‘a’ in the constraint:
      ‘Read a’ arising from a use of ‘read’
      at <interactive>:1:0-7
    Probable fix: add a type signature that fixes these
                  type variable(s)

ghci> :t read
read :: (Read a) => String -> a
```

type annotations

```
ghci> read "5" :: Int
5
ghci> read "5" :: Float
5.0
ghci> (read "5" :: Float) * 4
20.0
ghci> read "[1,2,3,4]" :: [Int]
[1,2,3,4]
ghci> read "(3, 'a')" :: (Int, Char)
(3, 'a')
```

overloaded numbers

```
ghci> :t 20
20 :: (Num t) => t

ghci> 20 :: Int
20
ghci> 20 :: Integer
20
ghci> 20 :: Float
20.0
ghci> 20 :: Double
20.0

ghci> :t (*)
(*) :: (Num a) => a -> a -> a
```

number conversion

```
ghci> :t fromIntegral
fromIntegral :: (Integral a, Num b) => a -> b

ghci> :t length
length :: [a] -> Int

ghci> length [1,2,3,4] + 3.2
<interactive>:1:19:
    No instance for (Fractional Int)
      arising from the literal ‘3.2’ at <interactive>:1:19-21
    Possible fix: ...

ghci> fromIntegral (length [1,2,3,4]) + 3.2
7.2
```

pattern matching

```
lucky :: (Integral a) => a -> String
lucky 7 = "LUCKY NUMBER SEVEN!"
lucky x = "Sorry, you're out of luck, pal!"

sayMe :: (Integral a) => a -> String
sayMe 1 = "One!"
sayMe 2 = "Two!"
sayMe 3 = "Three!"
sayMe 4 = "Four!"
sayMe 5 = "Five!"
sayMe x = "Not between 1 and 5"
```

pattern matching and recursion

```
factorial :: (Integral a) => a -> a
factorial 0 = 1
factorial n = n * factorial (n - 1)

factorial 3
3 * factorial 2
3 * (2 * factorial 1)
3 * (2 * (1 * factorial 1))
3 * (2 * (1 * 1))
```

pattern match failure

```
charName :: Char -> String
charName ’a’ = "Albert"
charName ’b’ = "Broseph"
charName ’c’ = "Cecil"

ghci> charName ’a’
"Albert"
ghci> charName ’b’
"Broseph"
ghci> charName ’h’
"*** Exception: tut.hs:(53,0)-(55,21):
Non-exhaustive patterns in function charName
```

tuple patterns

```
addVectors :: (Num a) => (a, a) -> (a, a) -> (a, a)
addVectors a b = (fst a + fst b, snd a + snd b)

addVectors :: (Num a) => (a, a) -> (a, a) -> (a, a)
addVectors (x1, y1) (x2, y2) = (x1 + x2, y1 + y2)

ghci> let xs = [(1,3), (4,3), (2,4), (5,3), (5,6), (3,1)]

ghci> [a+b | (a,b) <- xs]
[4,7,6,8,11,4]
```

list patterns

```
head' :: [a] -> a
head' [] = error "Can't call head on an empty list, dummy!"
head' (x:_) = x

ghci> head' [4,5,6]
4
ghci> head' "Hello"
'H'

tell :: (Show a) => [a] -> String
tell [] = "The list is empty"
tell (x:[]) = "The list has one element: " ++ show x
tell (x:y:[]) = "The list has two elements: "
             ++ show x ++ " and " ++ show y
tell (x:y:_) = "The list is long. The first two elements are: "
            ++ show x ++ " and " ++ show y
```

recursive list functions

```
length' :: (Num b) => [a] -> b
length' [] = 0
length' (_:xs) = 1 + length' xs

sum' :: (Num a) => [a] -> a
sum' [] = 0
sum' (x:xs) = x + sum' xs
```

check out `http://learnyouahaskell.com/recursion` for more!

guards

```
bmiTell :: (RealFloat a) => a -> String
bmiTell bmi
    | bmi <= 18.5 = "You're underweight!"
    | bmi <= 25.0 = "You're supposedly normal!"
    | bmi <= 30.0 = "You're fat!"
    | otherwise   = "You're a whale!"

otherwise :: Bool
otherwise = True
```

guards

```
bmiTell :: (RealFloat a) => a -> a -> String
bmiTell weight height
    | weight / height^2 <= 18.5 = "You're underweight!"
    | weight / height^2 <= 25.0 = "You're supposedly normal!"
    | weight / height^2 <= 30.0 = "You're fat!"
    | otherwise                 = "You're a whale!"

ghci> bmiTell 85 1.90
"You're supposedly normal!"
```

local bindings

```
bmiTell :: (RealFloat a) => a -> a -> String
bmiTell weight height
    | bmi <= 18.5 = "You're underweight!"
    | bmi <= 25.0 = "You're supposedly normal!"
    | bmi <= 30.0 = "You're fat!"
    | otherwise   = "You're a whale!"
    where bmi = weight / height^2
```

pattern bindings

```
bmiTell :: (RealFloat a) => a -> a -> String
bmiTell weight height
    | bmi <= skinny = "You’re underweight!"
    | bmi <= normal = "You’re supposedly normal!"
    | bmi <= fat    = "You’re fat!"
    | otherwise     = "You’re a whale!"
    where bmi = weight / height^2
          (skinny, normal, fat) = (18.5, 25.0, 30.0)
```

local functions

```
calcBmis :: (RealFloat a) => [(a, a)] -> [a]
calcBmis xs = [bmi w h | (w, h) <- xs]
    where bmi weight height = weight / height^2
```

let expressions

```
cylinder :: (RealFloat a) => a -> a -> a
cylinder r h =
    let sideArea = 2 * pi * r * h
        topArea  = pi * r^2
    in  sideArea + 2 * topArea

ghci> 4 * (let a = 9 in a + 1) + 2
42

ghci> [let square x = x*x in (square 5, square 3, square 2)]
[(25,9,4)]

calcBmis :: (RealFloat a) => [(a, a)] -> [a]
calcBmis xs = [bmi | (w, h) <- xs, let bmi = w / h ^ 2]
```

case expressions

```
head’ :: [a] -> a
head’ [] = error "No head for empty lists!"
head’ (x:_) = x

head’ :: [a] -> a
head’ xs = case xs of [] -> error "No head for empty lists!"
                      (x:_) -> x

case expression of pattern -> result
                   pattern -> result
                   pattern -> result
                   ...
```

case expressions

```
describeList :: [a] -> String
describeList xs = "The list is "
               ++ case xs of
                    [] -> "empty."
                    [x] -> "a singleton list."
                    xs -> "a longer list."

describeList :: [a] -> String
describeList xs = "The list is " ++ what xs
    where what [] = "empty."
          what [x] = "a singleton list."
          what xs = "a longer list."
```

curried functions

```
ghci> max 4 5
5
ghci> (max 4) 5
5

max :: (Ord a) => a -> a -> a
max :: (Ord a) => a -> (a -> a)
```

partial application

```
multThree :: (Num a) => a -> a -> a -> a
multThree x y z = x * y * z

multThree             :: (Num a) => a -> (a -> (a -> a))
multThree 3           :: (Num a) => a -> (a -> a)
(multThree 3) 5       :: (Num a) => a -> a
((multThree 3) 5) 9 :: (Num a) => a
```

partial application

```
ghci> let multTwoWithNine = multThree 9
ghci> multTwoWithNine 2 3
54
ghci> let multWithEighteen = multTwoWithNine 2
ghci> multWithEighteen 10
180
```

partial application

```
compareWithHundred :: (Num a, Ord a) => a -> Ordering
compareWithHundred x = compare 100 x

compareWithHundred :: (Num a, Ord a) => a -> Ordering
compareWithHundred = compare 100

ghci> :t compare 100
compare 100 :: (Num t, Ord t) => t -> Ordering

ghci> compare 100
<interactive>:1:0:
    No instance for (Show (t -> Ordering))
      arising from a use of ‘print’ at <interactive>:1:0-10
    Possible fix:
      add an instance declaration for (Show (t -> Ordering))
    In a stmt of a ’do’ expression: print it
```

infix sections

```
divideByTen :: (Floating a) => a -> a
divideByTen = (/10)

isUpperAlphanum :: Char -> Bool
isUpperAlphanum = (`elem` ['A'..'Z'])

ghci> :t (-4)
(-4) :: (Num a) => a
ghci> :t subtract 4
subtract 4 :: (Num t) => t -> t
ghci> subtract 4 7
3
ghci> (-) 4 7
-3
```

overview, again

past week

interactive evaluator, expressions, lists, functions, recursion, types, polymorphism, overloading, pattern matching, local bindings, currying

today

first-class functions: `map`, `filter`, `fold`, ..., application, composition, datatypes, records, class declarations, input/output, `do`-notation, files

first-class functions

higher-order functions:

- take other functions as arguments or
- return other functions as results

first-class functions

higher-order functions:

- take other functions as arguments or
- return other functions as results

```
applyTwice :: (a -> a) -> a -> a
applyTwice f x = f (f x)
```

first-class functions

```
ghci> applyTwice (+3) 10
16

ghci> applyTwice (++ " HAHA") "HEY"
"HEY HAHA HAHA"

ghci> applyTwice ("HAHA " ++) "HEY"
"HAHA HAHA HEY"

ghci> applyTwice (multThree 2 2) 9
144

ghci> applyTwice (3:) [1]
[3,3,1]
```

zipWith

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith _ [] _ = []
zipWith _ _ [] = []
zipWith f (x:xs) (y:ys) = f x y : zipWith f xs ys
```

zipWith

```
ghci> zipWith (+) [4,2,5,6] [2,6,2,3]
[6,8,7,9]

ghci> zipWith max [6,3,2,1] [7,3,1,5]
[7,3,2,5]

ghci> zipWith (*) (replicate 5 2) [1..]
[2,4,6,8,10]

ghci> zipWith (zipWith (*)) [[1,2,3],[3,5,6]] [[3,2,2],[3,4,5]]
[[3,4,6],[9,20,30]]
```

flip

```
flip :: (a -> b -> c) -> (b -> a -> c)
flip f = g
    where g x y = f y x
```

flip

```
flip :: (a -> b -> c) -> (b -> a -> c)
flip f = g
    where g x y = f y x
```

```
flip :: (a -> b -> c) -> b -> a -> c
flip f y x = f x y
```

flip

```
flip :: (a -> b -> c) -> (b -> a -> c)
flip f = g
    where g x y = f y x
```

```
flip :: (a -> b -> c) -> b -> a -> c
flip f y x = f x y
```

```
ghci> flip zip [1,2,3,4,5] "hello"
[('h',1),('e',2),('l',3),('l',4),('o',5)]

ghci> zipWith (flip div) [2,2..] [10,8,6,4,2]
[5,4,3,2,1]
```

map

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

map

```
ghci> map (+3) [1,5,3,1,6]
[4,8,6,4,9]

ghci> map (++ "!") ["BIFF", "BANG", "POW"]
["BIFF!","BANG!","POW!"]

ghci> map (replicate 3) [3..6]
[[3,3,3],[4,4,4],[5,5,5],[6,6,6]]

ghci> map (map (^2)) [[1,2],[3,4,5,6],[7,8]]
[[1,4],[9,16,25,36],[49,64]]

ghci> map fst [(1,2),(3,5),(6,3),(2,6),(2,5)]
[1,3,6,2,2]
```

filter

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

filter

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

```
ghci> filter (>3) [1,5,3,2,1,6,4,3,2,1]
[5,6,4]

ghci> filter (==3) [1,2,3,4,5]
[3]

ghci> filter even [1..10]
[2,4,6,8,10]
```

list comprehensions

```
map (+3) [1,5,3,1,6]
 == [ x+3 | x <- [1,5,3,1,6]]
```

list comprehensions

```
map (+3) [1,5,3,1,6]
 == [ x+3 | x <- [1,5,3,1,6]]
```

```
filter (>3) [1,2,3,4,5]
 == [ x | x <- [1,2,3,4,5], x > 3]
```

find the largest number under 100000 that's divisible by 3829

```
largestDivisible :: (Integral a) => a
largestDivisible = head (filter p [100000,99999..])
    where p x = x ‘mod‘ 3829 == 0
```

takeWhile

```
takeWhile :: (a -> Bool) -> [a] -> [a]
takeWhile _ [] = []
takeWhile p (x:xs)
    | p x       = x : takeWhile p xs
    | otherwise = []
```

takeWhile

```
takeWhile :: (a -> Bool) -> [a] -> [a]
takeWhile _ [] = []
takeWhile p (x:xs)
    | p x       = x : takeWhile p xs
    | otherwise = []
```

```
ghci> takeWhile (/=' ') "elephants know how to party"
"elephants"
```

find the sum of all odd squares that are smaller than 10000

```
ghci> sum (takeWhile (<10000) (filter odd (map (^2) [1..])))
166650
```

anonymous functions

```
map (+3) [1,6,3,2]  ==  map (\x -> x + 3) [1,6,3,2]

ghci> zipWith (\a b -> (a*30 + 3)/b) [5,4,3,2,1] [1,2,3,4,5]
[153.0,61.5,31.0,15.75,6.6]

ghci> map (\ (a,b) -> a + b) [(1,2),(3,5),(6,3),(2,6),(2,5)]
[3,8,9,8,7]
```

currying, again

```
addThree :: (Num a) => a -> a -> a -> a
addThree x y z = x + y + z

addThree :: (Num a) => a -> a -> a -> a
addThree = \x -> \y -> \z -> x + y + z
```

currying, again

```haskell
addThree :: (Num a) => a -> a -> a -> a
addThree x y z = x + y + z

addThree :: (Num a) => a -> a -> a -> a
addThree = \x -> \y -> \z -> x + y + z
```

```haskell
flip :: (a -> b -> c) -> b -> a -> c
flip f x y = f y x

flip :: (a -> b -> c) -> b -> a -> c
flip f = \x y -> f y x
```

folds

- many recursive list functions have similar structure
- one case for empty list
- one case for non-empty list, recursive call on tail
- folds reduce lists to a single value
- take starting value (accumulator) and binary function

left fold

```
sum’ :: (Num a) => [a] -> a
sum’ xs = foldl (\acc x -> acc + x) 0 xs

ghci> sum’ [3,5,2,1]
11
```

left fold

```
sum’ :: (Num a) => [a] -> a
sum’ xs = foldl (\acc x -> acc + x) 0 xs

ghci> sum’ [3,5,2,1]
11
```

```
0 + 3
   [3,5,2,1]
3 + 5
   [5,2,1]
8 + 2
   [2,1]
10+ 1
   [1]
11
```

left fold

```
sum' :: (Num a) => [a] -> a
sum' xs = foldl (+) 0 xs

ghci> sum' [3,5,2,1]
11
```

```
0 + 3
   [3,5,2,1]
3 + 5
   [5,2,1]
8 + 2
   [2,1]
10+ 1
   [1]
11
```

right fold

- binary operation takes accumulator as right argument
- generally, result of `foldl` or `foldr` can be of any type

right fold

- binary operation takes accumulator as right argument
- generally, result of `foldl` or `foldr` can be of any type

```
map’ :: (a -> b) -> [a] -> [b]
map’ f xs = foldr (\x acc -> f x : acc) [] xs
```

right fold

- binary operation takes accumulator as right argument
- generally, result of `foldl` or `foldr` can be of any type

```
map’ :: (a -> b) -> [a] -> [b]
map’ f xs = foldr (\x acc -> f x : acc) [] xs
```

```
map’ f xs = foldl (\acc x -> acc ++ [f x]) [] xs
```

folds

- right folds work on infinite lists, left folds don't!
- implement functions that traverse lists once
- element by element
- there are variants `foldl1` and `foldr1` for non-empty lists

folds

- right folds work on infinite lists, left folds don’t!
- implement functions that traverse lists once
- element by element
- there are variants `foldl1` and `foldr1` for non-empty lists

```
foldr f x [1,2,3,4]  ==  f 1 (f 2 (f 3 (f 4 x)))

foldl f x [1,2,3,4]  ==  f (f (f (f x 1) 2) 3) 4
```

example folds

```
filter’ :: (a -> Bool) -> [a] -> [a]
filter’ p = foldr (\x acc -> if p x then x : acc else acc) []

reverse’ :: [a] -> [a]
reverse’ = foldl (\acc x -> x : acc) []

head’ :: [a] -> a
head’ = foldr1 (\x _ -> x)

last’ :: [a] -> a
last’ = foldl1 (\_ x -> x)
```

example folds

```
filter' :: (a -> Bool) -> [a] -> [a]
filter' p = foldr (\x acc -> if p x then x : acc else acc) []

reverse' :: [a] -> [a]
reverse' = foldl (flip (:)) []

head' :: [a] -> a
head' = foldr1 (\x _ -> x)

last' :: [a] -> a
last' = foldl1 (\_ x -> x)
```

application

```
($) :: (a -> b) -> a -> b
f $ x = f x
```

application

```
($) :: (a -> b) -> a -> b
f $ x = f x
```

- usual application has high precedence
- `($)` has low precedence
- usual application is left associative
- `($)` is right associative

fewer parenthesis

```
sum (map sqrt [1..130])        sum $ map sqrt [1..130]

sqrt (3 + 4 + 9)               sqrt $ 3 + 4 + 9
```

fewer parenthesis

```
sum (map sqrt [1..130])        sum $ map sqrt [1..130]

sqrt (3 + 4 + 9)               sqrt $ 3 + 4 + 9


sum (filter (> 10) (map (*2) [2..10]))

sum $ filter (> 10) $ map (*2) [2..10]
```

fewer parenthesis

```
sum (map sqrt [1..130])        sum $ map sqrt [1..130]

sqrt (3 + 4 + 9)               sqrt $ 3 + 4 + 9


sum (filter (> 10) (map (*2) [2..10]))

sum $ filter (> 10) $ map (*2) [2..10]
```

not only for reducing parenthesis: application as a function

```
ghci> map ($ 3) [(4+), (10*), (^2), sqrt]
[7.0,30.0,9.0,1.7320508075688772]
```

composition

```
(.) :: (b -> c) -> (a -> b) -> a -> c
f . g = \x -> f (g x)
```

composition

```
(.) :: (b -> c) -> (a -> b) -> a -> c
f . g = \x -> f (g x)
```

like math: $(f \circ g)(x) = f(g(x))$

fewer lambdas

```
ghci> map (\x -> negate (abs x)) [5,-3,-6,7,-3,2,-19,24]
[-5,-3,-6,-7,-3,-2,-19,-24]

ghci> map (negate . abs) [5,-3,-6,7,-3,2,-19,24]
[-5,-3,-6,-7,-3,-2,-19,-24]
```

fewer lambdas

```
ghci> map (\x -> negate (abs x)) [5,-3,-6,7,-3,2,-19,24]
[-5,-3,-6,-7,-3,-2,-19,-24]

ghci> map (negate . abs) [5,-3,-6,7,-3,2,-19,24]
[-5,-3,-6,-7,-3,-2,-19,-24]


ghci> map (\xs -> negate(sum(tail xs))) [[1..5],[3..6],[1..7]]
[-14,-15,-27]

ghci> map (negate . sum . tail) [[1..5],[3..6],[1..7]]
[-14,-15,-27]
```

fewer lambdas

```
ghci> map (\x -> negate (abs x)) [5,-3,-6,7,-3,2,-19,24]
[-5,-3,-6,-7,-3,-2,-19,-24]

ghci> map (negate . abs) [5,-3,-6,7,-3,2,-19,24]
[-5,-3,-6,-7,-3,-2,-19,-24]
```

```
ghci> map (\xs -> negate(sum(tail xs))) [[1..5],[3..6],[1..7]]
[-14,-15,-27]

ghci> map (negate . sum . tail) [[1..5],[3..6],[1..7]]
[-14,-15,-27]
```

use `(.)` instead of `($)` in chains:

```
replicate 100 . product . map (*3) . zipWith max [1,3] $ [4,2]
```

point-free style

```
sum’ xs = foldl (+) 0 xs

sum’ = foldl (+) 0
```

point-free style

```
sum’ xs = foldl (+) 0 xs

sum’ = foldl (+) 0
```

```
f x = ceiling (negate (tan (cos (max 50 x))))

f = ceiling . negate . tan . cos . max 50
```

think about readers!

```
oddSquareSum :: Integer
oddSquareSum =
    sum (takeWhile (<10000) (filter odd (map (^2) [1..])))
```

think about readers!

```
oddSquareSum :: Integer
oddSquareSum =
    sum (takeWhile (<10000) (filter odd (map (^2) [1..])))
```

```
oddSquareSum :: Integer
oddSquareSum =
    sum . takeWhile (<10000) . filter odd . map (^2) $ [1..]
```

think about readers!

```
oddSquareSum :: Integer
oddSquareSum =
    sum (takeWhile (<10000) (filter odd (map (^2) [1..])))
```

```
oddSquareSum :: Integer
oddSquareSum =
    sum . takeWhile (<10000) . filter odd . map (^2) $ [1..]
```

```
oddSquareSum :: Integer
oddSquareSum =
    let oddSquares = filter odd $ map (^2) [1..]
        belowLimit = takeWhile (<10000) oddSquares
     in sum belowLimit
```

data types

```
data Bool = False | True
```

data types

```
data Bool = False | True
```

```
ghci> :t False
False :: Bool
ghci> :t True
True :: Bool
```

data types

```
data Shape = Circle Float Float Float
           | Rectangle Float Float Float Float
```

data types

```
data Shape = Circle Float Float Float
           | Rectangle Float Float Float Float
```

```
ghci> :t Circle
Circle :: Float -> Float -> Float -> Shape
ghci> :t Rectangle
Rectangle :: Float -> Float -> Float -> Float -> Shape
```

pattern matching

```
surface :: Shape -> Float
surface (Circle _ _ r) = pi * r ^ 2
surface (Rectangle x1 y1 x2 y2) = (abs $ x2-x1) * (abs $ y2-y1)
```

pattern matching

```
surface :: Shape -> Float
surface (Circle _ _ r) = pi * r ^ 2
surface (Rectangle x1 y1 x2 y2) = (abs $ x2-x1) * (abs $ y2-y1)
```

```
ghci> surface $ Circle 10 20 10
314.15927

ghci> surface $ Rectangle 0 0 100 100
10000.0
```

derived `Show` instance

```
data Shape = Circle Float Float Float
           | Rectangle Float Float Float Float
           deriving (Show)
```

derived `Show` instance

```
data Shape = Circle Float Float Float
           | Rectangle Float Float Float Float
           deriving (Show)
```

```
ghci> Circle 10 20 5
Circle 10.0 20.0 5.0

ghci> Rectangle 50 230 60 90
Rectangle 50.0 230.0 60.0 90.0

ghci> map (Circle 1 2) [4,5,6]
[Circle 1.0 2.0 4.0,Circle 1.0 2.0 5.0,Circle 1.0 2.0 6.0]
```

record syntax

```
data Car = Car String String Int deriving (Show)
```

record syntax

```
data Car = Car String String Int deriving (Show)
```

```
data Car = Car {company::String, model::String, year::Int}
           deriving (Show)

ghci> Car "Ford" "Mustang" 1967
Car "Ford" "Mustang" 1967

ghci> Car {company="Ford", year=1967, model="Mustang"}
Car {company = "Ford", model = "Mustang", year = 1967}
```

record selectors

```
ghci> :t company
company :: Car -> String

ghci> :t model
model :: Car -> String

ghci> :t year
year :: Car -> Int
```

type parameters

```
data Maybe a = Nothing | Just a
```

type parameters

```
data Maybe a = Nothing | Just a
```

```
ghci> Just "Haha"
Just "Haha"

ghci> Just 84
Just 84

ghci> :t Just "Haha"
Just "Haha" :: Maybe [Char]

ghci> :t Nothing
Nothing :: Maybe a
```

class constraints

```
data Vector a = Vector a a a deriving (Show)

plus :: (Num t) => Vector t -> Vector t -> Vector t
(Vector i j k) `plus` (Vector l m n) = Vector (i+l) (j+m) (k+n)

mult :: (Num t) => Vector t -> t -> Vector t
(Vector i j k) `mult` m = Vector (i*m) (j*m) (k*m)

scalarMult :: (Num t) => Vector t -> Vector t -> t
(Vector i j k) `scalarMult` (Vector l m n) = i*l + j*m + k*n
```

type synonyms

```
type String = [Char]
```

type synonyms

```
type String = [Char]
```

```
phoneBook :: [(String,String)]
phoneBook =
    [("betty","555-2938")
    ,("bonnie","452-2928")
    ,("patsy","493-2928")
    ,("lucille","205-2928")
    ,("wendy","939-8282")
    ,("penny","853-2492")
    ]
```

type synonyms

```
type PhoneBook = [(String,String)]

type PhoneNumber = String
type Name = String
type PhoneBook = [(Name,PhoneNumber)]

inPhoneBook :: Name -> PhoneNumber -> PhoneBook -> Bool
inPhoneBook name pnumber pbook = (name,pnumber) `elem` pbook
```

recursive data structures

```
data List a = Empty | Cons a (List a)
  deriving (Show, Read, Eq, Ord)
```

recursive data structures

```
data List a = Empty | Cons a (List a)
  deriving (Show, Read, Eq, Ord)
```

```
ghci> Empty
Empty

ghci> 5 `Cons` Empty
Cons 5 Empty

ghci> 4 `Cons` (5 `Cons` Empty)
Cons 4 (Cons 5 Empty)

ghci> 3 `Cons` (4 `Cons` (5 `Cons` Empty))
Cons 3 (Cons 4 (Cons 5 Empty))
```

recursion

```
infixr 5  .++
(.++) :: List a -> List a -> List a
Empty      .++ ys = ys
Cons x xs .++ ys = Cons x (xs .++ ys)
```

type-class declaration

```
class Eq a where
    (==) :: a -> a -> Bool
    (/=) :: a -> a -> Bool
    x == y = not (x /= y)
    x /= y = not (x == y)
```

type-class instances

```
data TrafficLight = Red | Yellow | Green

instance Eq TrafficLight where
    Red    == Red    = True
    Green  == Green  = True
    Yellow == Yellow = True
    _      == _      = False
```

type-class instances

```
data TrafficLight = Red | Yellow | Green

instance Eq TrafficLight where
    Red    == Red    = True
    Green  == Green  = True
    Yellow == Yellow = True
    _      == _      = False



instance Show TrafficLight where
    show Red    = "Red light"
    show Yellow = "Yellow light"
    show Green  = "Green light"
```

type-class instances

```
ghci> Red == Red
True

ghci> Red == Yellow
False

ghci> Red ‘elem‘ [Red, Yellow, Green]
True

ghci> [Red, Yellow, Green]
[Red light,Yellow light,Green light]
```

type-class instances

```
instance (Eq m) => Eq (Maybe m) where
    Just x  == Just y  = x == y
    Nothing == Nothing = True
    _       == _       = False
```

input/output

- in conflict with *purely* functional programming
  - describes what things are
  - no steps to execute
  - output depends only on input
  - no side effects
- how to print result? or read input?
- separate pure ind impure parts of a program using types!

hello, world

```
main = putStrLn "hello, world"
```

hello, world

```
main = putStrLn "hello, world"
```

```
$ ghc --make helloworld
[1 of 1] Compiling Main      ( helloworld.hs, helloworld.o )
Linking helloworld ...
```

hello, world

```haskell
main = putStrLn "hello, world"
```

```
$ ghc --make helloworld
[1 of 1] Compiling Main          ( helloworld.hs, helloworld.o )
Linking helloworld ...
```

```
$ ./helloworld
hello, world
```

IO actions

```
ghci> :t putStrLn
putStrLn :: String -> IO ()

ghci> :t putStrLn "hello, world"
putStrLn "hello, world" :: IO ()
```

IO actions

```
ghci> :t putStrLn
putStrLn :: String -> IO ()

ghci> :t putStrLn "hello, world"
putStrLn "hello, world" :: IO ()
```

- `putStrLn` takes a string and returns an *IO action*
- the result of this action has type `()` (unit)
- IO actions (may) perform side effects when executed
- they are executed when they become part of `main`

do notation

```haskell
main = do
    putStrLn "Hello, what’s your name?"
    name <- getLine
    putStrLn ("Hey " ++ name ++ ", you rock!")
```

binding IO results

```
ghci> :t getLine
getLine :: IO String
```

binding IO results

```
ghci> :t getLine
getLine :: IO String
```

- `name <- getLine` performs IO action `getLine`
- additionally, binds result to variable `name :: String`
- `getLine` action is impure (different results)
- `<-` pulls results out of IO

mixing pure and impure code

```
main = do
    putStrLn "Hello, what’s your name?"
    name <- getLine
    putStrLn $ "This is your future: " ++ tellFortune name
```

mixing pure and impure code

```
main = do
    putStrLn "Hello, what’s your name?"
    name <- getLine
    putStrLn $ "This is your future: " ++ tellFortune name
```

- `tellFortune :: String -> String`
- use *thin* IO wrapper that calls pure functions

let bindings

```
import Data.Char

main = do
    putStrLn "What’s your first name?"
    firstName <- getLine
    putStrLn "What’s your last name?"
    lastName <- getLine
    let bigFirst = map toUpper firstName
        bigLast  = map toUpper lastName
    putStrLn $ "Hi " ++ bigFirst ++ " " ++ bigLast ++ "!"
```

let bindings

```
import Data.Char

main = do
    putStrLn "What’s your first name?"
    firstName <- getLine
    putStrLn "What’s your last name?"
    lastName <- getLine
    let bigFirst = map toUpper firstName
        bigLast  = map toUpper lastName
    putStrLn $ "Hi " ++ bigFirst ++ " " ++ bigLast ++ "!"
```

- `<-` binds results of IO actions
- `let` binds results of pure computations

interactive loop

```
main = do
    line <- getLine
    if null line
        then return ()
        else do
            putStrLn $ reverseWords line
            main

reverseWords :: String -> String
reverseWords = unwords . map reverse . words
```

`return`

- does *not* interrupt execution
- creates IO action without side effect

return

- does *not* interrupt execution
- creates IO action without side effect

```
main = do
    return ()
    return "HAHAHA"
    line <- getLine
    return "BLAH BLAH BLAH"
    return 4
    putStrLn line
```

return

```
main = do
    a <- return "hell"
    b <- return "yeah!"
    putStrLn $ a ++ " " ++ b
```

return

```
main = do
    a <- return "hell"
    b <- return "yeah!"
    putStrLn $ a ++ " " ++ b
```

```
main = do
    let a = "hell"
        b = "yeah!"
    putStrLn $ a ++ " " ++ b
```

some IO functions

```
getChar :: IO Char

putChar :: Char -> IO ()

putStr :: String -> IO ()

print :: Show a => a -> IO ()
```

files

```
type FilePath = String

readFile  :: FilePath -> IO String
writeFile :: FilePath -> String -> IO ()
```

files

```
type FilePath = String

readFile  :: FilePath -> IO String
writeFile :: FilePath -> String -> IO ()
```

```
import System.IO
import Data.Char

main = do
    contents <- readFile "readme.txt"
    writeFile "README.txt" (map toUpper contents)
```

summary

past week

interactive evaluator, expressions, lists, functions, recursion, types, polymorphism, overloading, pattern matching, local bindings, currying

today

first-class functions: `map`, `filter`, `fold`, ..., application, composition, datatypes, records, class declarations, input/output, `do`-notation, files

Go, and learn you a Haskell!
`http://learnyouahaskell.com`