Haskell

- Pure, functional, lazy language, statically typed, parametric polymorphic types, type classes
- Pure - there are no side effects in functions, expressions
  - I/O is an exception
  - No Assignment operator, +=, ++
  - Promotes referential transparency
    - Substituting equals for equals: \( p(x) = x^2 + y \), \( p(3) / p(z) = (3^2 + y) / (z^2+y) \)
    - Once you give a variable \( x \) a value, it has that value as long as it's alive.
- Functional
  - higher-order functions - functions as parameters, functions as results
  - operations on functions - composition, .......
    - integral of \( f(x) \) dx from a to b --- integral is a higher-order function
  - Use recursion instead of iteration
- Lazy - don't do an operation unless you need the result.
  - Look at if (e) then \( e_1 \) else \( e_2 \) conditional expression
    - \( (e ? e_1 : e_2) \) in C
    - Not like a function cond(x,y,z) { if x == true then return y else return z } because cond(expr1, expr2, expr3) evaluates all the expressions before running the body of cond.
    - cond function uses eager evaluation.
  - Can speed things up (don't evaluate something you wind up not needing).
  - Can slow things down? (What if we want to repeatedly evaluate an expression?)
  - Makes figuring out resource use harder, generally speaking
- Types -- we'll discuss these later
  - statically typed, parametric polymorphic types, type classes
- Pure functional languages are often thought of as being more concise and "elegant"

Running Haskell

- Haskell platform - Glasgow Haskell Compiler (GHC)
- An interactive version lets you do read/eval/print loops

Sample Interactive Run of Haskell

unix > ghci
GHCi, version 8.4.3: http://www.haskell.org/ghc/ :? for help
Prelude> 3 + 5  -- typical read/eval/print loop
8
Prelude> sqrt 4 -- function application doesn't require parens
2.0
Prelude> sqrt (4 + 4) -- fcn appl higher precedence than +
2.8284271247461903
Prelude> sqrt 4 + 4
6.0
Prelude> 2 == 1+1 -- equality is ==
True
Prelude> 2 /= 3-4 -- but inequality is /=
True
Prelude> 2 == 2 && 4 == 2+2 -- logical and is &&
True
Prelude> 2 == 3 || 4 == 2+2 -- logical or is ||
True
Prelude> if 2 > 3 then 1 else 5 -- conditional expression is like (2 > 3 ? 1 : 5 in C)
5
Prelude> [1,2,3,4] -- List of 4 elements
[1,2,3,4]
Prelude> head [1,2,3,4] -- First element of list
1
Prelude> tail [1,2,3,4] -- Non-first elements of list
[2,3,4]
Prelude> [] -- empty list
[]
Prelude> 1 : [2,3,4] -- prepend ("cons") value as head using :
[1,2,3,4]
Prelude> 1 : 2 : 3 : 4 : []
[1,2,3,4]
Prelude> [1,2] ++ [3,4,5] -- concatenate lists using infix ++
[1,2,3,4,5]
Prelude> 'a' -- characters are surrounded by apostrophes
'a'
Prelude> "abc" -- strings are surrounded by double quotes
"abc"
Prelude> "abc" == ['a', 'b', 'c'] -- Strings are just lists of characters
True
[1,2,3,4,5]
Prelude> "abc" ++ "def" -- strings are lists of chars, so ++ is string
concatenation
"abcdef"

Prelude> -- let expression to define a variable having a value
Prelude> let x = 3 in x + 2
```haskell
Prelude> let x = 3 in [x+2, x*x]
[5,9]

Prelude> -- interpreter assumes whole expression is on one line, so second
line gives error
Prelude> let x = 3
Prelude> in x + 2

<interactive>:34:1: error: parse error on input ‘in’

Prelude> -- For multi-line input in interpreter, add line with :{, then your
stuff, then line with :}
Prelude> :{
  Prelude| let x = 3
  Prelude|   in x + 2
  Prelude| :}
5

Prelude> f x = x + 2 -- function definition
Prelude> f 3
5

Prelude> f (f 3) -- function application is right associative
7

Prelude> g x y = x * y + y + f(x * y) -- function of two arguments
Prelude> g 2 3
17

Prelude> -- comparisons work on chars, strings, ..., lists of comparable
things
Prelude> 'a' <'b'
True
Prelude> "aa" < "ab"
True
Prelude> "a" < "ab"
True
Prelude> ['a', 'a'] < ['a','b']
True
Prelude> [1,2] < [1,5]
True
Prelude> [2,3] < [6]
True
Prelude> [ [1,2], [2,3] ] < [ [1,5], [6] ] -- list of comparable items (i.e.
lists of numbers)
True

-- functions are not comparable: error message is saying that functions of
type Double -> Double
```
-- aren't comparable (the Double -> Double type is not in the collection Eq of equality-comparable types)
Prelude> sqrt == sqrt

<interactive>:71:1: error:
  • No instance for (Eq (Double -> Double))
    arising from a use of ‘==’
    (maybe you haven’t applied a function to enough arguments?)
  • In the expression: sqrt == sqrt
    In an equation for ‘it’: it = sqrt == sqrt

Prelude> [1..5] -- list range, 1 upward to 5
[1,2,3,4,5]
Prelude> [1, 3..10] -- list range: delta between elements is 3 - 1
[1,3,5,7,9]
Prelude> [10, 9..0] -- need to be explicit for downward-going lists
[10,9,8,7,6,5,4,3,2,1,0]
Prelude> [10..0] -- empty list because 10, 11, ... are all > 0 so we don't include them
[]

Prelude> -- list comprehensions, similar to set comprehensions
Prelude> [[x*x] | x <- [1..5]] -- list of squares of 1 through 5
[[1],[4],[9],[16],[25]]
Prelude> [x*x | x <- [1..5]]
[1,4,9,16,25]
Prelude> [ x*x | x <- [1..100], x*x < 64] -- use filter to get squares of 1 to 100 stopping once a square is >= 64
[1,4,9,16,25,36,49]
Prelude> [ x+y | x <- [1..5], y <- [3..9]] -- can choose from multiple lists
[4,5,6,7,8,9,10,5,6,7,8,9,10,11,6,7,8,9,10,11,12,7,8,9,10,11,12,13,8,9,10,11,12,13,14]

-- next time: tuples, operations on lists, infinite lists