Introductory Lecture

COMP0002 Haskell

But what is Haskell?

Haskell

- A Functional Language
- A pure, typed, lazy, functional language
- Fun if you like functions (some don't . . .)
  - write a function
  - apply it to some data (input)
  - function transforms data (output)

input data

output data

C, Java: state update

initial state: (x, 1), (y,2), (z, 3)

z=y;

(x, 1), (y,2), (z,2)

swap program
swaps variable values
by succession of state
updates

y=x;

(x, 1), (y, 1), (z,2)

x=z;

final state: (x,2), (y, 1), (z,2)

Haskell: data transformation

```
swap (1,2) = (2,1)
```

```
swap: (Int,Int) -> (Int,Int)
swap (x,y) = (y,x)
```

Notice: no side effects

= No state changes

= easy decomposition into parallelism

See Scala, F#, etc.

(parallelism + side effects = problems)

Course Book

- This course will follow this book:
- **Learn You a Haskell for Great Good** by Miran Lipovac
- Cover the first six chapters and some other things
- You can read it online (buy it if you love it)

GHCi

- Use the interactive Glasgow Haskell Compiler (GHCi) in this course
- GHCi has both an interpreter and a compiler
  - write simple things at the command line
  - compile larger programs
  - mix these up

Install GHCi on your own machine

- command line interface - runs in a terminal
- Mac OS X, Windows, Linux
- comes with a package of libraries and tools
- http://hackage.haskell.org/platform/

Using the Interpreter

Open a terminal window

type **ghci**

the prompt will become

Prelude>

Now you are ready to begin

arithmetic at the prompt

Type in an arithmetic expression and hit the return key: GHCi prints the evaluation on the next line

```
Prelude> 49*100
4900

Prelude> 67 + 84

151
```

```
Prelude> 132 - 78
54

Prelude> 7 / 4

1.75
```

Brackets can overcome operator precedence

```
Prelude> (132 - 78) * 3
162
```

```
Prelude> 132 - 78 * 3

-102
```

Brackets around negative number constants

```
Prelude> 7 * (-4)

-28

Prelude> 5 + 4.0

9.0
```

Boolean Algebra

```
Prelude> True && False

False

Prelude> True || False

True


Prelude> not False

True
```

Equality Tests

```
Prelude> 21 == 21

True

Prelude> 1 == 0

False
```

Use with any two items of the same type

```
Prelude> “Fred” == “Fred”

True
```

```
Prelude> “Fred” == 56

error

Prelude> 5 /= 7

True

Prelude> 82 /= 82

False
```

Functions

infix functions, e.g.

```
Prelude> 7 + 2
9
```

prefix functions, e.g.

```
Prelude> succ 4
5
```

succ works with any type that has a well defined ordering

```
Prelude> succ ‘w‘
‘x’

Prelude> succ 60 *3
183

Prelude> succ (60 * 3)
181
```

two parameter prefix functions can be used as infix functions

```
Prelude> div 92 7
13
```

```
Prelude> 92 `div` 7
13
```

Functions and Files

open text editor (I use vi) and write a file called **baby.hs** by writing the following

```
doubleMe x = x + x
```

and then saving the file in the working directory

load (compile and make available) by typing

```
Prelude> :l baby
```

then you can use it at the command line

```
Prelude> doubleMe 20
40

Prelude> doubleMe 20.7
41.4
```

Can add more functions into baby.hs - order does not matter

open baby.hs in editor and append the function

```
doubleUs x y = x * 2 + y * 2
```

```
Prelude> doubleUs 6 13
38

Prelude> doubleUs 7.92 8.1
32.04
```

functions can call functions; could define doubleUs as

```
doubleUs x y = doubleMe x +
doubleMe y
```

can use **if** in function definitions

```
doubleSmallNum x = if x > 100
                   then x
                   else x * 2
```

```
doubleSmallNum' x =
    (if x>1 then x else x*2) +1
```

definitions/names

```
davidO'Leary = "It's me, mate!"
```

No initial capital on functions

Lists

Haskell lists are homogeneous data structures

all the elements of the list must be the same type

a list is surrounded by square brackets and the elements are separated by commas

Use `let` as keyword to define a name in GHCi

Entering `let a = 1` in GHCi is the same as writing a script containing `a = 1` and then loading it using `:l`

```
Prelude> let lostNums =[4,78,5,900]

Prelude> lostNums
[4,78,5,900]
```

Concatenation

```
Prelude> [1,2,3,4] ++ [7,8,9]
[1,2,3,4,7,8,9]

Prelude> “Hello” ++ “ “ ++ “World!”
“Hello World!”

Prelude> [‘H’,’e’,’l’] ++ [‘l’,’o’]
“Hello”
```

In Haskell strings are lists of characters and we can use list functions on strings

The cons operator, :

cons adds an element to the head of a list

```
Prelude> ‘A’ : “ BIG MESS”
“A BIG MESS”

Prelude> 54 : [45,69,27]
[54,45,69,27]
```

++ takes a pair of lists of the same type
and : takes an element and then a list of
that type of element

prepending -

```
Prelude> 5 : [6,34,23]
[5,6,34,23]
```

appending -

```
Prelude> [6,34,23] ++ [5]
[6,34,23,5]
```

Accessing list elements

use the ! ! operator to get a list member by index number; indices start from 0

```
Prelude> “Claude Shannon” !! 10
‘n’

Prelude> [3.4,7.89,9.4,12.0] !! 3
12.0
```

error messages for out of range accesses

Lists inside lists

```
Prelude> let z = [[1,2,3,4],[5,3,3,3],[1,2,2,2,3,4], [1,2,3]]

Prelude> z
[[1,2,3,4],[5,3,3,3],[1,2,2,2,3,4], [1,2,3]]

Prelude> z ++ [[99]]
[[1,2,3,4],[5,3,3,3],[1,2,2,2,3,4], [1,2,3], [99]]
```

```
Prelude> [6,6] : z
[[6,6],[1,2,3,4],[5,3,3,3],[1,2,2,2,3,4], [1,2,3]]

Prelude> z !! 3
[1,2,3]
```

Comparing Lists

Lists can be compared if the items contained in the lists can be compared

compared in lexicographical order using
<, <=, >, >=

Prelude> [5,3,7] > [4,0,0]
True

Prelude> [7,90,45,5] > [7,67,5,6]
True

Prelude> [7,90,45,5] > [7,90,45,6]
False

nonempty list considered greater than an empty one

List operations

head

```
Prelude> head [9,7,3,7]
9
```

tail

```
Prelude> tail [9,7,3,7]
[7,3,7]
```

last

```
Prelude> last [9,7,3,7]
7
```

init

```
Prelude> init [9,7,3,7]
[9,7,3]

Prelude> head [ ]
***Exception: Prelude.head: empty list
```

length

```
Prelude> length [9,7,3,7]
4
```

null

Prelude> null [4,4,4,4]
False
Prelude> null [ ]
True

reverse

Prelude> reverse [9,7,3,7]
[7,3,7,9]

take

```
Prelude> take 3 [9,7,3,7]
[9,7,3]
Prelude> take 1 [9,7,3,7]
[9]
Prelude take 20 [9,7,3,7]
[9,7,3,7]
Prelude> take 0 [9,7,3,7]
[ ]
```

drop

Prelude> drop 3 [9,7,3,7]
[7]
Prelude> drop 0 [9,7,3,7]
[9,7,3,7]
drop 10 [9,7,3,7]
[ ]

maximum

Prelude> maximum [2,2,8,5,90,3]
90

minimum

Prelude> minimum [2,2,8,5,90,3]
2

sum

```
Prelude> sum [9,7,3,7]
26
```

product

```
Prelude> product [9,7,3,7]
1323
```

elem

```
Prelude> 4 `elem` [9,7,3,7]
False
Prelude> 3 `elem` [9,7,3,7]
True
```

Ranges

Prelude> [1..10]
[1,2,3,4,5,6,7,8,9,10]

Prelude> [‘m’..’q’]
“mnopq”

Prelude> [‘A’..’J’]
“ABCDEFGHIJ”

Steps and Ranges

```
Prelude> [2,4..10]
[2,4,6,8,10]
Prelude> [3,6..30]
[3,6,9,12,15,18,21,24,27,30]
Prelude> [20..1]
[ ]
Prelude> [20,19..1]
[20,19,18,17,16,15,14,13,12,11,10,9,8,7,6,5,4,3,2,1]
```

Laziness and infinite lists

```
Prelude> take 11 [13,26..]
[13,26,39,52,65,78,91,104,117,130,143]
```

cycle

```
Prelude> take 5 (cycle [4,3,2])
[4,3,2,4,3]
```

repeat

```
Prelude> take 6 (repeat 7)
[7,7,7,7,7,7]
```

replicate

Prelude> replicate 5 ‘a’
“aaaaa”

caveat emptor: floating point numbers only have finite precision
Prelude> [0.1, 0.3..1]
[0.1,0.3,0.5,0.7,0.8999999999999,1.0999999999]

List comprehension

- Similar to set comprehension in mathematics
- build lists out of other lists: filter, transform and combine lists

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

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

Prelude> [ x | x <- [50..100], x `mod` 7 == 5]
[54,61,68,75,82,89,96]
```

```
Prelude> [ x | x <- [10..30], x /= 13, x /= 23, odd x ]
[11,15,17,19,21,25,27,29]

Prelude> [ x + y | x <- [17..20], y <- [10,100,0]]
[27,117,17,28,118,18,29,119,19,30,120,20]

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

Prelude> [ x*y | x <- [2,5,10], y <- [8,10,11], x*y > 50]
[55,80,100,110]
```

add to baby.hs
length' xs = sum [1 | _ <- xs]

*Main> length' "David"
5

add to baby.hs
keepLowerCase st = [c | c <- st, c 'elem' ['a'..'z']]

*Main> keepLowerCase "David"
"avid"
*Main> keepLowerCase "David + 12345"
"avid"

nested list comprehension

add to baby.hs

```
xxs = [[1,3,5,7,8,7,6,2],[2,3,4,5,6,1],[12,6,7,8,9,4,6,77]]
```

```
*Main> [ [x | x <- xs, even x] | xs <- xxs ]
[[8,6,2],[2,4,6],[12,6,8,4,6]]
```

Tuples

- lists only have homogeneous elements
- tuples can have heterogeneous elements
- lists have flexible size: grow, shrink
- tuples have a fixed size

Prelude> (1,10)
(1,10)

Prelude> (“Hi!”, ‘a’, 72, 1.01)
(“Hi!”, ’a’, 72, 1.01)

Use of tuples enforces a type discipline: try entering [(1,2),(1,2,3),(5,6)] at the command line

The type discipline extends to within the tuples, so [(1,2),(1,’a’)] is a problem too.

functions on pairs

fst

```
Prelude> fst (45, 78)
45
```

snd

```
Prelude> snd (45, 78)
78
```

zip

```
Prelude> zip [1,3,5,7,9] [‘a’,’b’,’c’,’j’,’k’]
[(1,’a’),(3,’b’),(5,’c’),(7,’j’),(9,’k’)]

Prelude> zip [5,3,4,2,6,7,8,9,0,1,2,3] [5,6,7]
[(5,5),(3,6),(4,7)]

Prelude> zip [10..] [5,6,7]
[(10,5),(11,6),(12,7)]
```

comprehension example

```
Prelude> let triples = [ (a,b,c) | c <- [1..10], a <- [1..10], b <- [1..10] ]

Prelude> let rightTriangles = [ (a,b,c) | c <- [1..10], a <- [1..c], b <- [1..a],
a^2 + b^2 == c^2 ]

Prelude> let rightTriangles' = [ (a,b,c) | c <- [1..10], a <- [1..c], b <-
[1..a], a^2 + b^2 == c^2, a+b+c == 24 ]

Prelude> rightTriangles'
[(6,8,10)]
```