Haskell 01:
Haskell Basics

Soomin Kim

Based on http://www.seas.upenn.edu/~cis194/spring15/lectures/

Contents

- What is Haskell?
- Themes
- Literate Haskell
- Declarations and variables
- Basic Types
- GHCi
- Arithmetic
- Boolean logic

Contents

- Defining basic functions
- Pairs
- Using functions, and multiple arguments
- Lists
- Constructing lists
- Functions on lists
- Combining functions
- A word about error messages

What is Haskell?

- Properties of Haskell
  - Functional
  - Pure
  - Lazy
  - Statically typed

Haskell is FUNCTIONAL

- Functions in Haskell are first-class
  - Functions are values
  - We can use functions in exactly the same ways as any other values

- The meaning of Haskell programs is centered around *evaluating expressions* rather than *executing instructions*

Haskell is PURE

- Haskell expressions are always *referentially transparent*
  - Everything is *immutable*
  - Expressions never have *side effects*
  - Programs are *deterministic*

- Benefits of functional paradigm
  - Equational reasoning and refactoring
  - Parallelism
  - Fewer headaches

Haskell is LAZY

- In Haskell, expressions are *not evaluated until their results are actually needed*
  - Easy to define a new *control structure* just by defining a function
  - Possible to define and work with *infinite data structures*
  - Enables a more compositional programming style
  - Reasoning about time and space usage becomes much more complicated

Haskell is STATICALLY TYPED

- Every Haskell expression has a type

- Types are all checked at *compile-time*

Themes

- 3 main themes of Haskell
  - Types
  - Abstraction
  - Wholemeal programming

Types

- Haskell's type system helps clarify thinking and express program structure
  - To *write down all the types* is usually the first step in writing Haskell program
  - This is a non-trivial design step because type system is so expressive
- It serves as a form of documentation
  - Just looking at a function's type tells you a lot about function
- It turns run-time errors into compile-time errors
  - It's much better to be able to fix errors up front than to just test a lot and hope for the best

Abstraction

- Taking similar pieces of code and factoring out their commonality is known as the process of *abstraction*

- Haskell's features like parametric polymorphism, higher-order functions and type classes all aid in the fight against repetition

Wholemeal programming

- Wholemeal programming means to think big
  - Work with an entire list, rather than a sequence of elements
  - Develop a solution space, rather than an indivisual solution
  - Imagine a graph, rather than a single path

Literate Haskell

- Source codes with an extension of .lhs: literate Haskell document
  - Only lines preceded by > and a space are code
  - Everything else is a comment

- Non-literate Haskell source files use .hs

Declarations and variables

```
x :: Int
x = 3
-- Line comment
{- Block
   Comment -}
x = 4
```

- Declares a variable x with type Int
- :: is pronounced “has type”
- Declares a value of x to be 3
- x = 4 generates an error
  - Multiple declarations of ‘x’
- = denotes *definition*, not *assignment*

Declarations and variables

```
y :: Int
y = y + 1
-- when this statement is evaluated
y = y + 1
  = (y + 1) + 1
  = ((y + 1) + 1) + 1
  = ...
```

Basic Types

```
i :: Int
i = -78
biggestInt :: Int
biggestInt = maxBound
soBig :: Integer
soBig = 2^(2^(2^(2^2)))
nDigits :: Int
nDigits = length (show
soBig)
```

- In the Haskell language standard, Ints has a range of $\pm 2^{29}$, but the exact size depends on machine's architecture
- Howerver, the Integer type is limited only by the amount of memory on machine

Basic Types

```
d1, d2 :: Double
d1 = 4.5387
d2 = 6.2831e-4
b :: Bool
b = True
c :: Char
c = ‘x’
s :: String
s = ‘Hello, Haskell!’
```

- Double is for floating-point numbers
- For a single-precision, there is a Float

GHCi

- GHCi is an interactive Haskell REPL
  - REPL: Read-Eval-Print-Loop

- :load(:l) : load Haskell files
- :reload(:r) : reload Haskell files
- :type(:t) : ask for the type of an expression
- :? : for a list of commands

Arithmetic

```
ex01 = 3 + 2
ex02 = 19 - 27
ex03 = 2.35 * 8.6
ex04 = 8.7 / 3.1
ex05 = mod 19 3
ex06 = 19 `mod` 3
ex07 = 7 ^ 222
ex08 = (-3) * (-7)
```

- \`backticks\` make a function name into an infix operator
- Negative numbers must often be surrounded by parentheses

Arithmetic

```
-- i :: Int, n :: Integer
badArith1 = i + n
badArith2 = i / i

ex09 = i `div` i
ex10 = 12 `div` 5
```

- Addition is only between values of the same numeric type
  - using fromIntegral, round, floor, ceiling
- / performs floating-point division only
  - for integer division, use div

Boolean logic

```
ex11 = True && False
ex12 = not (False || True)
ex13 = (‘a’ == ‘a’)
ex14 = (16 /= 3)
ex15 = (5 > 3)
       && (‘p’ <= ‘q’)
ex16 = “Haskell” > “C++”
```

- Haskell also has if-expression
  - if b then t else f
  - different with if-statement
  - else part of if-statement is optional
  - else part of if-expression is mandatory
- Idiomatic Haskell does not use if-expressions, but using pattern-matching or guards

Defining basic functions

```
sumtorial :: Integer -> Integer
sumtorial 0 = 0
sumtorial n = n + sumtorial (n – 1)
```

- Integer -> Integer says that the first Integer is input and the second Integer is output
- Each clause is checked in order from top to bottom, and the first matching clause is chosen

Defining basic functions

```
hailstone :: Integer -> Integer
hailstone n
   | n `mod` 2 == 0 = n `div` 2
   | otherwise      = 3 * n + 1
```

- Choices can also be made based on arbitrary Boolean expressions using *guards*

Pairs

```
p :: (Int, Char)
p = (3, ‘x’)

s :: (Int, Int) -> Int
s (x, y) = x + y
```

- (x, y) notation is used both for the *type* of a pair and a pair *value*
- The elements of a pair can be extracted again with *pattern matching*
- Never use triples, quadruples, …

Using functions, and multiple arguments

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


f 3 n + 1 7
(f 3 n) + (1 7)
```

- Syntax for the type of a function with multiple arguments: Arg1Type -> Arg2Type -> … -> ResultType
- Function application has higher precedence than any infix operators
  - Should use like f 3 (n + 1) 7 for the example

Lists

```
nums, range, range2 ::
[Integer]
nums   = [1, 2, 3, 19]
range  = [1 .. 100]
range2 = [2, 4 .. 100]
```

- Haskell also has *list comprehensions*

Lists

```
hello1 :: [Char]
hello1 = [‘h’, ‘e’, ‘l’,
‘l’, ‘o’]

hello2 :: String
hello2 = “hello”

helloSame = hello1 ==
hello2
```

- Strings are just lists of characters
- All the standard library functions for processing lists can also be used to process Strings

Constructing lists

```haskell
emptyList = []

ex19 = 3 : (1 : [])
ex20 = 2 : 3 : 4 : []
```

- Lists are built up from the empty list using the *cons* operator, (:)
- Cons takes an element and a list, and produces a new list with the element prepended to the front
- Lists are really *singly linked lists*, NOT arrays

Functions on lists

```
intListLength :: [Integer]
-> Integer
intListLength []     = 0
intListLength (x:xs) = 1
+ intListLength xs
```

- Since we don’t use x at all we could also replace it by an underscore
  - intListLength (_:xs) = 1 + intListLength xs

Combining functions

```
hailstoneLen :: Integer -> Integer
hailstoneLen n = intListLength (hailstoneSeq n) - 1
```

- Because of Haskell's lazy evaluation, each element of the sequence is only generated as needed
- The whole computation uses only O(1) memory
- *Don't be afraid to write small functions that transform whole data structures, and combine them to produce more complex functions*

A word about error messages

- **Don’t be scared of error messages!**

- When youget a huge error message, resist your initial impulse to run away; take a deep breath; and read it