Built-in types, Variables and Control Flow in Haskell

By Caroline Cox, Trang Dang, and Vy Pham

Built-In Types

Comparing Haskell's built-in types with Java: (Source: Haskell without the theory)

Basic data-types

Java	Haskell	E
`int`	`Int`	
`float`	`Float`	
`bool`	`Bool`	
`char`	`Char`	
`String`	`String`	
`double`	`Double`	
`void`	`()`	
`int[]` , `bool[]` , `double[]` , `String[]`	`[Int]` , `[Bool]` , `[Double]` , `[String]`	
`int[][]` , `bool[][]` , `int[][][]` , `float[][][]`	`[[Int]]` , `[[Bool]]` , `[[[Int]]]` , `` `[[[Float]]]` ``	

We can see how similar Haskell's built-in Data types are to java!

Similarity between Haskell built-in types and Java built-in types

We also have `Int, Float, Double, Char,` and `Bool` to represent numbers, characters and Boolean values.

1. Booleans: `Bool`
   - There are two values of this data type: `True` or `False`.
   - The basic Boolean functions are `and (&&), or (||)` and `not`
   - There is also `otherwise` defined as `True`

2. Characters: `Char`
   - Represent Unicode characters
3. Numbers: `Int, Float` and `Double`
   - `Int`: slight difference – in Java, 32 bit – in Haskell, depends on the machine but usually 32 or 64 bits (represent Integers)
   - `Float`: 32-bit representation of floating-point numbers
   - `Double`: 64-bit representation of floating-point numbers

Some examples:

```
x :: Int

x = 1

'a' :: Char
```

Differences between Haskell's built-in types and Java's built-in types

`Int` vs `Integer`

First, Haskell has both `Integer` and `Int` types. Unlike Java, `Integer` isn't a wrapper class of `Int`: While `Int` represents integer with 32 or 64 bits, `Integer` can represent any large number up to the storage limits of your machine. (Source: haskell wiki)

Because of the promised 32- or 64-bits representation, it's quicker to operate on `Int` than on `Integer`. However, looking back to the problem with the voting assignment for CS113, it appears that overflow and underflow can cause strange bugs, so we may be better off with Integer when we are dealing with a very unpredictable number range.

Control Structures

If <condition> then <true-value> else <false-value>

Condition is boolean value and if it is true then the <true-value> is returned; else, it must be false and <false-value> is returned

Similar to if...else expressions in other languages except else clause is required because expression must return a value

Can also be written with guards `( | )` and using the `otherwise` keyword, which is an alias for `boolean True`. The last guard indicates a catch-all expression that returns a value for the expression if none of the previous conditions following previous guards are true

Case expression

Similar to the switch statement in C and Java

Syntax: f <variable> = case <variable> of

<value1> -> <result1>

<value2> -> <result2>

<value 3> -> <result3>

_-> <defaultValue>

Character _ is the wildcard pattern; stands for value of variable that is something other than previously tested values

Cases must be indented to the right after the keyword “of”

No for or while loops, which means recursion must be used to obtain the same results acquired in other imperative languages using these constructs

“Do” keyword is used in Haskell I/O, meaning I/O is more imperative. However, Haskell is still purely functional because the more imperative nature of I/O is distinct and kept separate from the purely functional aspects of Haskell.

Variables

Haskell is a programming language with immutable variables, meaning, after declaring a variable, its value cannot be changed within the same scope. Mutable variable can be programmed via monads.

The type of a variable does not have to be explicitly stated, since Haskell uses type inference.

When a new variable is declared, its value is automatically allocated. The garbage collector will collect values that are out of scope.

Since Haskell has immutable variables, local bindings can be used to declare a new variable of the same name in a specific local scope.

Example:

```
let x = 1

let x = 1 + 2 in x + 3
```