**Haskell: Introduction and Integers**

**Basic data-types**

|  |  |  |
| --- | --- | --- |
| **Java** | **Haskell** | **Example values in Haskell** |
| **int** | **Int** | **1, 2, -1, -2, etc** |
| **float** | **Float** | **1.0, -1.23, -2.5, etc** |
| **bool** | **Bool** | **True, False. Nothing else (see notes below)** |
| **char** | **Char** | **'a', 'b', 'c', '\n', '\r', '\t', etc.** |
| **String** | **String** | **"hello world", "abc", "a", "b" (notice the single quotes - ' vs double quotes - " difference between Char and String)** |
| **double** | **Double** | **1.0, -1.23, -2.5, etc (basically larger values than Float)** |
| **void** | **()** | **() - see notes below** |
| **int[], bool[], double[], String[]** | **[Int], [Bool], [Double], [String]** | **[1, 2, 3, 4] or [True, False, False], or [1.0, 5.6, 12.34] or ["hello", "world", "haskell", "rocks"]** |
| **int[][], bool[][], int[][][], float[][][]** | **[[Int]], [[Bool]], [[[Int]]], `[[[Float]]]`** | **[[1, 2], [3, 4, 5], [6, 7, 8]] or [["Multi", "dim"], ["list", "in", "haskell"]]** |

***Imp Note : In Haskell, all data-types start with a capital letter, eg. Bool instead of bool***

**System Commands**

*The following are the most important HUGS [] system commands:* **Here You Go** (Internet slang)

* **:q** ending session
* **:?** getting help
* **:e** editing file
* **:t** getting type

**Integral Types**

Haskell is a strongly typed language and it is good programming practice, though not a

requirement enforced by the HUGS system, to always include the type of any function that you define.

***Haskell has two integral types:***

* **Int** limited-precision or single-precision integers
* **Integer** arbitrary-precision integers

**Basic arithmetical operators**

Haskell has the usual binary infix arithmetical operators:

* **+** addition
* **-** subtraction
* **\*** multiplication
* **^** exponentiation

It also has the unary prefix operator - (minus or negative) and the following binary prefix operators:

* **min** minimum
* **max** maximum
* **gcd** greatest common divisor
* **lcm** lowest common multiple
* **div** integer division
* **mod** remainder after integer division

*The functions div and mod satisfy the following identity:*

**(div x y) \* y + (mod x y) = x**

Haskell also has a Boolean data type Bool, two of whose values are True and False.

It has the usual binary infix Boolean-valued operators:

* **==** equals
* **/=** not equal
* **<** less than
* **>** greater than
* **<=** less than or equal to
* **>=** greater than or equal to

There are also two infix binary logical operators:

* **||** or operator
* **&&** and operator

There is also a prefix unary logical operator:

* **not** not operator

**Defining Functions**

* There is no semi-colon at the end of the statement.
* No curly-braces.
* There are no parentheses around function arguments.
* Function arguments are separated by spaces instead of commas.
* Function signatures, i.e. Int -> Int -> Int, do not have variable names. The variable names are given directly in the function definition / body.
* In fact, function signatures don’t use any special syntax for the return-type of a function. The type after the last pointy-arrow (->) is the return-type of the function.
* The function definition is almost written like a mathematical equation. Do not be afraid of this. In a few days’/weeks’ you will become familiar with this syntax and will not even think about this.
* Do not confuse the = symbol in the function definition with variable assignment.
* There is no return statement. The return value of a function is the value of the last expression evaluated, which in this case is x + y.

Functions are defined like this in Haskell:

**addInts :: Int -> Int -> Int**

**Prelude**> **addInts x y = x + y**

**Prelude**> **addInts 10 20 // result : 30**

**sqInt :: Int -> Int**

**Prelude**> **sqInt x = x \* x**

**Prelude**> **sqInt 5 5 // result : 25**

**smallerInt :: Int -> Int -> Int**

**Prelude**>**smallerInt x y | x <= y = x | otherwise = y**

Repetition or iteration is obtained by using recursion. , For example, the following function given an integer x as its argument, returns the sum of all integers between 0

**and x:**

**sumInt :: Int -> Int**

**Prelude**>**sumInt x | x == 0 = 0 | otherwise = x + sumInt (x - 1)**

Using pattern-matching this can also be defined like this:

**sumInt :: Int -> Int**

**sumInt 0 = 0**

**sumInt x = x + sumInt (x - 1)**

This, however, fails miserably for negative arguments. These can be caught as follows:

**sumInt :: Int -> Int**

**sumInt 0 = 0**

**sumInt x**

**| x < 0 = error "sumInt undefined when x < 0"**

**| otherwise = x + sumInt (x - 1)**

**Type Classes and Qualified Types**

In addition to types like Int and Integer Haskell also has a type class. To motivate these consider the problem of defining a square function for arbitrary-precision integers:

**sqInteger :: Integer -> Integer**

**sqInteger x = x \* x**

This largely duplicates the earlier definition of sqInt. It is better to define a function:

**sqIntegral:**

**sqIntegral :: Integral a => a -> a**

**sqIntegral x = x \* x**

Here, Integral is a type class whose elements are the two types Int and Integer and **a** is a type variable. The type **Integral a => a -> a** is called a qualified type.

**Turning Prefix Functions into Infix Ones**

Haskell has a mechanism for converting prefix functions into infix ones. Enclosing a binary prefix operator in single opening quotation marks turns it into an infix operator.

*For example:*

**div 7 2 is equivalent to 7 ‘div‘ 2**

**Turning Infix Functions into Prefix Ones**

There is also a mechanism for turning infix operators into prefix ones. Enclosing a binary infix operator in parentheses turns it into a prefix operator.

*For example,*

**(+) 2 3 is equivalent to 2 + 3**

**(==) 3 4 is equivalent to 3 == 4**

**(\*2) 7 is equivalent to 7 \* 2**

**(0<) 8 is equivalent to 0 < 8**

*An expression of the form* ***(+)*** *or* ***(0<)*** *is called a section.*

**Haskell Lists**

**Introduction**

The most important data type in a functional language is the list. A list is a linearly ordered collection of elements. All elements of a list must be of the same type.

*For Examples:*

**[3, 7, 5, 88] :: [Int]**

**[’t’, ’i’, ’m’, ’e’] :: [Char]**

**"time" :: [Char]**

**[[2, 3], [4, 8, 17]] :: [[Int]]**

Haskell provides many list operators. Some are:

* **:** => binary infix sticks an element at the front of a list
* **head** =>unary prefix extracts the first element of a non-empty list
* **tail** =>unary prefix returns the tail of a non-empty list
* **length =>** unary prefix returns the length of a list
* **!! =>** binary infix extracts an element of a list

A function to sum the elements of a list of integers can be defined like this:

**sum :: Integral a => [a] -> [a]**

**sum ys | ys == [] = 0 | otherwise = head ys + sum (tail ys)**

**It is better, however, to use pattern-matching thus:**

**sum :: Integral a => [a] -> [a]**

**sum [] = 0**

**sum (y:ys) = y + sum ys**

**List Addition and Subtraction**

Two useful binary infix functions on lists are ++ (list addition) and \\ (list subtraction).

List addition takes two lists as its arguments and sticks them together. List subtraction removes elements from a list, for example:

**[1, 2, 3, 4, 5] \\ [1, 4] is equivalent to [2, 3, 5]**

**[1, 1, 1, 1] \\ [1, 4] is equivalent to [1, 1, 1]**

**[1, 1, 1, 1] \\ [1, 1] is equivalent to [1, 1]**

List subtraction is not predefined in the version of Haskell used here, but it can be defined like this:

**(\\) :: Eq a => [a] -> [a] -> [a]**

**[] \\ \_ = []**

**xs \\ [] = xs**

**(x:xs) \\ (y:ys)**

**| x == y = xs \\ ys**

**| otherwise = (x : (xs \\ [y])) \\ ys**

**Local Definition**

Haskell supports local definitions, for example:

**foo x**

**| x > 0 = p + q**

**| x <= 0 = p - q**

**where**

**p = x^2 + 1**

**q = 3\*x^3 - 5**

Local definitions obey Landing’s offside rule:

The southeast quadrant that just contains the phrase’s first symbol must contain the entire phrase, except possibly for bracketed sub expressions.

**Programming Style**

The following two definitions of a leap year illustrate bad and good programming style:

**leap1 y = (y `mod` 4 == 0) && (y `mod` 100 /= 0 || y `mod` 400 == 0)**

**leap2 y | y `mod` 100 == 0 = y `mod` 400 == 0 | otherwise = y `mod` 4 == 0**

*In Haskell leap2 is considered more elegant than leap1.*

**Haskell : Floating-point Numbers and Characters**

**Introduction**

Haskell has two types for floating-point numbers:

* Float single-precision
* Double double-precision

Floating-point numbers can be represented in two ways. First, using a decimal point:

* 2.0 , 33.873 , -8.3377

Second, by means of the so-called scientific notation:

* 33.61e6 (equivalent to 33.61 \_ 106)
* 3.7e-2 (equivalent to 3.7 \_ 10−2)
* -3.7e2 (equivalent to −3.7 \_ 102)

Haskell has the usual binary infix floating-point operators, namely

* **+** addition
* **-** subtraction
* **\*** multiplication
* **/** division
* **\*\*** exponentiation

It also has the unary prefix operator - (minus or negative) and the following unary prefix operators:

* **cos** cosine
* **sin** sine
* **tan** tangent
* **log** logarithms to base e
* **acos** inverse cosine
* **asin** inverse sine
* **atan** inverse tangent
* **exp** powers of e
* **sqrt** square root

Haskell has some useful functions for converting floating-point numbers into single precision integers:

* **ceiling 2.3 is equivalent to 3**
* **floor 2.3 is equivalent to 2**
* **round 2.3 is equivalent to 2**
* **round 2.7 is equivalent to 3**

These are all of type Float -> Int. The function fromInt of type Int -> Float converts a limited-precision integer into a single-precision floating-point number.

**Numerical Type Classes**

So far four numerical types in Haskell have been introduced, namely Int, Integer, Float and Double. It is tedious to define a new function that squares its argument, say, for each numerical type:

* **sqInt :: Int -> Int**
  + **sqInt x = x \* x**
* **sqInteger :: Integer -> Integer**
  + **sqInteger x = x \* x**
* **sqFloat :: Float -> Float**
  + **sqFloat x = x \* x**
* **sqDouble :: Double -> Double**
  + **sqDouble x = x \* x**

Haskell has several type classes which allow one definition to do the work of more than one of the above monomorphic (opposite of **polymorphism**) [i.e. single form] definitions:

* **sqIntegral :: Integral a => a -> a**
  + **sqIntegral x = x \* x**
* **sqFractional :: Fractional a => a -> a**
  + **sqFractional x = x \* x**
* **sqReal :: Real a => a -> a**
  + **sqReal x = x \* x**

The type class Integral contains the two types Int and Integer. The type class Fractional contains the two types Float and Double. The type class Real contains the four types Int, Integer, Float and Double. These, and some other important types, can be represented by the following inclusion diagram:

Eq

/ \

/ \

/ \

/ \

/ \

/ \

Ord Num

\ /

\ /

\ /

\ /

\ /

\ /

Real

/ \

/ \

/ \

/ \

/ \

/ \

Integral Fractional

**Characters**

The type Char contains characters. Elements of Char are written enclosed in single closing quotation marks, for example:

* **’a’**
* **’B’**
* **’4’**
* **’\t’ tab**
* **’\n’ newline**
* **’\\’ backslash**
* **’\’’ single closing quotation mark**
* **’\"’ double quotation mark**

There are several useful functions dealing with characters:

* **toUpper Char -> Char**
* **toLower Char -> Char**
* **ord Char -> Int into ASCII code**
* **chr Int -> Char from ASCII code**
* **isAscii Char -> Bool**
* **isUpper Char -> Bool**
* **isLower Char -> Bool**
* **isAlpha Char -> Bool**
* **isDigit Char -> Bool**
* **isAlphaNum Char -> Bool**

**if-then-else”**

if-then-else in Haskell works very similar to other languages. For example, here’s a function to return "odd" / "even" depending upon the input number:

**module Check where**

**checkNumber :: Int -> String**

**checkNumber y =**

**if (mod y 2) == 0**

**then "even"**

**else "odd"**

**Prelude**> **:l Check**

**Prelude**> **checkNumber 10**

**"even"**

**Prelude**> **checkNumber 7**

**"odd"**

**Summing up a list of integers**

**First, let’s quickly look at the head and tail functions that operate on lists:**

**GHCi**> **head [1, 2, 3, 4] -- result : 1**

**GHCi**> **tail [1, 2, 3, 4] -- result : [2,3,4]**

**Now, let’s use this functions to write our own function that computes the sum of a list of integers:**

**module SumOfList where**

**sumOfList :: Int -> [Int] -> Int**

**sumOfList total lst =**

**if (lst == [])**

**then total**

**else sumOfList (total + (head lst)) (tail lst)**

**Prelude**> **:l SumOfList**

**Prelude**> **sumOfList 0 [1, 2, 3, 4, 5] --result 15**

***Imp : If you notice, we had to pass-in the initial value of the total as 0. Notice, how we haven’t used any for or do-while loop. We aren’t even changing the value of any variable. Each step in the recursion is passing new values to the next function call (which happens to be itself!)***

**Summing-up all even numbers in a list of integers**

***Now, let’s modify our previous code to sum-up only the even numbers in a list:***

**module SumOfList where**

sumOfEven::Int-> **[**Int**]** ->Int

sumOfEven **total lst** =

**if (lst** ==[]**)**

**then total**

**else if (mod (head lst) 2)** == **0**

**then sumOfevent (total** + **(head lst)) (tail lst)**

**else sumOfEven total (tail lst)**

***Doubling all numbers in a list of integers***

**module ListNum where**

doubleList:: **[**Int**]** -> **[**Int**]** -> **[**Int**]**

doubleList **processedList remainingList** =

**if (remainingList** ==[]**)**

**then processedList**

**else doubleList (processedList** ++ **[(head remainingList)** \* **2]) (tail remainingList)**

**-- GHCi> :l ListNum**

**-- GHCi> doubleList [] [1, 2, 3, 4]**

**-- [2, 4, 6, 8]**

***Converting a String to uppercase***

**module UpperLetter where**

**-- NOTE: importing the `toUpper` function from the Data.Char module.**

**-- You should look up the signature of this function from the docs**

**import Data.Char (**toUpper**)**

toUppercase::String->String->String

toUppercase **processedString remainingString** =

**if (remainingString** ==[]**)**

**then processedString**

**else toUppercase (processedString** ++ **[toUpper (head remainingString)] (tail remainingString)**

**-- GHCi> :l UpperLetter**

**-- GHCi> toUppercase [] "hello world"**

**-- "HELLO WORLD"**