Haskell Programming

tutorialspoint

SIMPLY EASY LEARNING

About the Tutorial

Haskell is a widely used purely functional language. Functional programming is based on mathematical functions. Besides Haskell, some of the other popular languages that follow Functional Programming paradigm include: Lisp, Python, Erlang, Racket, F#, Clojure, etc.

Haskell is more intelligent than other popular programming languages such as Java, C, C++, PHP, etc. In this tutorial, we will discuss the fundamental concepts and functionalities of Haskell using relevant examples for easy understanding.

Audience

This tutorial has been prepared for beginners to let them understand the basic concepts of functional programming using Haskell as a programming language.

Prerequisites

Although it is a beginners' tutorial, we assume that the readers have a reasonable exposure to any programming environment and knowledge of basic concepts such as variables, commands, syntax, etc.

Copyright & Disclaimer

Table of Contents

1. HASKELL – OVERVIEW

Haskell is a Functional Programming Language that has been specially designed to handle symbolic computation and list processing applications. Functional programming is based on mathematical functions. Besides Haskell, some of the other popular languages that follow Functional Programming paradigm include: Lisp, Python, Erlang, Racket, F#, Clojure, etc.

In **conventional programing**, instructions are taken as a set of declarations in a specific syntax or format, but in the case of **functional programing**, all the computation is considered as a combination of separate mathematical functions.

Going Functional with Haskell

Haskell is a widely used purely functional language. Here, we have listed down a few points that make this language so special over other conventional programing languages such as Java, C, C++, PHP, etc.

- **Functional Language**: In conventional programing language, we instruct the compiler a series of tasks which is nothing but telling your computer "what to do" and "how to do?" But in Haskell we will tell our computer "what it is?"

- **Laziness**: Haskell is a lazy language. By **lazy**, we mean that Haskell won't evaluate any expression without any reason. When the evaluation engine finds that an expression needs to be evaluated, then it creates a **thunk data structure** to collect all the required information for that specific evaluation and a pointer to that **thunk data structure**. The evaluation engine will start working only when it is required to evaluate that specific expression.

- **Modularity**: A Haskell application is nothing but a series of functions. We can say that a Haskell application is a collection of numerous small Haskell applications.

- **Statically Typed**: In conventional programing language, we need to define a series of variables along with their type. In contrast, Haskell is a strictly typed language. By the term, Strictly Typed language, we mean the Haskell compiler is intelligent enough to figure out the type of the variable declared, hence we need not explicitly mention the type of the variable used.

- **Maintainability**: Haskell applications are modular and hence, it is very easy and cost-effective to maintain them.

Functional programs are more concurrent and they follow parallelism in execution to provide more accurate and better performance. Haskell is no exception; it has been developed in a way to handle **multithreading** effectively.

Hello World

It is a simple example to demonstrate the dynamism of Haskell. Take a look at the following code. All that we need is just one line to print "Hello Word" on the console.

```
main = putStrLn "Hello World"
```

Once the Haskell compiler encounters the above piece of code, it promptly yields the following output:

```
Hello World
```

We will provide plenty of examples throughout this tutorial to showcase the power and simplicity of Haskell.

Haskell ─ Environment Set Up

Try it Online

We have set up the Haskell programing environment online at: https://www.tutorialspoint.com/compile_haskell_online.php

This online editor has plenty of options to practice Haskell programing examples. Go to the terminal section of the page and type "**ghci**". This command automatically loads Haskell compiler and starts Haskell online. You will receive the following output after using the **ghci** command.

```
sh-4.3$ ghci

GHCi,version7.8.4:http://www.haskell.org/ghc/:?forhelp

Loading package ghc-prim...linking...done.

Loading packageinteger gmp...linking... done.

Loading package base...linking...done.

Prelude>
```

If you still want to use Haskell offline in your local system, then you need to download the available Haskell setup from its official webpage: **https://www.haskell.org/downloads**

There are three different types of **installers** available in the market:

- **Minimal Installer**: It provides GHC (The Glasgow Haskell Compiler), CABAL (Common Architecture for Building Applications and Libraries), and Stack tools.

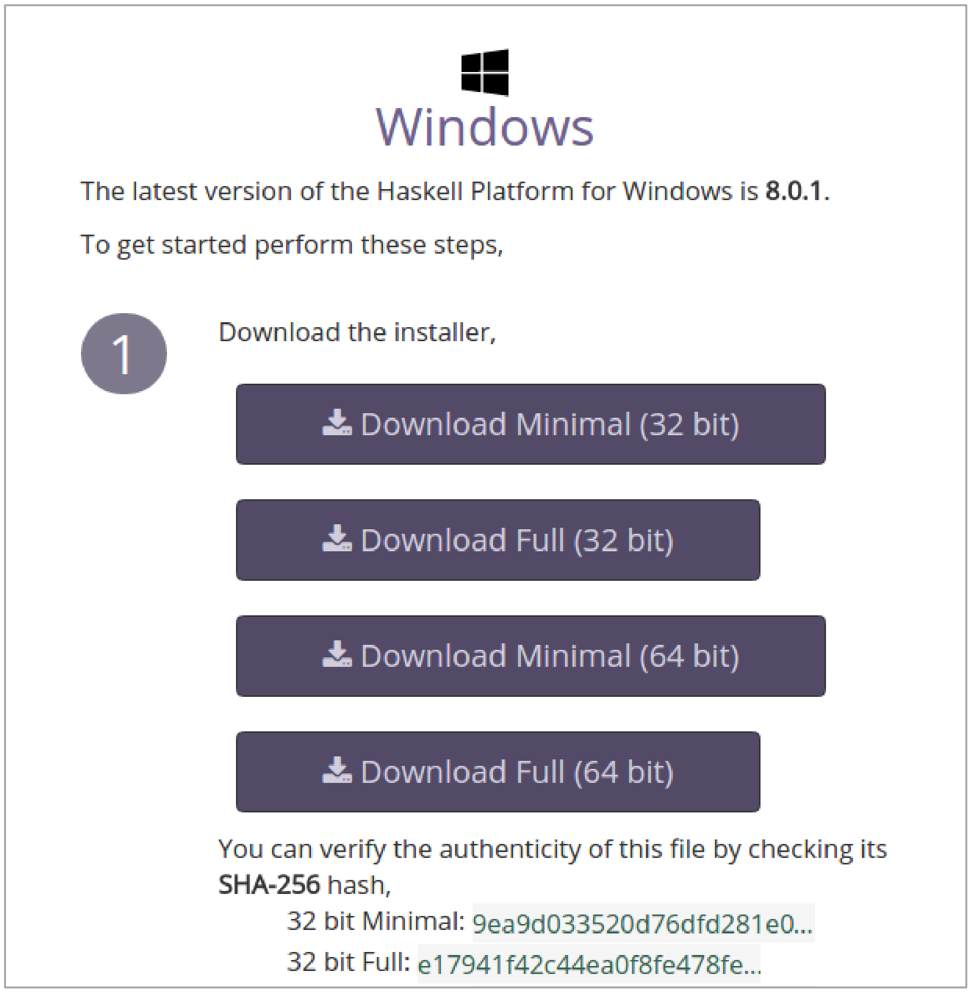
- **Stack Installer**: In this installer, the GHC can be downloaded in a cross-platform of managed toll chain. It will install your application globally such that it can update its API tools whenever required. It automatically resolves all the Haskell-oriented dependencies.

- **Haskell Platform**: This is the best way to install Haskell because it will install the entire platform in your machine and that to from one specific location. This installer is not distributive like the above two installers.

We have seen different types of installer available in market now let us see how to use those installers in our machine. In this tutorial we are going to use Haskell platform installer to install Haskell compiler in our system.

Environment Set Up in Windows

To set up Haskell environment on your Windows computer, go to their official website **https://www.haskell.org/platform/windows.html** and download the Installer according to your customizable architecture.



Windows

The latest version of the Haskell Platform for Windows is **8.0.1**.

To get started perform these steps,

1 Download the installer,

You can verify the authenticity of this file by checking its **SHA-256** hash,

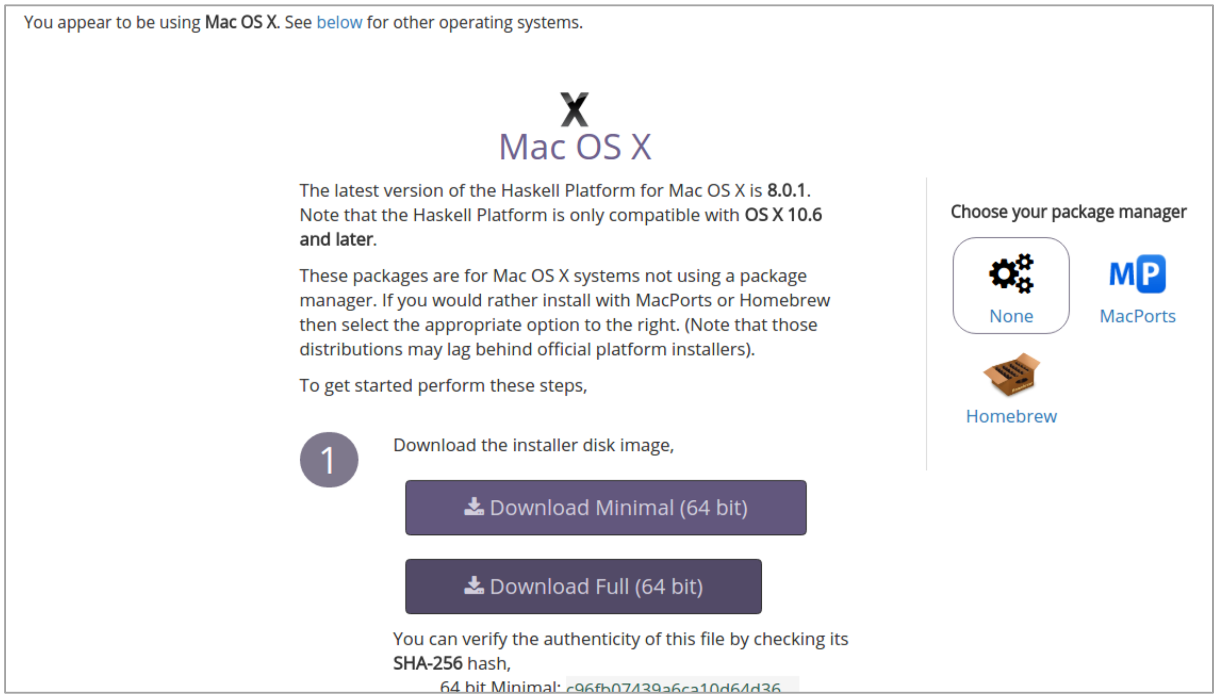
32 bit Minimal: 9ea9d033520d76dfd281e0...

32 bit Full: e17941f42c44ea0f8fe478fe...

Check out your system's architecture and download the corresponding setup file and run it. It will install like any other Windows application. You may need to update the CABAL configuration of your system.

Environment Set Up in MAC

To set up Haskell environment on your MAC system, go to their official website **https://www.haskell.org/platform/mac.html** and download the Mac installer.



You appear to be using **Mac OS X**. See below for other operating systems.

Mac OS X

The latest version of the Haskell Platform for Mac OS X is **8.0.1**. Note that the Haskell Platform is only compatible with **OS X 10.6 and later**.

These packages are for Mac OS X systems not using a package manager. If you would rather install with MacPorts or Homebrew then select the appropriate option to the right. (Note that those distributions may lag behind official platform installers).

To get started perform these steps,

1 Download the installer disk image,

You can verify the authenticity of this file by checking its **SHA-256** hash,

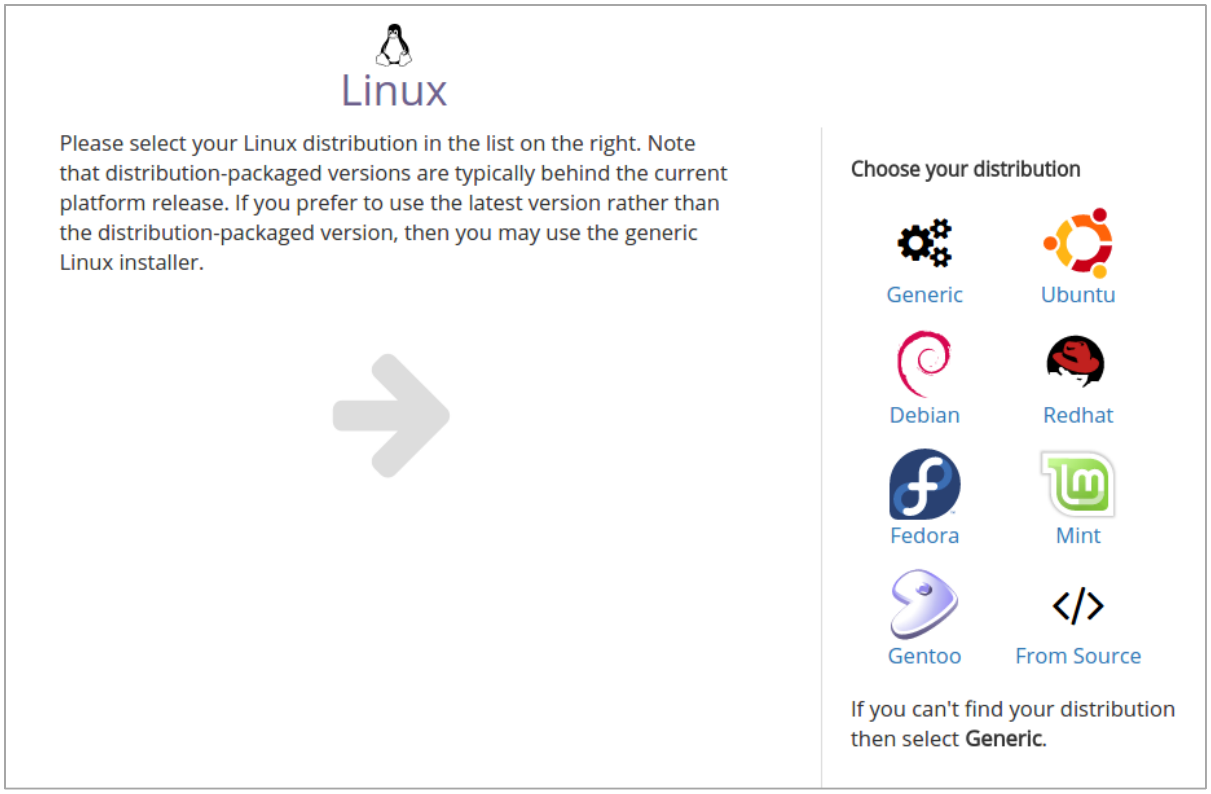
64 bit Minimal: c96fb07439a6ca10d64d36

Environment Set Up in Linux

Installing Haskell on a Linux-based system requires to run some command which is not that much easy like MAC and Windows. Yes, it is tiresome but it is reliable.

You can follow the steps given below to install Haskell on your Linux system:

Step 1: To set up Haskell environment on your Linux system, go to the official website **https://www.haskell.org/platform/linux.html** and choose your distribution. You will find the following screen on your browser.



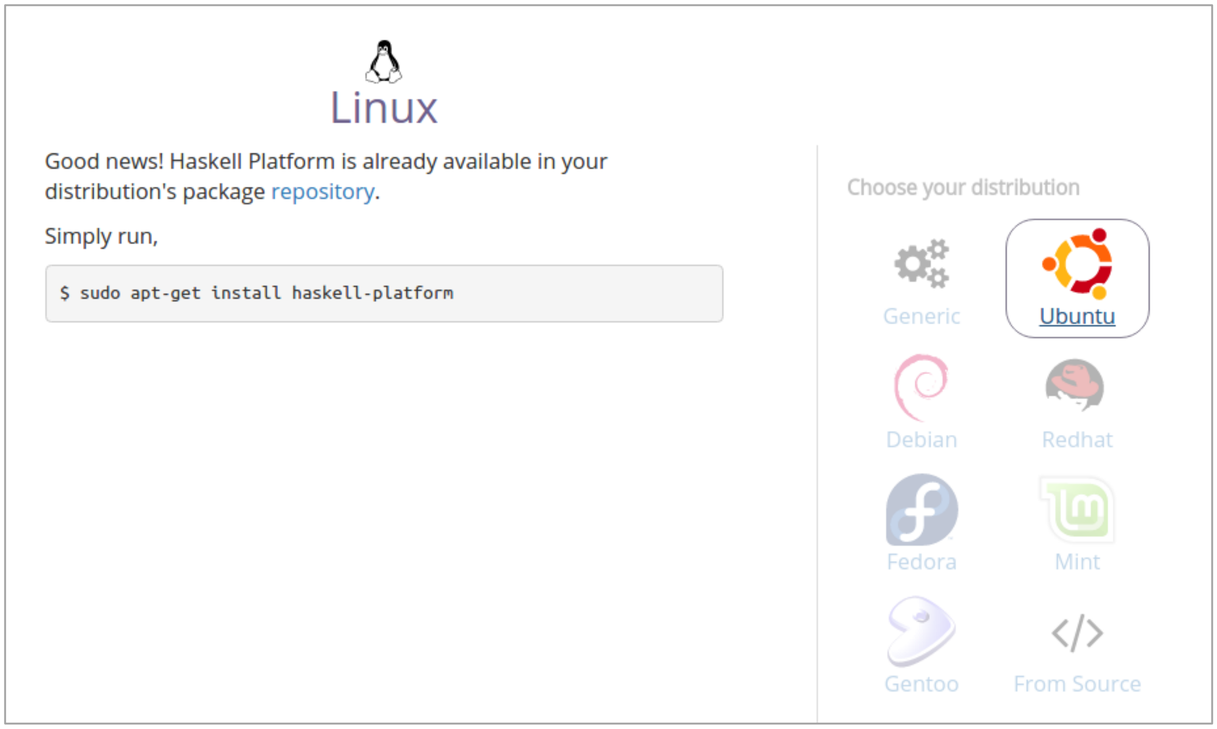
Linux

Please select your Linux distribution in the list on the right. Note that distribution-packaged versions are typically behind the current platform release. If you prefer to use the latest version rather than the distribution-packaged version, then you may use the generic Linux installer.

Choose your distribution

If you can't find your distribution then select **Generic**.

Step 2: Select your Distribution. In our case, we are using Ubuntu. After selecting this option, you will get the following page on your screen with the command to install the Haskell in our local system.



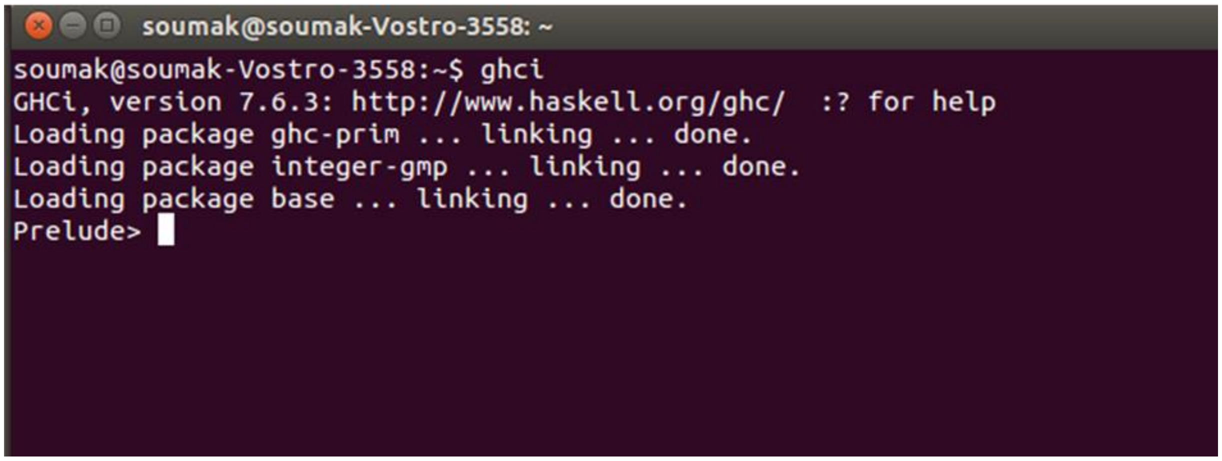
Good news! Haskell Platform is already available in your distribution's package repository.

Simply run,

```
$ sudo apt-get install haskell-platform
```

Step 3: Open a terminal by pressing Ctrl + Alt + T. Run the command "**$ sudo apt-get install haskell-platform**" and press Enter. It will automatically start downloading Haskell on your system after authenticating you with the root password. After installing, you will receive a confirmation message.

Step 5: Go to your terminal again and run the GHCI command. Once you get the Prelude prompt, you are ready to use Haskell on your local system.



```
soumak@soumak-Vostro-3558:~$ ghci
GHCi, version 7.6.3: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Prelude> 
```

To exit from the GHCI prolog, you can use the command "`:quit exit`".

2. HASKELL – BASIC DATA MODELS

Haskell is a purely functional programing language, hence it is much more interactive and intelligent than other programming languages. In this chapter, we will learn about basic data models of Haskell which are actually predefined or somehow intelligently decoded into the computer memory.

Throughout this tutorial, we will use the Haskell online platform available on our website (https://www.tutorialspoint.com/codingground.htm).

Numbers

Haskell is intelligent enough to decode some number as a number. Therefore, you need not mention its type externally as we usually do in case of other programing languages. As per example go to your prelude command prompt and just run "2+2" and hit enter.

```
sh-4.3$ ghci
GHCi, version 7.6.3: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Prelude> 2+2
```

You will receive the following output as a result.

```
4
```

In the above code, we just passed two numbers as arguments to the GHCI compiler without predefining their type, but compiler could easily decode these two entries as numbers.

Now, let us try a little more complex mathematical calculation and see whether our intelligent compiler give us the correct output or not. Try with "15+(5*5)-40"

```
Prelude> 15+(5*5)-40
```

The above expression yields "0" as per the expected output.

```
0
```

Characters

Like numbers, Haskell can intelligently identify a character given in as an input to it. Go to your Haskell command prompt and type any character with double or single quotation.

Let us provide following line as input and check its output.

```
Prelude> :t "a"
```

It will produce the following output:

```
"a" :: [Char]
```

Remember you use **(:t)** while supplying the input. In the above example, **(:t)** is to include the specific type related to the inputs. We will learn more about this type in the upcoming chapters.

Take a look at the following example where we are passing some invalid input as a char which in turn leads to an error.

```
Prelude> :t a
<interactive>:1:1: Not in scope: 'a'

Prelude> a
<interactive>:4:1: Not in scope: 'a'
```

By the error message "<interactive>:4:1: Not in scope: `a'" the Haskell compiler is warning us that it is not able to recognize your input. Haskell is a type of language where everything is represented using a number.

Haskell follows conventional ASCII encoding style. Let us take a look at the following example to understand more:

```
Prelude> '\97'
'a'

Prelude> '\67'
'C'
```

Look how your input gets decoded into ASCII format.

String

A **string** is nothing but a collection of characters. There is no specific syntax for using string, but Haskell follows the conventional style of representing a string with double quotation.

Take a look at the following example where we are passing the string "Tutorialspoint.com".

```
Prelude> :t "tutorialspoint.com"
```

It will produce the following output on screen:

```
"tutorialspoint.com" :: [Char]
```

See how the entire string has been decoded as an array of Char only. Let us move to the other data type and its syntax. Once we start our actual practice, we will be habituated with all the data type and its use.

Boolean

Boolean data type is also pretty much straightforward like other data type. Look at the following example where we will use different Boolean operations using some Boolean inputs such as "True" or "False".

```
Prelude> True && True

True


Prelude> True && False

False



Prelude> True || True
True

Prelude> True || False

True
```

In the above example, we need not mention that "True" and "False" are the Boolean values. Haskell itself can decode it and do the respective operations. Let us modify our inputs with "true" or "false".

```
Prelude> true
```

It will produce the following output:

```
<interactive>:9:1: Not in scope: 'true'
```

In the above example, Haskell could not differentiate between "true" and a number value, hence our input "true" is not a number. Hence, the Haskell compiler throws an error stating that our input is not its scope.

List and List Comprehension

Like other data types, **List** is also a very useful data type used in Haskell. As per example, [a,b,c] is a list of characters, hence, by definition, List is a collection of same data type separated by comma.

Like other data types, you need not declare a List as a List. Haskell is intelligent enough to decode your input by looking at the syntax used in the expression.

Take a look at the following example which shows how Haskell treats a List.

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

It will produce the following output:

```
[1,2,3,4,5]
```

Lists in Haskell are homogeneous in nature, which means they won't allow you to declare a list of different kind of data type. Any list like [1,2,3,4,5,a,b,c,d,e,f] will produce an error.

```
Prelude> [1,2,3,4,5,a,b,c,d,e,f]
```

This code will produce the following error:

```
<interactive>:17:12: Not in scope: 'a'
<interactive>:17:14: Not in scope: 'b'
<interactive>:17:16: Not in scope: 'c'
<interactive>:17:18: Not in scope: 'd'
<interactive>:17:20: Not in scope: 'e'
<interactive>:17:22: Not in scope: 'f'
```

List Comprehension

List comprehension is the process of generating a list using mathematical expression. Look at the following example where we are generating a list using mathematical expression in the format of [output | range ,condition].

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

Prelude> [x*2| x<-[1..5]]
[2,4,6,8,10]

Prelude> [x| x<-[1..5]]
[1,2,3,4,5]
```

This method of creating one List using mathematical expression is called as **List Comprehension**.

Tuple

Haskell provides another way to declare multiple values in a single data type. It is known as **Tuple**. A Tuple can be considered as a List, however there are some technical differences in between a Tuple and a List.

A Tuple is an immutable data type, as we cannot modify the number of elements at runtime, whereas a List is a mutable data type.

On the other hand, List is a homogeneous data type, but Tuple is heterogeneous in nature, because a Tuple may contain different type of data inside it.

Tuples are represented by single parenthesis. Take a look at the following example to see how Haskell treats a Tuple.

```
Prelude> (1,1,'a')
```

It will produce the following output:

```
(1,1,'a')
```

In the above example, we have used one Tuple with two **number** type variables, and a **char** type variable.

3. HASKELL – BASIC OPERATORS

In this chapter, we will learn about different operators used in Haskell. Like other programming languages, Haskell intelligently handles some basic operations like addition, subtraction, multiplication, etc. In the upcoming chapters, we will learn more about different operators and their use.

In this chapter, we will use different operators in Haskell using our online platform (https://www.tutorialspoint.com/codingground.htm). Remember we are using only **integer** type numbers because we will learn more about **decimal** type numbers in the subsequent chapters.

Addition Operator

As the name suggests, the addition (+) operator is used for addition function. The following sample code shows how you can add two integer numbers in Haskell:

```
main = do
    let var1 = 2
    let var2 = 3
    putStrLn "The addition of the two numbers is:"
    print(var1 + var2)
```

In the above file, we have created two separate variables **var1** and **var2**. At the end, we are printing the result using the **addition** operator. Use the **compile** and **execute** button to run your code.

This code will produce the following output on screen:

```
The addition of the two numbers is:
5
```

Subtraction Operator

As the name suggests, this operator is used for subtraction operation. The following sample code shows how you can subtract two integer numbers in Haskell:

```
main=do
     let var1 = 10
     let var2 = 6
     putStrLn "The Subtraction of the two numbers is:"
     print(var1 - var2)
```

In this example, we have created two variables **var1** and **var2**. Thereafter, we use the subtraction (–) operator to subtract the two values.

This code will produce the following output on screen:

```
The Subtraction of the two numbers is:
4
```

Multiplication Operator

This operator is used for multiplication operations. The following code shows how to multiply two numbers in Haskell using the Multiplication Operator:

```
main = do
     let var1 =2
     let var2 =3
     putStrLn "The Multiplication of the Two Numbers is:"
     print(var1 * var2)
```

This code will produce the following output, when you run it in our online platform:

```
The Multiplication of the Two Numbers is:
6
```

Division Operator

Take a look at the following code. It shows how you can divide two numbers in Haskell:

```
main = do
     let var1 = 12
     let var2 = 3
     putStrLn "The Division of the Two Numbers is:"
     print(var1/var2)
```

It will produce the following output:

```
The Division of the Two Numbers is:
4.0
```