Introduction to Functional Programming: Haskell

The Cardano blockchain is implemented using Plutus which is itself is just Haskell. For us to be able to develop smart contract we first have to understand how Haskell works. Haskell is a functional language meaning that it supports and encourages the functional programming style.

1. **Functional Programming**

Functional programming is style of programming in which the basic method of computation is the application of functions to arguments. This paradigm only uses *pure functions* (ones that avoid changing states) and *immutable values (meaning there are no variables just constants)*.

Example:

Summing the integers 1 to 100 in Python

sum = 0

for i in range(1,101):

sum = sum + i

print(sum)

The computation method is variable assignment. An imperative language uses a sequence of statements to determine how to reach a certain goal. These statements are said to change the state of the program as each one is executed in turn

Example:

Summing the integers 1 to 100 in Haskell

sum [1..100]

The only way of computation is through function application.

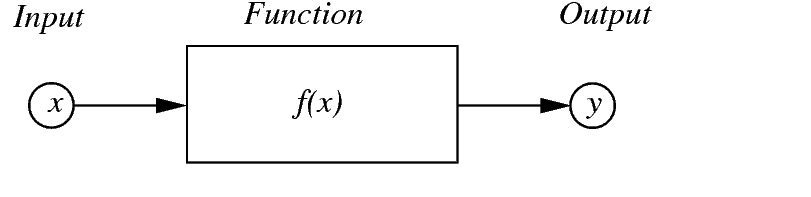
*So, what do I mean by Pure Functions?*

A pure function is one whose **output depends only on Input and has No Side Effects!**

1. This means a pure function will always result in the same output for the same input every time it is evaluated!
2. And does not rely on or modify anything outside its parameters.

This is a function in the purest mathematical sense. Math functions are rules that map an input to an output.

***f(x) => x2***



*So we can say FP* is a math-oriented way to write code, because these function are more like expressions which map input to output.

We consider that a function has a side effect if:

1. It modifies a mutable data structure or variable,
2. Uses IO,
3. Throws an exception or halts an error;

All of these things are considered side effects. Side effects are considered evil in this paradigm.

The reason why side effects are bad is that, if you had them, a function can be unpredictable because it can produce different results depending on factors external to the function; when a function has no side effects we can execute it anytime, it will always return the same result, given the same input.

But in the programming world we cannot overlook writing to files, reading from databases or reading from console. A program would make no sense if it doesn't talk to the outside world at some point of time! That's where IO monads come into picture, Monads encapsulate and handles side-effects in an elegant way.

**Advantages of Functional programming**

**Free from side effects and bugs**: Functions not having side effects means you can write code free from unpredictable bugs.

**Efficiency**: Functional programs consist of independent units (or functions) that can be run concurrently. This means the efficiency is higher. If there is no data dependency between two pure expressions, then their order can be reversed, or they can be performed in parallel and they cannot interfere with one another (in other terms, the evaluation of any pure expression is thread-safe)

**Lazy evaluation**: Program may postpone evaluating an expression until just-in-time when its value is needed. Leads to better optimization since we eliminate things we don’t have to execute.

**Nested functions**: Functional programming support nested functions.

**Easier to** **test and debug**, pure function depends upon the input and the algorithm and not on the outside world parameters.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

* **Referentially transparent-**

An expression is referentially transparent if it can be replaced with its value (and vice-versa) without changing the program's behavior. This requires that the expression be pure – its value must be the same for the same inputs and its evaluation must have no side effects.

Example:

The term "the capital of Kenya" refers to the city of Nairobi. That is a straightforward example of a "referent".

A context in a sentence is "referentially transparent" if replacing a term in that context by another term that refers to the same entity doesn't alter the meaning. For example

*The Kenyan Parliament meets in the capital of Kenya.*

means the same as

*The Scottish Parliament meets in Nairobi.*

* **Idempotent-**

For a give value *'a'* it will produce the same result no matter how many times it is called!

It is safe to evaluate the function any number of times.

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

1. **Haskell**

**Haskell** is a functional programming language, named after mathematician Haskell Brooks Curry, who made important contributions to (combinatory logic) mathematical logic that were used as the basis for functional programming.

The main characteristics of Haskell:

1. It is purely functional, which means that all functions written in Haskell are also functions in the mathematical sense the do not have side effects. The variables are immutable: cannot be changed by any expression. Haskell does not contain statements or instructions, just expressions that are evaluated.
2. It is lazy, meaning the expressions are evaluated when it is really necessary. Combined with the purity of Haskell, you can create chains of functions, which improves performance.
3. It is statically typed, which means that every expression has a type, established at compile time.

*Statically typed languages perform type checking at compile-time, while dynamically-typed languages perform type checking at run-time* *This means that scripts written in dynamically-typed languages (like Groovy) can compile even if they contain errors that will prevent the script from running properly (if at all). If a script written in a statically-typed language (such as Java) contains errors, it will fail to compile until the errors have been fixed.*

1. It enables type inference; this means you don't have to explicitly label every piece of code with a type because the type system can intelligently figure out a lot about it.

*If you say* a = 5 + 4*, you don't have to tell Haskell that* a *is a number, it can figure that out by itself.*

1. Haskell is concurrent because it works without side effects.
2. It has many open source packages.

From these features, you can identify some of the advantages of using Haskell as:

1. The quality of the code is high.
2. You can work with abstract mathematical concepts.
3. The type system is flexible.
4. Errors are kept to a minimum.
5. The syntax is optimized and well-designed.
6. It brings good performance due to concurrency.

Why Cardano used Haskell?! There are several reasons

1. **Haskell has an extremely high level of abstraction**: Which means that you can express complicated concepts in the language itself. That also means that it's much closer to mathematics than other languages. Why is this important?

2. **Haskell is very expressive and terse**: The code is usually smaller and more concise than the same implementations in other languages. There's less clutter. Which makes it easier to read, write, and verify.

3. **Haskell is perfect for writing domain-specific languages** (like Marlowe and Plutus): That's important because we need not only to write code for smart contracts but also for network protocols and other implementations.

4. **His declarative/functional nature facilitates formal verification**: This is extremely important when a failure means losses of millions or billions of dollars. (Like if you were developing a blockchain, per example.)

5. **Haskell is statically typed with a sophisticated type system**: That means that many bugs are already caught by the compiler while you write the code. There are many other statically typed programming languages (C, Java, etc), but Haskell has a much more powerful type system.

1. **Getting started:**

**GHC**

GHC stands for *Glasgow Haskell Compiler*.

It is the leading implementation of Haskell, and comprises a compiler and interpreter.

The interactive nature of the interpreter makes it well suited for teaching and prototyping;

GHC can be downloaded from <https://www.haskell.org/downloads#platform> .Or through nix shell which can be used to setup a Haskell Programming Environment.

*Nix* is a cross-platform package manager that utilizes a purely functional deployment model.

Haskell comes with an important tool called Cabal which is used for building and packaging Haskell libraries and programs.

You can also use [replit](https://replit.com)  (web-based IDE) to run Haskell code.

**Starting GHCi**

The interpreter can be started from the terminal command prompt by simply typing ghci:

$ ghci

GHCi, version X: http://www.haskell.org/ghc/ :? for help

Prelude>

The GHCi prompt prelude> means that the interpreter is now ready to evaluate an expression.

The prompt here is Prelude> but because it can get longer when you load stuff into the session, you can change it to ghci>. If you want to have the same prompt, just type :set prompt "ghci> “**.**

The can be used as a desktop calculator to evaluate simple numeric expresions like:

> 2+3

14

> (2+3)\*4

20

> sqrt (3^2 + 4^2)

5.0

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Haskell Script and Interactive Modes

GHC can take a Haskell script (they usually have a .hs extension) and compile it but it also has an interactive mode which allows you to interactively interact with scripts.

Interactively, you can call functions from scripts that you load and the results are displayed immediately. For learning it's a lot easier and faster than compiling every time you make a change and then running the program from the prompt.

The interactive mode is invoked by typing in ghci at your prompt. If you have defined some functions in a file called, say, myfunctions.hs, you load up those functions by typing in :l myfunctions and then you can play with them, provided myfunctions.hs is in the same folder from which ghci was invoked. If you change the .hs script, just run :l myfunctions again or do :r, which is equivalent because it reloads the current script.

The workflow we are going to use is defining some functions in a .hs file, loading it up and messing around with them and then changing the .hs file, loading it up again and so on.

**Starting Out**

The first thing we're going to do is run ghc's interactive mode and call some function to get a very basic feel for haskell. Open your terminal(can be on replit) and type in ghci.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Haskell Types**

A type is a kind of label that every expression has. It tells us in which category of things an expression fits. The expression True is a Boolean, “hello “ is a string, etc.

We can use GHCI to examine the types of some expressions. We'll do that by using the **:t** command which, followed by any valid expression, tells us its type.

It will print out the expression followed by **::** and its type signature.

**::** is read as "has type of".

ghci> :t 'a'

'a' :: Char

ghci> :t True

True :: Bool

ghci> :t "HELLO!"

"HELLO!" :: [Char]

ghci> :t (True, 'a')

(True, 'a') :: (Bool, Char)

ghci> :t 4 == 5

4 == 5 :: Bool

Note that Explicit types are always denoted with the first letter in capital case.

**Basic Types in Haskell**

The following are the Basic types in Haskell:

• Char represents a single Unicode character. Denoted with single quotes.

[Char] a type list of Char is equivalent to a String. Denoted with double quotes.

• Bool represents logical values. True or False.

• Int is an integer and represents a bounded type, which means it has a maximum value and a minimum value. These two values depend on the machine. For a 32-bit machine, the minimum value is -2147483648, and the maximum value is 2147483647.It is different for 64 bit. Values of this type have a fixed amount of memory being used for their

storage

• The Integer type represents integer values also (very large integers, such as those used in cryptography), but its type is not bounded. Values have as much memory as necessary thus avoiding the imposition of lower and upper limits on the range of numbers. The bounded Int is more efficient because most computers have built-in hardware for fixed-precision integers, whereas arbitrary-precision integers are usually processed using the slower medium of software, as sequences of digits..

• Float represents values with decimal points of single precision.

• Double represents values of floating-point numbers with double precision. Uses twice as many bits (or memory) as a regular floating-point number.

**Note** that a single number may have more than one numeric type. For example, the number 3 could be of type Int, Integer, Float or Double. So during type inferencing a type class Num is used instead of individual types. At this the moment, it is enough to know that Num includes Int, Integer, Double, and Float types.

Classes are not types; they are categories of types, which means that an instance of a class is a type.

Standard Classes

In Haskell, the most common standard classes are the following ones:

* **Num**
* **Eq** is used when you work with == (is equal) and /=(is not equal).
* **Ord** is used when you work with the operators <, <=, >,and >= and the functions min, max, and compare.
* **Enum** is used in enumerations and lets you use syntax such as [Red .. Yellow].
* **Read** includes the function read, which takes a string as a parameter and parses it into a value.
* **Show** includes the function show, which takes a value as a parameter and converts it to a string.
* **Bounded** is used in enumerations and includes the functions minBound and maxBound.

**Type Errors**

Applying a function to one or more arguments of the wrong type results in a type error.

1 + ‘a’

> error..

All type errors are found at compile time, which makes programs safer and faster by removing the need for type checks at run time.

**List Types**

A list is sequence of values of the same type.

Lists are denoted by square brackets and values separated by commas.

We write **[T]** for the type of all lists whose elements have type **T**.

Strings are just lists of characters. **"hello"** is just syntactic sugar for **['h','e','l','l','o']**

Lists can also contain lists. They can be of different lengths but can’t be of different types.

[1,2,3,4] : list of numbers :: Num a => [a]

[‘a’, ‘b’, ‘c’] : list of characters :: [char]

* The type definition of list does not include its size and its individual components.
* There are no restrictions on the type of the elements of a list, any basic types and non-basic types can be used.
* There is no restriction that a list must have a finite length.

## Operations on lists

**Get an element out of a list by index**, use !**!**. The indices start at 0.

ghci> [9.4,33.2,96.2,11.2,23.25] !! 1

33.2

**Put two lists together**: ++ operator, appends a list at the end of another. example:

ghci> [1,2,3,4] ++ [9,10,11,12]

[1,2,3,4,9,10,11,12]

**Put a list element at the beginning** of a list using the: operator example:

ghci> 5:[1,2,3,4,5]

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

Notice how**:** takes a number and a list of numbers or a character and a list of characters, whereas++ takes two lists.

**Comparing lists**: Lists can be compared if the stuff they contain can be compared. Comparison happens in lexicographical order. First the heads are compared. If they are equal then the second elements are compared, etc

ghci> [3,2,1] > [2,10,100]

True

**Ranges**: is a way of making lists that are sequences of elements that can be enumerated. Eg alphabet and numbers. To create a list of numbers from 1 to 20 use **[1..20]** , alphabets from K to Z **[K..Z]**

You can also specify a step.

ghci> [2,4..20]

[2,4,6,8,10,12,14,16,18,20]

It's simply a matter of separating the first two elements with a comma and then specifying what the upper limit is.

Watch out when using floating point numbers in ranges! Because they are not completely precise (by definition), their use in ranges can yield some pretty funky results. Avoid them.

You also create infinite lists: **[1..]**

Let's examine how you would get the first 24 multiples of 13. Sure, you could do **[13,26..24\*13]** . But there's a better way: take 24 **[13,26..].**

**List functions**

**head** takes a list and returns its head.

**tail** takes a list and returns its tail. In other words, it chops off a list's head.

**last** takes a list and returns its last element.

**init** takes a list and returns everything except its last element.

**Note**: When using head, tail, last and init, be careful not to use them on empty lists. This error cannot be caught at compile time so it's always good practice to take precautions against accidentally telling Haskell to give you some elements from an empty list.

**length** takes a list and returns its length.

**null** checks if a list is empty. returns **True** or **False reverse** reverses a list.

**take** takes number and a list. It extracts that many elements from the beginning of the list. **drop** works in a similar way, only it drops the number of elements from the beginning of a list. **maximum** returns the biggest element.

**minimum** returns the smallest.

**sum** takes a list of numbers and returns their sum. **product** takes a list of numbers and returns their product. **elem** tells us if a value is an element of the list.

**cycle** takes a list and cycles it into an infinite list. You have to slice it off somewhere.

ghci> take 10 (cycle [1,2,3])

[1,2,3,1,2,3,1,2,3,1]

**repeat** takes an element and produces an infinite list of just that element.

ghci> take 10 (repeat 5)

[5,5,5,5,5,5,5,5,5,5]

**replicate** also does same thing. **replicate 3 10 returns [10,10,10].**

List comprehension:

Offers a shorter syntax when you want to create a new list based on the values of an existing list

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

The right side of the vertical bar is the first to be executed. X is drawn from the list [1..10]

Then doubled in the expression on the right. The final value is then stored in a list.

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

This includes a predicate x\*2>=12 which must be satisfied first for x to enter the expression part.

Note that weeding out lists by predicates is also called **filtering.** We took a list of numbers and we filtered them by the predicate.

For convenience, we can put that comprehension inside a function so we can easily reuse it.

boomBangs xs = [ if x < 10 then "BOOM!" else "BANG!" | x <- xs, odd x]

We can also multiple values from two different lists in a list comprehension.

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

**Tuples types**

A tuple is a finite sequence of values of possibly different types.

It is denoted with parentheses and their components are separated by commas.

It is heterogenous: can contain a combination of several types.

We write **(T1,T2,...,Tn)** for the type of all tuples whose **ith** components have type **Ti** for any i in the range 1 to n.

(False,True) : tuple of 2 Boolean values.

(False,’a’,True) : tuple of bool, char and bool

* The type of a tuple encodes its size.
* The type of the components is unrestricted.

Tuples can also be used to represent a wide variety of data, since it can combine different data types.

Use tuples when you know in advance how many components some piece of data should have. Tuples are much more rigid because each different size of tuple is its own type, so you can't write a general function to append an element to a tuple— you'd have to write a function for appending to a pair, one function for appending to a triple, one function for appending to a 4-tuple, etc.

While there are singleton lists, there's no such thing as a singleton tuple they are not permitted because they would conflict with the use of parentheses to make the evaluation order explicit, such as in (1+2)\*3.

Like lists, tuples can be compared with each other if their components can be compared. Only you can't compare two tuples of different sizes.

Two useful functions that operate on pairs:

**fst** takes a pair and returns its first component.

ghci> fst (8,11)

8

**snd** takes a pair and returns its second component. Surprise!

ghci> snd (8,11)

11

A cool function that produces a list of pairs: zip. It takes two lists and then zips them together into one list by joining the matching elements into pairs.

ghci> zip [1,2,3,4,5] [5,5,5,5,5]

[(1,5),(2,5),(3,5),(4,5),(5,5)]

What happens if the lengths of the lists don't match? The longer list simply gets cut off to match the length of the shorter one.

**Haskell operations**

---------- Arithmetic operations ----------

ghci> 2 + 15

17

ghci> 49 \* 100

4900

ghci> 1892 - 1472

420

ghci> 5 / 2

2.5

We can also use several operators on one line and all the usual precedence rules are obeyed. We can use parentheses to make the precedence explicit or to change it.

ghci> 50 \* 100 - 4999

1

ghci> 50 \* (100 - 4999)

-244950

**Negative number**, has to always be surrounded by parentheses.

Doing **5 \* -3** will make GHCI yell at you but doing **5 \* (-3)** will work just fine.

**Built in math functions**

ghci> pi

ghci> exp 9

ghci> log 9

ghci> 9 \*\* 2

ghci> truncate 9.999

ghci> round 9.999

ghci> ceiling 9.999

ghci> floor 9.999

ghci> sum [1 ..10]

ghci> sum [1 ..10]

Also sin, cos, tan, asin, atan, acos, sinh, tanh, cosh, asinh, atanh, acosh

---------- Boolean Operators ----------

is also pretty straightforward. As you probably know:

&& means a Boolean and,

|| means a Boolean or.

not negates a True or a False.

ghci> True && False

False

ghci> True && True

True

ghci> False || True

True

ghci> not False

True

ghci> not (True && True)

False

---------- equality Operators----------

== is equal to

/= is not equal to

ghci> 5 == 5

True

ghci> 1 == 0

False

ghci> 5 /= 5

False

ghci> 5 /= 4

True

ghci> "hello" == "hello"

True

---------- Comparison operators----------:

**<** is less than**,**

**<=** is less than or equal to**,**

**>** is greater than

**>=** is greater than or equal to

There is an expectation that the left and right side have to be of same data type for these arithmetic operations to work.

Note: you can do **5 > 4.0** or 5 + 4.0 because **5** is sneaky and can act like an integer or a floating-point number. **4.0** can't act like an integer, so **5** is the one that has to adapt.

**Defining Your Own Types**

Suppose you want to create a structure that simulates a date. You need three integer values corresponding to the day, the month, and the year. You can define it using the data keyword, as shown here:

Prelude> data DateInfo = Date Int Int Int

Prelude> myDate = Date 1 10 2018

Prelude> :t myDate

myDate :: DateInfo

Prelude> :info DateInfo

data DateInfo = Date Int Int Int -- Defined at

<interactive>:101:1

DateInfo is the name of your new type, and it is called a type constructor, which is used to refer to the type. The Date after the equal sign is the value constructor (or data constructor), which is used to create values of DateInfo type. The three Ints after Date are components of the type. Note that the name of the type constructor and the name of the value constructor begin with capital letters.

If you want to print myDate, you would get an error, because the print function does not have an argument of type DateInfo. To print myDate, you need to modify the definition of DateInfo, adding deriving (Show) to the end of definition (we will talk in detail about deriving in Chapter 10). The procedure is shown here:

Prelude> print myDate

<interactive>:34:1: error:

• No instance for (Show DateInfo) arising from a use of 'print'

-- Defined at <interactive>:27:44

• In the expression: print myDate

In an equation for 'it': it = print myDate

Prelude> data DateInfo = Date Int Int Int deriving (Show)

Prelude> myDate = Date 1 10 2018

Prelude> print myDate

Date 1 10 2018

The comparison between two dates will give a similar error. You need to add Eq after Show, which tells the compiler that you allow comparison between two dates.

Prelude> data DateInfo = Date Int Int Int deriving (Show, Eq)

Prelude> myDate1 = Date 1 10 2018

Prelude> myDate2 = Date 15 10 2018

Prelude> myDate3 = Date 1 10 2018

Prelude> myDate2 == myDate1

False

Prelude> myDate3 == myDate1

True

Another example studentInfo type:

Prelude> data StudentInfo = Student string DateInfo string deriving (Show, Eq)

Prelude> student\_1 = Student "Alice Brown" (Date 21 8 1992)

"Computer Science"

Prelude> :t student

student :: StudentInfo

**Records**

Let’s suppose you want to add more information in your StudentInfo type.(this time for simplicity let’s call it Student) The student will be described by first name, last name, birth date, specialization, study year, and average grade. The Student type looks like this:

Prelude> data Student = Student String String DateInfo String Int Float deriving (Show, Eq)

Note The type constructor now has the same name as the dataconstructor. Haskell allows you to do this.

Another way other than type to make it more intuitive is to make it a record, as shown here:

Prelude> :{

Prelude| data Student = Student { firstName :: String

Prelude| , lastName :: String

Prelude| , birthDate :: DateInfo

Prelude| , specialization :: String

Prelude| , studyYear :: Int

Prelude| , averageGrade :: Float

Prelude| } deriving (Show, Eq)

Prelude| :}

Prelude> student\_1 = Student "Emily" "Brian" (Date 23 4 1990) "Computer Science" 2 74

In this piece of code, you can see :{ and :}. This means you are writing a command on multiple lines. You can use this just in GHCi, not in .hs files.

Using record, you can easily access a field of the structure just by typing the name of the field followed by the name of the variable.

Prelude> firstName student\_1

"Emily"

Prelude> averageGrade student\_1

9.14

Prelude> :t averageGrade

averageGrade :: Student -> Float

**Type System**

In Haskell, the system has the following characteristics:

• **Strong type**: A strong type system ensures that the program will contain errors resulting from wrong expressions. An expression that meets all the conditions of a language is called well-typed; otherwise, it is ill-typed and will lead to a type error. In Haskell, strong typing does not allow automatic conversions. So, if a function has a Double argument but the user provides an Int parameter, then an error will occur. Of course, the user can explicitly convert the Int value to a Double value using the predefined conversion functions and everything will be fine.

• **Static type**: In a static type system, the types of all values and expressions are known by the compiler at compile type, before executing the program. If something is wrong with the types of an expression, then the compiler will tell you, as in the example of lists. Combining strong and static types will avoid runtime errors.

• **Inference type**: In an inference type system, the system recognizes the type of almost all expressions in a program. Of course, the user can define explicitly any variable, providing its type, but this is optional.

**Function**

1. **We've been using functions all along.**

For instance, **\*** is a function that takes two numbers and multiplies them. We call it by sandwiching it between numbers, this type of function is called infix function. Most functions that aren't used with numbers are prefix functions.

Example:

mod is a prefix operator

ghci> mod 5 4

With back ticks we can use it as an infix operator

ghci>5 `mod` 4

**Note**: this is back quotes, not forward

1. **Function Application**

In Haskell, function application is denoted using space.

**func a**

A function with one parameter **a**.

**func2 a b**

A function with two parameters **a** and **b**

1. **Functions also have types.**

A function is a mapping from arguments of one type to results of another type.

even :: Int -> Bool

It is a good practice to preceding function definitions by their types as it serves as useful documentation. Any such types provided manually by the user are checked for consistency with the types calculated automatically using type inference.

You can also check on the type of the function by using :t.

addThree :: Int -> Int -> Int -> Int

addThree x y z = x + y + z

ghci>:t (+)

Num a => a -> a -> a

ghci>:t sqrt

sqrt :: Floating a => a -> a

**The main function**

The main function serves **as the starting point for program execution**. It usually controls program execution by directing the calls to other functions in the program. A program usually stops executing at the end of main, although it can terminate at other points in the program for a variety of reasons

**User defined Functions**

As well as the functions in the standard library, you can also define your own functions;

New functions are defined within a script, a text file comprising a sequence of definitions;

By convention, Haskell scripts usually have a .hs suffix on their filename. This is not mandatory, but is useful for identification purposes.

Create a Haskell script practice\_1.hs

double x = x + x

quadruple x = double (double x)

Leaving the editor open, in another window start up GHCi with the new script:

$ ghci practice\_1.hs

Or

You can launch ghci on your terminal and load the script with :load haskell.hs.

Now both the standard library and the file test.hs are loaded, and functions from both can be used:

> quadruple 10

40

> take (double 2) [1,2,3,4,5,6]

[1,2,3,4]

Leaving GHCi open, return to the editor, add the following two definitions, and resave:

factorial n = product [1..n]

average ns = sum ns `div` length ns

GHCi does not automatically detect that the script has been changed, so a reload command must be executed before the new definitions can be used:

> :reload

> factorial 10

3628800

**Useful GHCi Commands**

Command Meaning

:load *name* load script *name*

:reload reload current script

:set editor *name* set editor to *name*

:edit *name* edit script *name*

:edit edit current script

:type *expr* show type of *expr*

:? show all commands

:quit quit GHCi

**Rules for Naming functions**

Function and argument names must begin with a lower-case letter and a camelCaseNotation is usually preferred.

For example:

myFunc x

By convention, **list** arguments usually have an s suffix on their name. For example:

xs , ns, xss

Definitions of same levels should be lined up at the same indentation to avoid the need for explicit syntax (like brackets and curly braces) to indicate a grouping.

a = b + c

where

b = 1

c = 2

d = a \* 2

**Functions with multiple arguments**

Because there are no restrictions on the types of the arguments and results of a function, the simple notion of a function with a single argument and a single result is already sufficient to handle the case of multiple arguments and results, by packaging multiple values using lists or tuples.

add :: (Int,Int) -> Int

add (x,y) = x+y

zeroto :: Int -> [Int]

zeroto n = [0..n]

**Curried functions**

Functions with multiple arguments can also be handled in another perhaps less obvious way by exploiting the fact that functions are free to return functions as results.

add’ :: Int -> (Int -> Int)

similar to: add’ :: Int -> Int -> Int

add’ x y = x+y

add’ takes an integer x and returns a function (int -> int), which in turn takes an integer y and returns the integer result x+y.

These kind of functions that take their arguments one at a time are called **curried functions.**

The advantage of curried functions is that they are more flexible than functions on tuples, because useful functions can often be made by partially applying a curried function with less than its full complement of arguments.

---------- Importing modules ----------

A Haskell module is a collection of related functions, types and typeclasses.

import Data.List :

import System.IO

---------- Haskell comments ----------

-- single line comment

{-

Beginning of

multiline comment

-}