Haskell Programming Language

Seyed Ali Nedaaee Oskoee

Spring 2025

1- Introduction

Haskell is a purely functional, statically typed, and lazy programming language that is widely used due to its power and flexibility in various domains, including web development, data science, artificial intelligence, and finance.

Advantages of Haskell:

- **Purely Functional:**

- Functions in Haskell have no side effects, meaning they always produce the same output for the same inputs. This makes code easier to understand and test, and helps prevent bugs caused by unintended changes in program state.

- **Strong Type System:**
  - Haskell features a strong, static type system that prevents many runtime errors. This type system allows the compiler to catch numerous bugs before the program is even executed.
- **Laziness:**
  - Haskell is a lazy language, which means expressions are not evaluated until their values are actually needed. This enables the definition of infinite data structures and can lead to performance improvements.
- **High Power and Flexibility:**
  - Haskell is a powerful and flexible language capable of solving a wide range of problems. It provides strong tools for abstraction and code composition.
- **Strong Community:**
  - Haskell has an active and passionate community that offers a wealth of resources and libraries for developers.

Disadvantages of Haskell:

- **Difficult to Learn:**
  - Haskell includes many advanced concepts, making it potentially challenging for beginner programmers to learn.
- **Relatively Lower Performance in Some Cases:**
  - Due to its laziness and purely functional nature, Haskell's performance in certain cases might be lower compared to imperative languages.
- **Fewer Libraries Compared to Some Languages:**
  - In some areas, Haskell has fewer libraries available compared to languages like Python and Java.

Differences Between Haskell and Racket:

- **Programming Paradigm:**
    - Haskell is a purely functional language, whereas Racket is a multi-paradigm language that supports functional, imperative, and object-oriented programming.
- **Type System:**
    - Haskell has a static and strong type system, while Racket is dynamically typed.
- **Laziness:**
    - Haskell is lazy, while Racket is eager in its evaluation.

- **Applications:**
    - Haskell is mostly used in areas like web development, data science, and artificial intelligence. In contrast, Racket is commonly used in education, language development, and scripting.
- **Syntax:**
    - Haskell's syntax is quite different from Racket's and is closer to mathematics, while Racket's syntax is similar to Lisp family languages.

Summary:

Haskell is a powerful and flexible language well-suited for developing complex and reliable applications. However, it has a relatively steep learning curve, and in some scenarios, it may perform less efficiently than imperative languages. On the other hand, Racket is a dynamic, multi-paradigm language suitable for education, programming language development, and scripting tasks.

2- Installing Haskell

Installation steps for Haskell vary depending on your operating system:

2-1 Windows

1. **Download:**
    - Go to the official Haskell website: haskell.org
    - Download the GHCup installer. GHCup is a tool that simplifies the installation and management of different versions of GHC (the Haskell compiler) and related tools.
2. **Install GHCup:**
    - Run the downloaded file and follow the installation steps.
    - GHCup will automatically install GHC and the required tools.
3. **Verify Installation:**
    - Open the Command Prompt.
    - type the command ghci.

2-2 macOS:

1. **Download:**
    - Go to the official Haskell website: haskell.org
    - Download the GHCup installer.
2. **Install GHCup:**
    - Run the installer and follow the on-screen instructions.
    - GHCup will automatically install GHC and other necessary tools.
3. **Verify Installation:**
    - Open Terminal.
    - Enter ghci.
    - If installed correctly, GHCi will start.

2-3 Linux (Ubuntu/Debian):

1. **Install GHCup:**
    - Open a terminal window.
    - Run the following commands to update your system and install GHC:

```
sudo apt-get update
sudo apt-get install ghc
```

    - Follow the instructions to complete the installation.
2. **Verify Installation:**
    - After installation, run ghci in the terminal.
    - If successful, the GHCi interpreter will launch.
3. **Runs** :

```
ghc example.hs
./example
```

3- Haskell Tutorial

3-1 Variables and Data Types

3-1-1 Integer:

```haskell
x :: Int
x = 42
main :: IO ()
main = print x
```

3-1-2 Double and Float:

```haskell
piValue :: Float
piValue = 3.1415
main :: IO ()
main = print piValue
```

```haskell
eValue :: Double
eValue = 2.718281828459045
main :: IO ()
main = print piValue
```

3-1-3 Rational or Fractional Numbers:

```haskell
import Data.Ratio
half :: Rational
half = 1 % 2
main :: IO ()
```

```
main = print half -- output 1 % 2
```

3-1-4 Boolean:

```
bool :: Bool
bool = True
main :: IO ()
main = print bool
```

3-1-5 Char and String:

```
letter :: Char
letter ='A'
main :: IO ()
main = print letter
```

```
message :: String
message = "Hello World!"
main :: IO ()
main = print message
```

3-1-7 Lists:

```
numbers :: [Int]
numbers = [1, 2, 3, 4, 5]
main :: IO ()
main = print numbers
```

3-1-7 Tuples:

```
person :: (String, Int)
person = ("Ali", 25)
main :: IO ()
main = print person
```

3-1-8 The Maybe Type (Unknown or Optional Values):

```haskell
safeDivide :: Double -> Double -> Maybe Double
safeDivide _ 0 = Nothing
safeDivide x y = Just (x / y)
result1 = safeDivide 10 2  -- output: Just 5.0
result2 = safeDivide 10 0  -- output: Nothing
main :: IO ()
main = print (result1,result2)
```

- It is used to handle unknown or undefined values (for example, division by zero.

3-1-9 The Either Type (Handling Success and Error):

```haskell
divide :: Double -> Double -> Either String Double
divide _ 0 = Left "Error: Division by zero"
divide x y = Right (x / y)
result1 = divide 10 2  -- مقدار Right 5.0 را برمی‌گرداند
result2 = divide 10 0  -- مقدار Left "Error: Division by zero" را برمی‌گرداند
main :: IO ()
main = print (result1,result2)
```

3-2 Operators:

3-2-1 Arithmetic Operators

Operator	Description	Example	Output
+	Addition	2 + 3	5
-	Subtraction	7 - 2	5
*	Multiplication	4 * 3	12
/	Decimal Division	10 / 4	2.5

`div`	Integer Division	`10 'div' 3`	`3`
`mod`	Remainder of Division	`10 `mod` 3`	`1`
`^`	Integer Exponentiation	`2 ^ 3`	`8`

Example:

```haskell
main :: IO ()
main = print (5 + 3 * 2 - 4)  -- output: 7
```

- div works only for integers (Int), while / is used for decimal numbers (Float and Double).

3-2-2 Comparative Operators:

Operator	**Description**	**Example**	**Output**
==	Equal to	5 == 5	True
/=	Not equal to	3 /= 5	True
<	Less than	3 < 5	True
>	Greater than	5 > 3	True
<=	Less than or equal to	7 <= 4	False
>=	Greater than or equal to	7 >= 7	True

Example:

```haskell
main :: IO ()
main = print (10 > 5)  -- output: True
```

3-2-3 Logical Operators:

Operator	Description	Example	Output
&&	And	True && False	False
||	Or	True || False	True
not	Not	not True	False

Example:

```haskell
main :: IO ()
main = print (True && (not False))  -- output: True
```

3-2-4 even :

The even function is a predefined function used to check if an integer is even.

```haskell
even 4  -- output: True
even 7  -- output: False
even 0  -- output: True
even (-2) -- output: True


sumEven :: [Int] -> Int
sumEven xs = sum [x | x <- xs, even x]


sumEven [1, 2, 3, 4, 5, 6]  -- output: 12
```

3-2-5 odd :

This function works the opposite of even and checks if an integer is odd.

```
odd 5  -- output: True
odd 8  -- output: False
odd (-3) -- output: True


sumOdd :: [Int] -> Int
sumOdd xs = sum [x | x <- xs, odd x]


sumOdd [1, 2, 3, 4, 5, 6]  -- output: 9
```

3-2-6 isDigit :

This function is used to check whether a character is a digit or not.

```
isDigit '5'  -- output: True
isDigit 'a'  -- output: False
isDigit ' '  -- output: False

countDigits :: String -> Int
countDigits xs = length [x | x <- xs, isDigit x]

countDigits "abc123def456"  -- output: 6
```

3-2-7 isAlpha :

This function is used to check whether a character is an alphabet letter or not.

```haskell
isAlpha 'A'   -- output: True
isAlpha '7'   -- output: False
isAlpha ' '   -- output: False


countLetters :: String -> Int
countLetters xs = length [x | x <- xs, isAlpha x]


countLetters "abc123def456"  -- output: 6
```

3-3 Lists and Their Functions:

A list in Haskell is a linked collection of elements of the same type. Lists are defined using [ ] and can contain any number of elements. All elements in a list must be of the same type.

```haskell
numbers = [1, 2, 3, 4, 5]        -- A list of integers.
chars = ['a', 'b', 'c']          -- A list of Characters.
wordsList = ["hello", "world"]   -- A list of Strings.
```

3-3-1 Adding an Element to a List:

```haskell
newList = 0 : [1, 2, 3]  -- output: [0,1,2,3]
```

3-3-2 Combining Two Lists:

```haskell
list1 = [1, 2, 3]
list2 = [4, 5, 6]
combined = list1 ++ list2  -- [1,2,3,4,5,6]
```

3-3-3 Accessing an Element in a List:

```haskell
myList = [10, 20, 30, 40]
element = myList !! 2  -- output: 30
```

3-3-4 Important List Functions in Haskell:

Function	Description	Example	Output

head	Returns the first element of a list.	head [1,2,3]	1
tail	Returns all elements except the first one.	tail [1,2,3]	[2,3]
last	Returns the last element of a list.	last [1,2,3]	3
init	Returns all elements except the last one.	init [1,2,3]	[1,2]
length	Returns the number of elements in a list.	length [1,2,3]	3
null	Checks if the list is empty.	null []	True
reverse	Reverses the order of elements in a list.	reverse [1,2,3]	[3,2,1]
take n	Returns the first n elements of a list.	take 2 [1,2,3]	[1,2]
drop n	Removes the first n elements of a list.	drop 2 [1,2,3]	[3]

3-3-5 Infinite Lists in Haskell:

Haskell allows us to create infinite lists! (These lists are only evaluated when used).

```
infiniteList = [1..]    -- An infinite list from 1 to infinity
first10 = take 10 infiniteList  -- [1,2,3,4,5,6,7,8,9,10]
```

3-3-6 concat :

The concat function is used to combine lists that are themselves lists. It flattens a list of lists into a single list.

```
concat [[1, 2], [3, 4], [5, 6]]  -- [1,2,3,4,5,6]
```

3-3-7 intercalate :

The intercalate function is used to insert a list as a separator between elements of other lists.

```
import Data.List
intercalate [0] [[1, 2], [3, 4], [5, 6]]  -- [1,2,0,3,4,0,5,6]
```

3-3-8 splitAt :

The splitAt function splits a list into two parts at a given index..

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

3-3-9 elem :

The elem function checks if an element is present in a list.

```
elem 2 [1, 2, 3]  -- output: True
elem 5 [1, 2, 3]  -- output: False
```

3-3-10 find :

The find function returns the first element that satisfies a given condition.

```
import Data.List
find (> 3) [1, 2, 3, 4, 5]  -- output: Just 4
```

3-3-11 filter :

The filter function filters the elements of a list based on a given condition, returning only those that satisfy the condition.

```
filter even [1, 2, 3, 4]  -- [4,2]
```

3-3-12 partition :

The partition function splits a list into two parts based on a given condition: one part contains the elements that satisfy the condition, and the other contains the elements that do not.

```
import Data.List
partition even [1, 2, 3, 4]  -- ([2,4],[1,3])
```

3-3-13 foldl :

The foldl function (fold left) combines the elements of a list using a function and an initial value, processing the list from left to right.

```
foldl (+) 0 [1, 2, 3]  -- output: 6
```

3-3-14 foldr :

The foldr function (fold right) is similar to foldl, but it processes the list from right to left.

```
foldr (+) 0 [1, 2, 3]  -- output: 6
```

3-3-15 sum :

The sum function calculates the sum of all elements in a list.

```
sum [1, 2, 3]  -- output: 6
```

3-3-16 product :

The product function calculates the product (multiplication) of all elements in a list.

```
product [1, 2, 3]  -- output: 6
```

3-3-17 concatMap:

The concatMap function is similar to map, but it concatenates the results of applying a function to each element.

```
concatMap (\x -> [x, x+1]) [1, 2, 3]  -- [1,2,2,3,3,4]
```

3-3-18 zip :

The zip function combines two lists into a list of tuples, pairing corresponding elements from both lists.

```
zip [1, 2, 3] ["a", "b", "c"]  -- [(1,"a"),(2,"b"),(3,"c")]
```

3-3-19 zipWith :

The zipWith function is similar to zip, but it applies a custom function to combine the elements from two lists.

```
zipWith (+) [1, 2, 3] [4, 5, 6]  -- [5,7,9]
```

3-3-20 unzip :

The unzip function splits a list of tuples into two separate lists, one containing the first elements of the tuples and the other containing the second elements.

```
unzip [(1, "a"), (2, "b"), (3, "c")]  --: ([1, 2, 3], ["a", "b",
"c"])
```

3-4 Tuples and Their Functions in Haskell:

In Haskell, a **tuple** can contain values of different types. Tuples are enclosed in parentheses ( ), and unlike lists, the number of elements in a tuple is fixed and cannot be changed. Additionally, unlike lists, a tuple can hold values of various types, whereas a list can only contain elements of the same type.

Example:

```
+person = ("Ali", 25)  -- A tuple contain one String and Integer
point = (3, 4)         -- A tuple contain two Integer
```

3-4-1 Accessing Elements of a 2-Element Tuple:

For 2-element tuples in Haskell, you can use the fst and snd functions to access the first and second elements, respectively.

```
myTuple = ("Apple", 5)
firstElement = fst myTuple    -- خروجی: "Apple"
secondElement = snd myTuple   -- 5 :خروجی
```

3-4-2 Multi-element tuples:

Tuples with more than two values are processed using pattern matching.

```
personInfo :: (String, Int, Bool) -> String
personInfo (name, age, isStudent) = name ++ " is " ++ show age ++
" years old."

main :: IO ()
main = print (personInfo ("Ali", 25, True))
-- output: "Ali is 25 years old."
```

3-5 Functional Programming Principles:

3-5-1 Functions and Their Definition:

In Haskell, functions are defined as follows

```
sumNumbers :: Int -> Int -> Int -- Function signature (input and
output types )
sumNumbers x y = x + y            -- تعریف تابع
main = print (sumNumbers 3 4)  -- output: 7
```

3-5-2 Parameters and Function Outputs:

Functions in Haskell can have no input, multiple inputs, or process a list of values.

3-5-2-1 Function Without Parameters:

```
greet :: String
greet = "Hello, Haskell!"
main = print greet  -- output: "Hello, Haskell!"
```

3-5-2-2 Function with List Input:

```
sumList :: [Int] -> Int
sumList xs = sum xs
```

```
main = print (sumList [1,2,3,4,5])  -- output: 15
```

3-5-3 Immutability Concept:

In Haskell, variable values cannot be changed after they are initialized. A code like x = x + 1 is impossible! This feature makes Haskell programs safer and free of side effects.

3-5-4 Recursion:

Haskell does not support traditional loops like `for`. Instead, it uses recursion for repetition.

Example: Factorial function with recursion:

```
factorial :: Int -> Int
factorial 0 = 1  -- Base case
factorial n = n * factorial (n - 1)
main = print (factorial 5)  -- output : 120
```

3-5-5 Higher-Order Functions:

These are functions that take other functions as arguments or return a function.

Example: A function that processes two numbers but takes the type of operation as input.

```
applyFunction :: (Int -> Int) -> Int -> Int
applyFunction f x = f x
double x = x * 2
square x = x * x
```

```
main = do
    print (applyFunction double 5)  -- output: 10
    print (applyFunction square 4)  -- output: 16
```

3-5-6 Sorting Functions (map, filter, fold):

- **map** → Applies a function to each element of a list.
- **filter** → Keeps only the elements that satisfy a specific condition.
- **fold** → Generates a final value from a list.

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

doubles = map (*2) numbers       -- [2,4,6,8,10]
evens = filter even numbers      -- [2,4]
sumNumbers = foldl (+) 0 numbers  -- 15

main = do
    print doubles
    print evens
    print sumNumbers
```

3-6 Types and Strong Type System in Haskell:

3-6-1 Static Typing:

Every value has a specific type, and types are checked at compile time.

```
x :: Int -- x can only be an integer.
x = 42
```

3-6-2 Type Inference:

Haskell can automatically infer the type of a variable, even if you don't explicitly specify it.

```haskell
x = 10      -- Haskell automatically          x       Int
y = "Hi"    -- Haskell knows that y is a String.
```

3-6-3 Defining New Types (type and data):

You can define new types. These types are just aliases for existing types.

```haskell
type Name = String
type Age = Int

personInfo :: Name -> Age -> String
personInfo name age = name ++ " is " ++ show age ++ " years old."

main = print (personInfo "Ali" 25)
```

3-6-4 Pattern Matching:

Pattern matching examines the input data and performs different actions based on its value.

```haskell
describeNumber :: Int -> String
describeNumber 0 = "Zero"
describeNumber 1 = "One"
describeNumber _ = "Other"  -- The value _ means "anything else".

main = print (describeNumber 1)  -- output: "One"
```

3-6-5 Derived Types:

You can define more complex types using data.

```haskell
data Color = Red | Green | Blue deriving Show

favoriteColor :: Color -> String
favoriteColor Red = "You like Red!"
favoriteColor Green = "You like Green!"
favoriteColor Blue = "You like Blue!"

main = print (favoriteColor Red)
```

3-6-6 Records in Haskell:

Records are structures similar to objects in other languages.

```haskell
data Person = Person { name :: String, age :: Int } deriving Show

main = do
    let p = Person { name = "Ali", age = 25 }
    print (name p)  -- output : "Ali"
    print (age p)   -- output : 25
```

3-7 Conditional Statements:

3-7-1 if-then-else Expression:

This expression is similar to conditional statements in other programming languages. Its structure is as follows:

```haskell
if condition then expression1 else expression2
```

Example:

```haskell
absoluteValue :: Int -> Int
absoluteValue x = if x >= 0 then x else -x
```

3-7-2 Guards:

Guards are a more powerful way to write conditional expressions. Using guards, you can check multiple conditions sequentially and execute the corresponding expression based on the first true condition. The structure of guards is as follows:

```haskell
functionName argument1 argument2 ...
  | condition1 = expression1
  | condition2 = expression2
  | otherwise = expression3
```

Example :

```haskell
compareNumbers :: Int -> Int -> String
compareNumbers x y
  | x > y = "x is greater than y"
  | x < y = "x is less than y"
  | otherwise = "x is equal to y"
```

3-7-3 Pattern Matching:

Pattern matching is a powerful method for deconstructing data and selecting different expressions based on data patterns. It can be used for implementing conditional expressions based on different input patterns:

```haskell
describeList :: [a] -> String
describeList [] = "The list is empty"
describeList [x] = "The list has one element"
describeList xs = "The list has multiple elements"
```

3-8 Example Projects in Haskell:

3-8-1 Generate First n Prime Numbers:

```haskell
is_prime_rec :: Int -> Int -> Bool
is_prime_rec number index=
    if (index == ((round(sqrt (fromIntegral number))) + 1)) then
True
    else
        if ((mod number index) == 0) then False
        else (is_prime_rec number (index + 1))

is_prime :: Int -> Bool
is_prime number
    | number < 2 = False
    | number == 3 = True
    | number == 5 = True
    | number == 4 = False
    | otherwise = (is_prime_rec number 2)
```

```haskell
gen_prime_rec :: Int -> Int -> [Int] -> [Int]
gen_prime_rec number index list =
    if (index >= number) then list
    else
        if (is_prime index) then (gen_prime_rec number (index + 1)
(index:list))
        else (gen_prime_rec number (index + 1) (list))

gen_prime :: Int -> [Int]
gen_prime number =
    (gen_prime_rec number 2 [])


main = do
    print (gen_prime 2)
    print (gen_prime 3)
    print (gen_prime 4)
    print (gen_prime 30)
    print (gen_prime 100)
```

3-8-2 Quicksort:

```haskell
et_samller:: Int -> [Int] -> [Int]
get_samller number [] = []
get_samller number (first:list) =
    if (first < number) then first:(get_samller number list)
    else (get_samller number list)

get_greater:: Int -> [Int] -> [Int]
get_greater number [] = []
get_greater number (first:list) =
    if (first >= number) then first:(get_greater number list)
    else (get_greater number list)

quick_sort :: [Int] -> [Int]
quick_sort [] = []
quick_sort (pivot:list) =
    quick_sort (get_samller pivot list) ++ [pivot] ++ (get_greater
pivot list)
```

```
main = do
    print (quick_sort [1])
    print (quick_sort [2,7,3,4,6,5,1])
    print (quick_sort [10,1,2,3,4,5])
```

3-8-3 Finding Tree Size:

```
data Tree a =
    Leaf |
    Node a (Tree a) (Tree a)
    deriving Show

tree :: Tree Int
tree = (Node 3 (Node 4 Leaf (Node 5 Leaf Leaf)) Leaf)

leaf_size:: Tree a -> Int
leaf_size Leaf = 1
leaf_size (Node value left rigth) =
    (leaf_size left) + (leaf_size rigth)

main = print(leaf_size tree)
```

4. Resources:

4-1 For more examples, you can check this GitHub repository.

4-2 Official Haskell Documentation on the official website.