Declarative Programming Language HASKELL

a brief look into its history and syntax

Author
Penek Suksuda
Student ID : 6634445623

Supervisor
Chotiros Surapholchai

Department of Mathematics and Computer Science
Chulalongkorn University

Preface

This report has been prepared as part of the 2301475 Logic and Functional Programming course for the 2025/1 academic term at Chulalongkorn University. The primary objective of this document is to explore Haskell, a purely functional programming language with lazy evaluation that differs significantly from imperative languages like Python or Java.

The content of this report covers the brief history of Haskell as a language from Haskell Standards to the modern Glasgow Haskell Compiler (GHC). It also covers basic syntax and semantics of the Haskell language, including its types, function definitions and common syntactic sugars that are unique to the language. Additionally, I have included a practical component: a basic application that applies these principles to Music Theory. By representing musical scales mathematically, It is evident how functional logic can simplify complex transposition tasks. I hope this report proves useful for anyone interested in getting started with Haskell or in the intersection of mathematics, programming and music.

Penek Suksuda
14 November, 2025

Table of contents

Introduction

The Official Logo of the Haskell Language

“Haskell is a purely functional programming language that features referential transparency, immutability and lazy evaluation.” — haskell.org

Haskell Standards

In September of 1987, a meeting was held at the conference on Functional Programming Languages and Computer Architecture (FPCA ‘87) in Portland, Oregon. The meeting was to address an unfortunate reality in the functional programming languages world where more than a dozen non-strict, purely functional programming languages existed at the time with no open standard. This was believed to have stifled adoption rates, considering Miranda, a lazy purely functional programming language widely used in this area, was proprietary. The meeting concluded that a committee should be formed to address such problems and declare an open standard that is to be a language named after the logician Haskell B. Curry, whose work serves as a basis for the language. The first version of Haskell was defined in 1990, named “Haskell 1.0”. After that, several iterations of the language were done, adding in new features like monadic IO and others to replace the old controversial way of doing things. This culminated in Haskell 98, the first “stable, minimal and portable” version with Standard Libraries defined, Monadic IO and Module System standardized. It was a stable base language suitable for teaching and research. Haskell Prime was introduced later and envisioned as a regularly updated standard with only one official version released, Haskell 2010. Currently in 2025, there are no official Haskell standards released after Haskell 2010.

Glasgow Haskell Compiler (GHC)

Glasgow Haskell Compiler (GHC) is currently the main, most advanced and most widely used Haskell Compiler in the world. After Haskell 2010, new versions are essentially GHC extensions, some may refer to them as GHC Haskell. Even though Haskell has formal standards (Haskell 98, Haskell 2010), GHC is the implementation that everybody uses. It was developed originally by Glasgow University (Simon Peyton Jones & team), but is now maintained by a large open-source team. Apart from compiling Haskell, GHC also offers several other tools and helpers with the notable one being ghci, an interactive REPL used constantly in learning and development by students and developers around the world.

Syntax & Expressions

Types

Primitive Types

Primitives Types are built-in value types that represent singular pieces of data.

- `Int     : Fixed-size whole numbers (e.g. 64-bit)`
- `Integer : Arbitrary-precision whole numbers`
- `Float   : Single-precision floating-point numbers`
- `Double  : Double-precision floating-point numbers`
- `Bool    : Boolean values (e.g. True, False)`
- `Char    : A single Unicode character`

Composite Types

Composite Types are structures that build new types out of existing ones.

- `Lists  [] : Homogeneous e.g. [Int], [Double], [Char]`
- `Tuples () : Heterogeneous e.g. (Integer, Int, String)`

`Note: String is just a type synonym to [Char]`

- `Functions -> : Arrow is type constructor for function`

`Note: All functions are curried by default`

`E.g. Int -> Int -> Bool`

Algebraic Data Types (ADTs)

Algebraic Data Types (ADTs) are a combination of two basic mathematical concepts.

- `Product Type : a value has all of these fields`

`E.g. data Point = Point Double Double`

- `Sum Type     : a value is one of these fields`

`E.g. data Direction = North | South | East | West`

Functions

Defining and Applying Functions

```
-- takes Int, then another Int, and returns Int

add :: Int -> Int -> Int

add x y = x + y

-- called using spaces, not parentheses.

add 1 2         -- evaluates to 3

add 1 (2+3)     -- evaluates to 6
```

Backticks in Function Application

Backticks (`) treats a function as an infix operator

```
add :: Int -> Int -> Int

add x y = x + y

-- prefix (the standard way)

add 1 2

-- infix (using backticks)

1 `add` 2
```

Pattern Matching

```
-- definitions checked sequentially from top to bottom

factorial :: Int -> Int

factorial 0 = 1                       -- Base Case

factorial n = n * factorial (n - 1)   -- Recursive Case
```

Significant Whitespaces

In Haskell, whitespaces do matter. They serve as indentation similar to Python.

```
-- where introduces local bindings (local variables)

calculatePrice :: Double -> Double

calculatePrice x = itemPrice + tax

  where

    itemPrice = x * 0.8                -- 20% Discount

    tax       = itemPrice * 0.07       --  7% VAT
```

Common Syntactic Sugars in Haskell

The Dollar Sign ($) serves as a shortcut to function application.
The dot symbol (.) serves as a shortcut to function composition.

```
-- $ feeds right side as the argument to the left side

putStrLn $ show (1 + 1)      -- feeds show (1+1) to putStrLn

putStrLn (show (1 + 1))      -- same as above

-- . chains functions together (no application yet)

f . g . h $ 10               -- feeds 10 into chained f.g.h

f (g (h 10))                 -- same as above
```

Lists and Tuples

```
-- lists are homogeneous (all must be the same type)
listExample :: [Int]
listExample = [1, 2, 3, 4, 5]
-- tuples are heterogenous (multiple types allowed)
tupleExample :: (Int, Bool, Char, Double)
tupleExample = (1, True, 'a', 5.0)
```

List Comprehension

```
-- Syntax: [ expression | generator, filter(s) ]
[x * x | x <- [1..10], x 'mod' 2 == 0]
-- Evaluates to [4, 16, 36, 64, 100]
[x * x | x <- [1..10], x 'mod' 2 /= 0]
-- Evaluates to [1, 9, 25, 49, 81]
```

Infinite Lists

```
-- Syntax: [start,second..end], step is second - start
[2,4..10]  -- Evaluates to [2, 4, 6, 8, 10]
[1..]      -- Evaluates to [1, 2, 3, 4, 5, 6, 7, 8, 9, ..]
take 10 [x | x <- [1..], x 'mod' 7 == 0]
-- Evaluates to [7, 14, 21, 28, 35, 42, 49, 56, 63 ,70]
```

Application

In the previous months, I chose to explore Music Theory from a theoretical perspective with a college textbook intended for students in music school to use for study in their first and second years. I only got through a few pages. I consider the material to be extremely dense even with my past experience playing piano casually (enrolled in 1-hour weekly sessions) for around 6 years as a kid. With my past piano experience combined with my adventure into music theory, I was inspired to represent basic musical scales translation using Haskell.

The Major and Natural Minor Scales

The Major Scale consisted of intervals with the below pattern starting from the root note:
WHOLE - WHOLE - HALF - WHOLE - WHOLE - WHOLE - HALF

Some may prefer the word TONE in place of WHOLE and SEMITONE in place of HALF. Throughout my training I was more comfortable with TONE and SEMITONE, but the book uses WHOLE and HALF so I chose to represent it in accordance with the book. WHOLE is an interval that is exactly 2 steps apart, with this it is obvious that a HALF is 1 step apart. If we call a WHOLE a TONE, we would say that it is 2 SEMITONES apart.

The Natural Minor Scale is one of the Minor Scales and it also has a pattern:
WHOLE - HALF - WHOLE - WHOLE - HALF - WHOLE - WHOLE

If we represent a HALF as an interval that is 1 unit in length, we can do scales in numbers. Enabling us to do computations and transformations with scales.

The Major Scale: 2 - 2 - 1 - 2 - 2 - 2 - 1
The Natural Minor Scale: 2 - 1 - 2 - 2 - 1 - 2 - 2

Similarly, we can represent motifs using numbers (steps apart from the root note).
We can use a list to store the motif.
Twinkle Twinkle Little Star: [1, 1, 5, 5, 6, 6, 5, 4, 4, 3, 3, 2, 2, 1]

My objective is to be able to represent scales mathematically, which will allow us to transpose this motif into any scale with ease. The app should be able to do the following.

1. Given a root note (e.g. C) and a pattern of intervals (e.g. majorPattern) it should be able to construct and return a list of notes in that particular scale. For a C Major (root_note = C, pattern = majorPattern), it would return [C,D,E,F,G,A,B]
2. Given a motif (e.g. [1, 1, 5, 5, 6, 6, 5, 4, 4, 3, 3, 2, 2, 1]) and a list of notes in a scale (e.g. [C,D,E,F,G,A,B]), it should be able to transpose that motif into a scale that has the list of notes provided.

The first objective is fulfilled by the *buildscale* function and second by *transposeMotif*.
I provided the syntax highlighted image of the code from the next page of this document.

```haskell
module Music where

data Note = C | Cs | D | Ds | E | F | Fs | G | Gs | A | As | B
  deriving (Eq, Ord, Enum, Bounded, Show)

data Interval = H | W deriving (Show)

type ScalePattern = [Interval]

majorPattern :: ScalePattern
majorPattern = [W, W, H, W, W, W, H]

minorPattern :: ScalePattern
minorPattern = [W, H, W, W, H, W, W]

:
```

The 1st portion of the code

```haskell
intervalToInt :: Interval -> Int
intervalToInt H = 1
intervalToInt W = 2

intToNote :: Int -> Note
intToNote n = toEnum (n `mod` 12)

buildScale :: Note -> ScalePattern -> [Note]
buildScale root pattern =
  init $ map intToNote $ scanl (+) (fromEnum root) (map intervalToInt pattern)

type Motif = [Int]

transposeMotif :: Motif -> [Note] -> [Note]
transposeMotif motif scale =
  map (\degree -> scale !! (degree - 1)) motif

:
```

The 2nd portion of the code

```haskell
main :: IO ()
main = do
  let twinkleMotif = [1, 1, 5, 5, 6, 6, 5, 4, 4, 3, 3, 2, 2, 1]

  let cMajorScale = buildScale C majorPattern
  let eMajorScale = buildScale E majorPattern
  let aMinorScale = buildScale A minorPattern

  putStrLn "==> Scales"
  putStr "C Major Scale:   " >> print cMajorScale
  putStr "E Major Scale:   " >> print eMajorScale
  putStr "A Natural Minor: " >> print aMinorScale
  putStrLn ""

  putStrLn "==> Transposed Motif"
  putStr "Twinkle in C Major: " >> print (transposeMotif twinkleMotif cMajorScale)
  putStr "Twinkle in E Major: " >> print (transposeMotif twinkleMotif eMajorScale)
  putStr "Twinkle in A Minor: " >> print (transposeMotif twinkleMotif aMinorScale)
:
```

The 3rd portion of the code

```
~/HaskellMusic> runghc Music.hs
==> Scales
C Major Scale:   [C,D,E,F,G,A,B]
E Major Scale:   [E,Fs,Gs,A,B,Cs,Ds]
A Natural Minor: [A,B,C,D,E,F,G]

==> Transposed Motif
Twinkle in C Major: [C,C,G,G,A,A,G,F,F,E,E,D,D,C]
Twinkle in E Major: [E,E,B,B,Cs,Cs,B,A,A,Gs,Gs,Fs,Fs,E]
Twinkle in A Minor: [A,A,E,E,F,F,E,D,D,C,C,B,B,A]
~/HaskellMusic>
```

Terminal Output (runghc)

References

Peyton Jones, Simon, ed. (2003). Haskell 98 Language and Libraries: The Revised Report. Cambridge University Press. ISBN 978-0521826143.

Marlow, Simon, ed. (2010). Haskell 2010 Language Report (PDF). Haskell.org.

Davie, Antony (1992). An Introduction to Functional Programming Systems Using Haskell. Cambridge University Press. ISBN 978-0-521-25830-2.

Bird, Richard (1998). Introduction to Functional Programming using Haskell (2nd ed.). Prentice Hall Press. ISBN 978-0-13-484346-9.

Hudak, Paul (2000). The Haskell School of Expression: Learning Functional Programming through Multimedia. New York: Cambridge University Press. ISBN 978-0521643382.

Hutton, Graham (2007). Programming in Haskell. Cambridge University Press. ISBN 978-0521692694.