Programming Language Survey Paper on Haskell

Kurt O'Hearn
Nicholas Olesak

April 22, 2013

Contents

1 Introduction

This paper provides an overview of the Haskell programming language. The sections are outlined as follows: Section 1 describes the background surrounding Haskell, including its connections to functional programming and its usage today. Section 2 provides a survey of many of the data and control constructs native to Haskell. Section 3 presents a brief look at several extensions beyond the Haskell core.

1.1 Origin and History

In the early 1930s, Alonzo Church formulated a mathematical abstraction known as lambda calculus, which involves the description and evaluation of functions in mathematics. Lambda calculus became the foundation for functional programming [3]. However, as numerous functional languages began to develop, many exhibited similar expressiveness and semantics. This led to desire for a standard functional language to aid the spread of functional programming. A meeting was held in 1987 at a Functional Programming Languages and Computer Architecture conference to address this need [9]. The meeting resulted in the formation of a committee to design a standard functional language. In 1990, the committee performed a consolidation of the aspects of existing functional programming languages into an open standard, common language to be used for functional programming research: Haskell [8].

Haskell is named after Haskell Curry, a logician whose work greatly contributed to functional programming. The logo of Haskell, lambda, symbolizes that lambda calculus is the foundation from which Haskell was based. Today, Haskell is widely used in many different aspects of computing.

1.2 Functional Programming Paradigm

The core of Haskell is a pure functional language, which means that its functions possess no concept of memory locations or other hardware-related statefullness. Every function acts independently of other functions and does not change the state of anything outside its scope. This concept is referred to as having no "side-effects," and this affords Haskell a certain simplicity in design and implementation, as well as lends it to other means such as optimization toward parallel processing. In addition, functions do not have

to be declared in a particular order since they all exist independently of one another.

Beside function independence, Haskell also follows the lazy function evaluation paradigm which characterizes some other functional languages. Because of lazy evalution, functions in Haskell are not evaluated until their return value is needed by some other function. This leads to some interesting constructs that cannot be implemented in non-lazy languages, such as infinite lists.

1.3 Application Domains

Haskell is most commonly used for the development of substantial software systems. The language is designed to increase productivity from programmers in fewer lines of code. In addition, fewer errors are often produced by developers due to the lack of side-effects in functions. These features are very important in the development of large software systems, and Haskell is an ideal choice.

Other popular uses of Haskell include the development of domain specific languages, web startups, mathematics, hardware design firms, finance, and aerospace and defense. Haskell's use in commercial industry ranges from AT&T's automated processing of abuse complaints to the National Radio Astronomy Observatorys implementation of core science algorithms.

2 Data and Control Abstractions

This section presents a tutorial-style introduction to the core concepts of the Haskell language [11]. Data and control abstractions are covered in the proceeding sections. To begin with, the obligatory "Hello, World!" program is Haskell is presented below.

```
module Main where

main :: IO ()
main = putStrLn "Hello World!"
```

2.1 Data Abstractions

Haskell supports a very rich, strong, static type system for programmers. As Haskell is a compiled language — with the Glasgow Haskell Compiler (GHC) being the most common implementation — most of the type checking is performed and enforced at compile time.

2.1.1 Data Types

Haskell includes many of the data types that are standard in most languages, such as Integers, Floats, Characters, and Boolean values.

```
let x = 2
let pi = 3.14
let c = 'c'
let isHaskellAwesome = True
```

Two more foundational data types in Haskell are lists and tuples. Lists are the most used data type due to their expendability and the wide range of functions that are available to manipulate them. They can easily be created, traversed, concatenated, reversed, sorted, and more with standard, built-in functions. Lists, however, are homogeneous, ordered collections of similar data types. Tuples, on the other hand, are heterogeneous collections. Tuples are most useful when the types and exact number of values to be combined are known. Like lists, there is a wide array of built-in functions for tuples including, for instance, a function for zipping two lists into one by joining matching elements into tuples. Simple yet elegant code can be written by employing lists, tuples, and the other various other data types in Haskell.

```
let evens = [2, 4, 6, 8, 10]          -- list
let t = (3.14, 'pi')                  -- tuple
```

Haskell also allows definition Strings, which are represented by surrounding text with double quotes. Strings, however, are not their own data type. Although string literals can be initialized using the double quote syntax, the compiler converts the string into a list of characters at compile time. The use of the String notation is simply syntatic sugar for programmers. It also helps increase the readability of code. In the following example, `str1` shows how the data for a string is actually stored: as a list of characters. However, Haskell allows the definition of a string using double quotes, as seen in `str2`, which is much more understandable.

```
let str1 = ['H','a','s','k','e','l','l','!']
let str2 = "Haskell!"   -- shorthand for str1
```

In addition to these basic data types, Haskell also offers type variables. Type variables are variables whose type has not been explicitly declared; rather, they can be any type. Type variables are most often employed to define the parameter and return value types of functions. By defining a function without declaring data types, the variables in the function can be used as any type thereby enabling polymorphic behaviors. In the following example, the function can handle a list of any type because its uses a type variable.

```
lastTwo :: [a] -> [a]
lastTwo a = [a !! ((length a) - 2), a !! ((length a) - 1)]
```

2.1.2 Type Checking

Type checking in Haskell is performed statically at compile time. Haskell uses type-declarations for functions, in which the types of parameters and return values are explicitly defined. Any use of a function in which parameters do not meet the defined function type is flagged by the compiler, and compilation will be aborted. In addition, Haskell supports type inferencing by context. The type system in Haskell deduces obvious types and makes them concrete. In the following example, the `read` function is used to convert `myStr` to another data type. Although the type of `myNum` is unknown at compile time, the compiler infers that the Int type is desired because `myStr` is added to an Int type.

```
let myStr = "2"
let myNum = (read myStr) + 2
```

If the context does not suggest the type, as in the example below, then an exception will result because the compiler cannot infer what the type is going to be and is unable to handle it.

```
let myStr = "2"
let myNum = read myStr
```

To avoid ambiguity in type conversion, the following syntax can be employed to indicated to the compiler what type is desired from the `read` function.

```
let myStr = "2"
let myNum = (read myStr) :: Float
```

Haskell uses typeclasses to categorize the type system. Typeclasses define the behavior of a general type. A type must support the definition of its typeclass, but may also be more specific. For example, all of the operators in Haskell are categorized under the Eq typeclass. Other examples of typeclasses in Haskell include Ord, Show, Read, Enum, Num, Integral, and Floating. Typeclasses can be used to allow multiple types to be passed to a function. In the following example, the function declares its typeclass as Integral. Therefore, valid functions invocations consist of Int or Integer type parameters.

```
incrementMe :: (Integral a) => a -> a
incrementMe a = a + 1
```

2.2 Control Abstractions

Haskell supports several constructs for controlling program execution. These constructs can be further divided into: expressions, functions, and operators, selection and iteration mechanisms, modules, and execption handling. In order to further understand the functionality of these constructs, scoping and parameter passing techniques are discussed.

2.2.1 Expressions and Functions

Function definitions in Haskell are very simple and straightforward. A function consists of the function name, parameters, and an expression. The return value of the function is the evaluation of the expression, which uses the parameters. The naming standard for functions in Haskell is to use camel case. All functions in Haskell must take at least one parameter and must return only one value. They can, however, be passed multiple parameters. Functions are usually called with prefix notation; that is, the name of the function is followed by its arguments. It is possible, however, to call functions using infix notation, which is sometimes appropriate if the function takes two arguments. Additionally, all operators are called using infix notation. The following example demonstrates how a function can be called using infix notation.

```haskell
let evens = [2, 4, 6, 8]
print (elem 2 evens)    -- prefix notation
print (2 `elem` evens) -- infix  notation
```

The data type for each parameter of a function can be explicitly declared by using the `::` symbol, which is read as "has a type of." The types of the parameters can be listed in order, and the last type in the list is the return type. The following function is called `greet`. It takes in a string and returns a string. The string is passed in as the variable `name`. Everything that follows the equal sign is the expression, and the evaluation of this expression is the return value.

```haskell
greet :: String -> String
greet name = "Hello, " ++ name ++ "!"
```

Bindings in Haskell can be created in two different ways. The first is using `where` bindings. `where` bindings allow the definition of a function inside another function. The scope of the function declared by a `where` is locally bound by the function that created it. Functions outside of the declaring function cannot call the function declared inside the `where`. Additionally, `where` bindings can be nested; that is, a function declared inside a `where` can declare its own helper functions inside a `where`. `where` bindings can also span across guards, which will be covered shortly.

In the following example, the function `rectPrismVol` declares two functions: `faceArea` and `baseArea`. Only the `rectPrismVol` function is allowed to call these functions because they are locally bound.

```haskell
rectPrismVol :: (Num a) -> a -> a -> a -> a
rectPrismVol length width height = 4 * (faceArea length width)
    + 2 * (baseArea length height)
where   faceArea l w = l * w
        baseArea = l * h
```

The second type of binding available in Haskell is done using `let` bindings. `let` bindings allow the declaration of variables that are expressions themselves and are locally bound. `let` bindings are often used for pattern matching. The syntax of `let` bindings involves declaring the bindings inside a `let` declaration, followed by defining an expression inside an `in` declaration. The expression in the `let` and `in` parts must be aligned in the same column.

```haskell
myExp :: (Num a) => a -> a
myExp x =
    let e = 2.7182
    in  e ** x
```

The difference between `let` and `where` bindings is that `let` bindings are expressions themselves, whereas `where` bindings are just syntactic constructs attached to some expresion. Additionally, unlike `where` bindings, `let` bindings cannot span across guards.

2.2.2 Operators and Precedence

Operators in Haskell are typically represented with symbols. Some of the basic operators in Haskell include addition, subtraction, multiplication, divison, exponentiation, concatenation, and equality and logic operators. The precedence of operators conform to the standard orders used in mathematics: exponentiation is highest, followed by muliplication and division, with addition and subtraction assuming the lowest precedence.

All operators in Haskell are binary, and they are usually called using infix notation; that is, the operator is placed between its two arguments. Although it is possible to use operators in prefix notation, it is customary to use infix notation to increase readability. This contrasts from most functional programming languages.

```haskell
print (2 + 5)     -- infix notation is the standard
print ((+) 2 5)   -- prefix notation is still possible
```

Operators in Haskell can also be overloaded. The number and type of arguments as well as the expression can be modified.

```haskell
(*) :: (Num a) => a -> a -> a
(*) a b = a + a + a + a
print (2 * 3)    -- yields 8 rather than 6
```

2.2.3 Selection Constructs

Selection constructs in Haskell consist of if-else statements, guards, pattern matching, and case expressions.

If-else statements in Haskell conform to the traditional syntax of the if-else selection construct. The only notable difference is that Haskell requires that every `if` conditional statement is followed by a `then` expression and an `else` expression. The compiler will prevent compilation for any `if` conditional that does not also have `then` and `else` statements because without them it would be possible for an input to cause a function not to have a return value, which Haskell does not allow.

```
guessGame :: Int -> String
guessGame x = if x == 42
    then "You have a deep understanding of life."
    else "You need to think more about life."
```

If-else statements are not used in Haskell as often as they are in imperative languages because Haskell offers another feature known as guards, which are fundamentally equivalent to if-else statements except they are much more easily understood, work with multiple conditions, and blend well with pattern matching. Guards are created using pipes in the expression of a function and consist of a conditional, which must evaluate to true or false, and an expression. An equals sign separates the conditional from the expression. Arguments are checked against the guards in sequential order. If the condition is true, then the expression for that guard is evaluated and returned. If the condition is false, then the next guard condition is evaluated, and this pattern continues until the `otherwise` guard is reached, which catches all input that did not match any guard previous conditions. The following function has three guards, each following a pipe. A notable aspect of this function is that if the input is value is greater than 100,000 it will cause the first guard condition to be true and will evaluate the first guard expression even though the input would also be true for the second guard. This is because the guards are checked in the order which they are defined, so the second guard is never checked in this case because the first guard is true and the return value is determined from it. The `otherwise` guard is placed at the bottom to catch any value that does not cause the first two guards to be true.

```haskell
let balance = 200
assessWorth :: Int -> String
assessWorth =
    | balance > 100000 = "You're rich!"
    | balance > 100 = "You're rich by student standards!"
    | otherwise = "Need a loan, friend?"
```

Another selection construct in Haskell is pattern matching. Pattern matching is simply a way to have a function perform different actions based on its input. Pattern matching is done by defining a function multiple times in succession using patterns in place of parameters. When the function is called, its arguments are checked against all patterns, starting with the first and proceeding accordingly. If no match is made, an exception is raised. A default match can be specified by using variables as function parameters rather than a pattern.

In the following example, an input of "Haskell" will produce the first expression, "Clojure" will skip the first and produce the second, and any other input will produce the third. Note that the third and final expression does not have a pattern to match, but instead uses a variable to catch all other input.

```haskell
favoriteLang :: String -> String
favoriteLang "Haskell" = "You have refined taste, my friend."
favoriteLang "Clojure" =
    "The wise do not build a house on unsturdy ground."
favoriteLang lang = lang ++
    " is a fine choice, but you should give Haskell a spin."
```

The final selection construct in Haskell is case expressions. Case expressions evaluate expressions based on the values of a parameter. They are similar to case constructs in imperative languages, except they can also pattern match the input. Note that this is exactly how pattern matching parameters in function definitions works. In fact, pattern matching parameters is simply an easier way to write case expressions, and the two are interchangeable.

The following function demonstrates how to evaluate a specific expression that depends on which pattern an argument matches by using the `case` construct. Note that if this function were written using the pattern matching technique it would work exactly the same way.

```haskell
favoriteMathSymbol :: String -> String
favoriteMathSymbol x = case x of
    "Pi" -> "Geometry is your mind's eye."
    "e" -> "May the calculus forever be with you."
    "Epsilon" -> "I think I am within the limit, "
        ++ "therefore, I am within the limit."
    x = "You have much to learn of the maths."
```

2.2.4 Iterative Constructs

In Haskell, multiple constructs are supplied for the programmer to perform repetitive tasks including list comprehensions, map- and fold-style functions, and function recursion.

List comprehensions in Haskell are a means to transform the elements in a list in some manner. To facilitate the transformation, multiple parts are in the construct are used: the input elements, the predicates, and the output function. The input elements are typically derived from another list. Then, the predicates are applied to each of these elements in other to restrict the input set. Finally, the restricted set then has the output function applied to every element in this set and the resulting list is returned. An example below is given where `xs` is the input list, `x `mod` 3 == 0` is a predicate used for filtering, and `x` is the simple output function.

```haskell
filterModThree :: [Int] -> [Int]
filterModThree xs = [x | x <- xs, x `mod` 3 == 0]
```

Maps and folds are in many ways similar to list comprehensions. In a map, a function is specified to apply to every element of a list to produce a new list. A fold in Haskell aggregates all of the elements in a list into a single data type using some user-specified function. An example using both map and reduce is given below. In this example, every element in a list is first squared (mapped) and the resulting list is then summed up (folded).

```haskell
sumSquares :: (Num a) => [a] -> a
sumSquares xs = foldr1 (+) (map (**2) xs)
```

All of the previous iterative constructs covered to this point share one thing in common: they all could be implemented by recursion. Recursion is the fundamental construct for iterating in Haskell. Recursion is implemented

in functions in Haskell through pattern matching. The boundary case in a recursive definition terminates the iteration; the recursive process is typically defined using pattern matching, where the boundary case is specified with a pattern while the recursive case is set up as the default case. The function below demonstrates this scenario by pattern matching on an integer value of the input parameter `x`.

```
countdown :: Int -> String
countdown 0 = "Blast-off!"
countdown x = (show x) ++ "...\n" ++ countdown (x-1)
```

It is worth noting that in Haskell, tail and guarded recursion are both utilized for improved performance by minimizing the function frame allocations on the stack. However, because of the lazy-evaluation philosophy of Haskell, recursions may be foregone until required, thereby negating the need for recursion optimization in some instances [6].

2.2.5 Functions and Parameter Passing

Up to this point, several functions have been defined and used in the examples. As Haskell is a functional programming language, functions are the primary modular construct employed to accomplish tasks. Worth noting here, the parameter passing techniques supported in functions are explicit by default. That is, bindings can be scoped to functions by explicitly invoking a function with the arguments that want to be scoped to the function in general. However, there are a few notable exceptions that the community has developed to support implicit parameter passing to functions of some degree:

- Constant function definitions can be employed at the top level of a module [4]
- Monads can be utilized to effectively hide global definitions [4]

2.2.6 Scoping

In Haskell, static scoping is employed in resolving the origins of bindings. As such, active bindings are first those localized to the function followed by global scope. Thus, bindings within callee functions are not active in the

caller function unless explicitly passed as parameters. This tenent of staticly binding within functions works toward the end of keeping functions from producing any side-effects [1].

2.2.7 Modules

Modules in Haskell are the basic unit of organizing functions and other constructs. The standard Haskell library is composed of thousands of well-developed and maintained modules for a variety of purposes. In the example below, three functions are defined and made available to any other program that imports this module.

```
module myMod
(
addTwo,
addThree,
addFour
) where

addTwo :: (Num a) => a -> a
addTwo x = x + 2

addThree :: (Num a) => a -> a
addThree x = x + 3

addFour :: (Num a) => a -> a
addFour x = x + 4
```

2.2.8 Exception Handling

Haskell, like many other languages, provides ways to utilize error handling code in the event of abnormal program execution. However, Haskell is different from other languages in the sense that the preferred way to handle these circumstances depends upon the portion of the code which requires action. To explain this point, functions in Haskell must be classifed as either pure or impure. Recall that in previous sections, the Haskell language core was claimed to be a pure functional programming in the sense that functions do not produce any side-effects beyond their output values. However, certain functions in Haskell must break this trend in some respect; such functions

include those responsible for I/O, concurrency, and interfaces to other languages [10]. As a side note, this split from the functional paradigm for these types of functions has been mended by the introduction of monads in Haskell, a topic which unfortunately will not be covered here.

For impure functions, `exceptions` are the favored way to perform error handling. The following example provides an error handling function `handler` associated with the `tryThis` function for intervening with I/O file-related issues.

```
import System.Environment
import System.IO
import System.IO.Error

main = tryThis `catch` handler

tryThis :: IO ()
tryThis = do (fileName:_) <- getArgs
    fileContents <- readFile fileName
    putStrLn $ "File contains "
        ++ show (length (lines fileContents)) ++ " lines"

handler :: IOError -> IO ()
handler e
    -- handle exception
    | isDoesNotExistError e = putStrLn "The file does not exist."
    -- re-raise exception
    | otherwise = ioError e
```

For pure functions, the Haskell typeclass system was broadened to allow the introduction of the `Maybe` algebraic type. This type extension provides a function a way to indicate that it may or may not return a value of the specified type in its declaration (i.e., it may fail to return any value). If a function implementing this typeclass fails this way, the return value will be `Nothing`. If it does return a value, say `x`, that value will be prefaced by the keyword `Just`. A pure function illustrating this concept is given below.

```haskell
doubleHead :: [Int] -> Maybe Int
doubleHead [] = Nothing
doubleHead xs = Just ((head xs) * 2)

main = do
    print (doubleHead [2,4,6])  -- output: Just 4
    print (doubleHead [])       -- output: Nothing
```

3 Advanced Topics

Beyond the basic data and control structures discussed to this point, Haskell contains a wide array of constructs, extension modules, and and other features. This section considers the following advanced topics: inheritance, concurrency, and runtime type checking.

3.1 Inheritance

Inheritance in Haskell can be achieved through class extension, which involves defining a class as a subclass of some other class. The subclass then inherits all of the operations in the superclass. A benefit to class extension is shorter contexts: a type expression for a function that uses operations from both the subclass and the superclass only has to use the context of the subclass. Additionally, methods for subclass operations can assume the existence of methods for superclass operations.

The following class definition of the `Ord` typeclass inherits all of the operations in the `Eq` typeclass but also obtains comparison operators as well as the `maximum` and `minimum` functions.

```haskell
class  (Eq a) => Ord a  where
    (<), (<=), (>=), (>)  :: a -> a -> Bool
    max, min              :: a -> a -> a
```

3.2 Concurrency Support

In Haskell, there is a mature library of modules for built-in and extension-level concurrency and parallelism [2]. Platforms that support concurrent and

parallel execution include shared- and distributed-memory systems, accelerator-style devices (e.g., general-purpose programming), and symmetric and unsymmetric architectures [12]. Multi-threading and multiprocessing capabilities are available in the standard Haskell library. In the following example, multiple user-level threads are created for printing to the main terminal device, one of which prints the word "Hello" while the other prints "world."

on graphics processing units)

```
import Control.Concurrent
import System.IO

main :: IO ()
main = do { forkIO (hPutStr stdout "Hello")
          ; hPutStr stdout " world\n" }
```

3.3 Introspection and Reflection

As mentioned previously, Haskell posesses a strong static type system implented at compile time. As such, the Haskell '98 language definition does not provide support for runtime type examination or manipulation (i.e., introspection and reflection, respectively). However, effors have been made toward adding support similar to these features:

- Scrap your boilerplate: generic programming [5]
- Template Haskell: an extension module for compile-time metaprogramming [7]

A Project Implementation in Haskell

```haskell
{-
 - Author: Kurt O'Hearn
 - Author: Nick Olesak
 - Purpose: CIS 343-02 Project
 - Date: 2013-04-08
 - Descrption: This program servers as a simple calculator using
 - postfix expressions. Internally, the program converts postfix
 - expressions to infix notation and subsequently evaluates the infix.
-}

module InfixPostfix
( infixToPostfix,
evaluatePostfix
) where

import Data.List.Split

-- converts an infix expression to a postfix expression
infixToPostfix :: String -> String
infixToPostfix expr =
    let concatHigherPrec :: [String] -> String -> String
        concatHigherPrec [] _ = ""
        concatHigherPrec (x:xs) y
            | (stackPrec x) >= (inputPrec y) = x ++ " "
                ++ (concatHigherPrec xs y)
            | otherwise = ""

        keepLowerPrec :: [String] -> String -> [String]
        keepLowerPrec [] _ = []
        keepLowerPrec all@(x:xs) y
            | (stackPrec x) >= (inputPrec y) = (keepLowerPrec xs y)
            | otherwise = all

        concatBeforeLeftParen :: [String] -> String
        concatBeforeLeftParen ("(":xs) = ""
        concatBeforeLeftParen (x:xs) = x ++ " "
```

```
            ++ (concatBeforeLeftParen xs)

        removeToLeftParen :: [String] -> [String]
        removeToLeftParen ("(":xs) = xs
        removeToLeftParen (x:xs) = removeToLeftParen xs

        parseExpr [] stack =
            (foldl (++) "" (zipWith (++)
                stack (replicate (length stack - 1) " ")))
            ++ (last stack)
        parseExpr (x:xs) stack
            | isOperand x = x ++ " " ++ (parseExpr xs stack)
            | isLeftParen x = (parseExpr xs (x:stack))
            | isOperator x = (concatHigherPrec stack x)
                ++ (parseExpr xs (x:(keepLowerPrec stack x)))
            | (isRightParen x) && ("(" `elem` stack) =
                (concatBeforeLeftParen stack)
                ++ (parseExpr xs (removeToLeftParen stack))
            | otherwise = error "Not a valid identifier: " ++ x

    in parseExpr (splitOn [' '] expr) []


-- evaluates a postfix expression
evaluatePostfix :: String -> String
evaluatePostfix expr = (parsePostfixExp (splitOn [' '] expr) [])
    where parsePostfixExp [] stack = head stack
          parsePostfixExp (x:xs) (s1:s2:stack)
              | isOperand x = parsePostfixExp xs (x:s1:s2:stack)
              | isOperator x =
                    parsePostfixExp xs ((applyOperator s2 s1 x):stack)
              | otherwise = error ("Not a valid identifier")
          parsePostfixExp (x:xs) stack = parsePostfixExp xs (x:stack)


-- returns true if param is an operator
isOperator :: String -> Bool
isOperator operator
```

```
    | operator `elem` ["+","-","*","/","%","^"] = True
    | otherwise = False


-- returns true if param is an operand
isOperand :: String -> Bool
isOperand operand = case reads operand :: [(Integer, String)] of
    [(_, "")] -> True
    _         -> False


-- returns true if param is a left parenthesis
isLeftParen :: String -> Bool
isLeftParen "(" = True
isLeftParen operator = False


-- returns true if param is a right parenthesis
isRightParen :: String -> Bool
isRightParen ")" = True
isRightParen operator = False


-- get the input precedence of an operator
inputPrec :: (Integral a) => String -> a
inputPrec operator
    | operator `elem` ["+", "-"] = 1
    | operator `elem` ["*", "/", "%"] = 2
    | operator == "^" = 4
    | operator == "(" = 5
    | otherwise = error ("Not a valid operator: " ++ operator)


-- get the stack precedence of an operator
stackPrec :: (Integral a) => String -> a
stackPrec operator
    | operator `elem` ["+", "-"] = 1
    | operator `elem` ["*", "/", "%"] = 2
```

```
    | operator == "^" = 3
    | operator == "(" = -1
    | otherwise = error ("Not a valid operator: " ++ operator)


-- apply an operator on two values. Returns value as string
applyOperator :: String -> String -> String -> String
applyOperator num1 num2 operator = case operator of
    "+" -> show ((read num1 :: Int) + (read num2 :: Int))
    "-" -> show ((read num1 :: Int) - (read num2 :: Int))
    "*" -> show ((read num1 :: Int) * (read num2 :: Int))
    "/" -> show ((read num1 :: Int) `div` (read num2 :: Int))
    "%" -> show ((read num1 :: Int) `mod` (read num2 :: Int))
    "^" -> show ((read num1 :: Int) ^ (read num2 :: Int))
    otherwise -> error ("Not a valid operator: " ++ operator)
```

```
import Test.HUnit
import InfixPostfix

-- main method
main = do
    let test1 = TestCase (assertEqual
                "infixToPostfix Test 0" "3 4 2 5 ^ - * 6 +"
                (infixToPostfix "3 * ( 4 - 2 ^ 5 ) + 6"))
    let test2 = TestCase (assertEqual
                "infixToPostfix Test 1" "3 2 1 2 + ^ ^"
                (infixToPostfix "3 ^ 2 ^ ( 1 + 2 )"))
    let test3 = TestCase (assertEqual
                "infixToPostfix Test 2" "10 2 2 2 ^ ^ * 10 50 * +"
                (infixToPostfix "10 * ( 2 ^ 2 ^ 2 ) + 10 * 50"))
    let test4 = TestCase (assertEqual
                "infixToPostfix Test 3" "100 50 2 3 ^ - / 50 10 / - 5 +"
                (infixToPostfix "100 / ( 50 - 2 ^ 3 ) - 50 / 10 + 5"))
    let test5 = TestCase (assertEqual
                "infixToPostfix Test 4" "10 54 10 % 25 10 - 2 2 ^ + * 3 / +"
                (infixToPostfix "10 + 54 % 10 * ( 25 - 10 + 2 ^ 2 ) / 3"))
    let test6 = TestCase (assertEqual
                "infixToPostfix Test 5" "10 5 2 % + 5 3 3 ^ * + 25 5 / -"
                (infixToPostfix "( 10 + 5 % 2 ) + ( 5 * 3 ^ 3 ) - ( 25 / 5 )"))

    let test7 = TestCase(assertEqual
                "evaluatePostfix Test 0" "-78"
                (evaluatePostfix "3 4 2 5 ^ - * 6 +"))
    let test8 = TestCase(assertEqual
                "evaluatePostfix Test 1" "6561"
                (evaluatePostfix "3 2 1 2 + ^ ^"))
    let test9 = TestCase(assertEqual
                "evaluatePostfix Test 2" "660"
                (evaluatePostfix "10 2 2 2 ^ ^ * 10 50 * +"))
    let test10 = TestCase(assertEqual
                "evaluatePostfix Test 3" "2"
                (evaluatePostfix "100 50 2 3 ^ - / 50 10 / - 5 +"))
    let test11 = TestCase(assertEqual
                "evaluatePostfix Test 4" "35"
```

```
            (evaluatePostfix "10 54 10 % 25 10 - 2 2 ^ + * 3 / +"))
let test12 = TestCase(assertEqual
            "evaluatePostfix Test 5" "141"
            (evaluatePostfix "10 5 2 % + 5 3 3 ^ * + 25 5 / -"))

let tests = TestList [test1, test2, test3, test4,
                test5, test6, test7, test8,
                test9, test10, test11, test12]
runTestTT tests
```

References

[1] Cse 341 – static and dynamic scoping. http://www.cs.washington.edu/education/courses/cse341/03wi/imperative/scoping.html. Accessed: 2013-04-05.

[2] Haskell for multicores. http://www.haskell.org/haskellwiki/Haskell_for_multicores. Accessed: 2013-04-12.

[3] The haskell programming language. http://www.haskell.org/haskellwiki/Haskell. Accessed: 2013-04-02.

[4] Other type system extensions. http://www.haskell.org/ghc/docs/latest/html/users_guide/other-type-extensions.html#implicit-parameters. Accessed: 2013-04-13.

[5] Scrap your boilerplate. http://www.haskell.org/haskellwiki/Scrap_your_boilerplate. Accessed: 2013-04-15.

[6] Tail recursion. http://www.haskell.org/haskellwiki/Tail_recursion. Accessed: 2013-04-09.

[7] Template haskell. http://www.haskell.org/haskellwiki/Template_Haskell. Accessed: 2013-04-15.

[8] P. Hudak. Conception, evolution, and application of functional programming languages. *ACM Comput. Surv.*, 21(3):359–411, Sept. 1989.

[9] P. Hudak, J. Hughes, S. Peyton Jones, and P. Wadler. A history of haskell: being lazy with class. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, HOPL III, pages 12–1–12–55, New York, NY, USA, 2007. ACM.

[10] S. P. Jones. Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in haskell. In *Engineering theories of software construction*, pages 47–96. Press, 2002.

[11] M. Lipovača. Learn you a haskell for great good! http://learnyouahaskell.com/, 2011. Accessed: 2013-03-22.

[12] R. Newton. Parfunk. http://parfunk.blogspot.co.at/2012/05/how-to-write-hybrid-cpugpu-programs.html. Accessed: 2013-04-12.