Seven Languages in Seven Weeks

Correl Roush

August 12, 2015

Created 1990

Author A committee of researchers and application programmers, including John Hughes, Simon Peyton Jones, and Philip Wadler.

- https://www.haskell.org/
- http://learnyouahaskell.com/

Haskell is a functional programming language. Its first distinguishing characteristic is that it is a pure functional language. A function with the same arguments will always produce the same result.

- Expressions
- Types
- Functions
- Tuples and Lists
- Function Composition
- List Comprehensions

Numbers

```
4           -- 4
4 + 1       -- 5
4 + 1.0     -- 5.0
4 + 2.0 * 5 -- 14.0
```

Strings

```
"hello" ++ " world" -- "hello world"
'a'                 -- 'a'
['a', 'b']          -- "ab"
```

Booleans

```
(4 + 5) == 9  -- True
(5 + 5) /= 10 -- False
if (5 == 5) then "true" else "false" -- "true"
```

if

```
if 1 then "true" else "false"
```

```
<interactive>:2:4:
    No instance for (Num Bool) arising from the literal ‘1’
    In the expression: 1
    In the expression: if 1 then "true" else "false"
    In an equation for ‘it’: it = if 1 then "true" else "false"
```

+

```
"one" + 1
```

```
<interactive>:4:7:
    No instance for (Num [Char]) arising from a use of ‘+’
    In the expression: "one" + 1
    In an equation for ‘it’: it = "one" + 1
```

Simple

```
let double x = x + x
double 2
```

```
4
```

```
:t double
```

```
double :: Num a => a -> a
```

Recursive

```
factorial :: Integer -> Integer
factorial x
    | x > 1 = x * factorial (x - 1)
    | otherwise = 1
```

Code

```
module Main where

fibTuple :: (Integer, Integer, Integer) -> (Integer, Integer, Integer)
fibTuple (x, y, 0) = (x, y, 0)
fibTuple (x, y, index) = fibTuple (y, x + y, index - 1)

fibResult :: (Integer, Integer, Integer) -> Integer
fibResult (x, y, z) = x

fib :: Integer -> Integer
fib x = fibResult (fibTuple (0, 1, x))
```

Results

```
:l fib_tuple
fib 100

354224848179261915075
```

```haskell
module Main where

fibNextPair :: (Integer, Integer) -> (Integer, Integer)
fibNextPair (x, y) = (y, x + y)

fibNthPair :: Integer -> (Integer, Integer)
fibNthPair 1 = (1, 1)
fibNthPair n = fibNextPair (fibNthPair (n - 1))

fib :: Integer -> Integer
fib = fst . fibNthPair
```

Pattern Matching

```
let (h:t) = [1, 2, 3 4]
-- h = 1
-- t = [2,3,4]
```

Recursive Traversal

```
size [] = 0
size (h:t) = 1 + size t

prod [] = 1
prod (h:t) = h * prod t
```

Zip

```
zip ["kirk", "spock"] ["enterprise", "reliant"]
-- [("kirk","enterprise"),("spock","reliant")]
```

Recursion

```
allEven :: [Integer] -> [Integer]
allEven [] = []
allEven (h:t) = if even h then h:allEven t else allEven t
```

Ranges and Composition

```
[1..4]        -- [1,2,3,4]
[10..4]       -- []
[10, 8 .. 4]  -- [10,8,6,4]
```

```
take 5 [1..]     -- [1,2,3,4,5]
take 5 [0, 2..]  -- [0,2,4,6,8]
```

List Comprehensions

```
[x * 2 | x <- [1, 2, 3]] -- [2,4,6]
[(4 - x, y) | (x, y) <- [(1, 2), (2, 3), (3, 1)]] -- [(3,2),(2,3),(1,1)]
```

```
let crew = ["Kirk", "Spock", "McCoy"]
[(a, b) | a <- crew, b <- crew, a < b]
-- [("Kirk","Spock"),("Kirk","McCoy"),("McCoy","Spock")]
```

The original goals were not modest: we wanted the language to be a foundation for research, suitable for teaching, and up to industrial uses.

Haskell's great strength is also that predictability and simplicity of logic. Many universities teach Haskell in the context of reasoning about programs. Haskell makes creating proofs for correctness far easier than imperative counterparts.

- Higher Order Functions
- Partial Application and Currying
- Lazy Evaluation

Anonymous Functions

```
(\x -> x ++ " captain.") "Logical, "
-- "Logical, captain."
```

map and where

```
squareAll list = map square list
  where square x = x * x

squareAll [1, 2, 3] -- [1,4,9]
map (+ 1) [1, 2, 3] -- [2,3,4]
```

filter, foldl, foldr

```
filter odd [1, 2, 3, 4, 5] -- [1,3,5]
foldl (\x carryOver -> carryOver + x) 0 [1 .. 10] -- 55
foldl (+) 0 [1 .. 3] -- 6
```

```
let prod x y = x * y
:t prod

prod :: Num a => a -> a -> a

let double = prod 2
let triple = prod 3

double 3 -- 6
triple 4 -- 12
```

So, the mystery is solved. When Haskell computes `prod 2 4`, it is really computing `(prod 2) 4`, like this:

- First, apply `prod 2`. That returns the function `(\y -> 2 * y)`.
- Next, apply `(\y -> 2 * y) 4`, or `2 * 4`, giving you 8.

```
let lazyFib x y = x:(lazyFib y (x + y))
let fib = lazyFib 1 1
let fibNth x = head (drop (x - 1) (take (x) fib))

take 5 (fib) -- [1,1,2,3,5]
take 5 (drop 20 (lazyFib 0 1)) -- [6765,10946,17711,28657,46368]

take 5 (map ((* 2) . (* 5)) fib) -- [10,10,20,30,50]
```

Composition

In Haskell, `f . g x` is shorthand for `f(g x)`.

Apart from purity, probably the most unusual and interesting feature of Haskell is its type system. Static types are by far the most widely used program verification technique available today: millions of programmers write types (which are just partial specifications) every day, and compilers check them every time they compile the program. Types are the UML of functional programming: a design language that forms an intimate and permanent part of the program.

- Classes and Types
- Monads

```
'c'             :: Char
"abc"           :: [Char]
['a', 'b', 'c'] :: [Char]
True            :: Bool
False           :: Bool
```

```
data Suit = Spades | Hearts
          deriving Show
data Rank = Ten | Jack | Queen | King | Ace
          deriving Show

type Card = (Rank, Suit)
type Hand = [Card]

value :: Rank -> Integer
value Ten   = 1
value Jack  = 2
value Queen = 3
value King  = 4
value Ace   = 5

cardValue :: Card -> Integer
cardValue (rank, suit) = value rank
```

Generic Functions

```
backwards [] = []
backwards (h:t) = backwards t ++ [h]
```

Could be typed as

```
backwards :: Hand -> Hand
```

or

```
backwards :: [a] -> [a]
```

Polymorphic Data Types

```
data Triplet a = Trio a a a deriving (Show)
```

Could be used as:

```
Trio 'a' 'b' 'c' :: Triplet Char
```

Defining a tree data type

```
data Tree a = Children [Tree a]
            | Leaf a
            deriving (Show)
```

Constructing and deconstructing a tree of integers

```
let tree = Children [Leaf 1, Children [Leaf 2, Leaf 3]] :: Tree Integer
let (Children ch) = tree
-- ch = [Leaf 1, Children [Leaf 2, Leaf 3]]
let (fst:tail) = ch
-- fst = Leaf 1
```

Calculating the depth of a tree

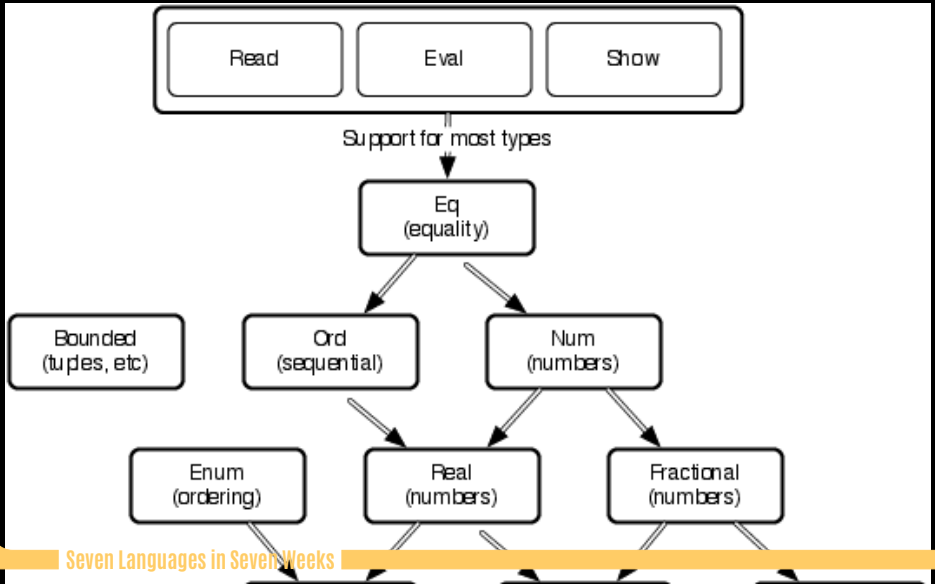
```
depth (Leaf _) = 1
depth (Children c) = 1 + maximum (map depth c)
```

- It's not an object-oriented class, because there's no data involved.
- A class defines which operations can work on which inputs.
- A class provides some function signatures. A type is an instance of a class if it supports all those functions.

```
class Eq a where
  (==), (/=) :: a -> a -> Bool

  -- Minimal complete definition:
  -- (==) or (/=)

  x /= y = not (x == y)
  x == y = not (x /= y)
```


Read
Eval
Show
Support for most types
Eq
(equality)
Bounded
(tuples, etc)
Ord
(sequential)
Num
(numbers)
Enum
(ordering)
Real
(numbers)
Fractional
(numbers)

```
def treasure_map(v)
  v = stagger(v)
  v = stagger(v)
  v = crawl(v)
  return(v)
end
```

We have several functions that we call within `treasure_map` that sequentially transform our state, the `distance` traveled. The problem is that we have mutable state.

```
module Main where

    stagger :: (Num t) => t -> t
    stagger d = d + 2
    crawl d = d + 1

    treasureMap d =
        crawl (
        stagger (
        stagger d))

    letTreasureMap (v, d) = let d1 = stagger d
                                d2 = stagger d1
                                d3 = crawl d2
                            in d3
```

The inputs and outputs are the same, so it should be easier to compose these kinds of functions. We would like to translate `stagger(crawl(x))` into `stagger(x)` · `crawl(x)`, where · is function composition. That's a monad.

At its basic level, a monad has three basic things:

- A type constructor that's based on some type of container. The container could be a simple variable, a list, or anything that can hold a value. We will use the container to hold a function. The container you choose will vary based on what you want your monad to do.
- A function called `return` that wraps up a function and puts it in the container. The name will make sense later, when we move into do notation. Just remember that `return` wraps up a function into a monad.
- A bind function called >>= that unwraps a function. We'll use bind to chain functions together.

All monads will need to satisfy three rules. I'll mention them briefly here. For some monad `m`, some function `f`, and some value `x`:

- You should be able to use a type constructor to create a monad that will work with some type that can hold a value.
- You should be able to unwrap and wrap values without loss of information. (`monad >> return = monad=`)
- Nesting bind functions should be the same as calling them sequentially. (`(m >> f) »= g = m »= (-> f x »=`

```
module Main where
    data Position t = Position t deriving (Show)

    stagger (Position d) = Position (d + 2)
    crawl (Position d) = Position (d + 1)

    rtn x = x
    x >>== f = f x

    treasureMap pos = pos >>==
                      stagger >>==
                      stagger >>==
                      crawl >>==
                      rtn
```

```
module Main where
    tryIo = do  putStr "Enter your name: " ;
                line <- getLine ;
                let { backwards = reverse line } ;
                return ("Hello. Your name backwards is " ++ backwards)
```

```
instance Monad [] where
    m >>= f = concatMap f m
    return x = [x]
```

```
let cartesian (xs,ys) = do x <- xs; y <- ys; return (x,y)
cartesian ([1..2], [3..4])
-- [(1,3),(1,4),(2,3),(2,4)]
```

Example (Password Cracker)

```
module Main where
    crack = do x <- ['a'..'c'] ; y <- ['a'..'c'] ; z <- ['a'..'c'] ;
               let { password = [x, y, z] } ;
               if attempt password
                    then return (password, True)
                    else return (password, False)

    attempt pw = if pw == "cab" then True else False
```

In this section, we'll look at the `Maybe` monad. We'll use this one to handle a common programming problem: some functions might fail.

```
data Maybe a = Nothing | Just a


instance Monad Maybe where
    return          = Just
    Nothing  >>= f = Nothing
    (Just x) >>= f = f x
```

Without a monad

```
case (html doc) of
  Nothing -> Nothing
  Just x -> case body x of
    Nothing -> Nothing
    Just y -> paragraph 2 y
```

With the Maybe monad

```
Just someWebpage >>= html >>= body >>= paragraph >>= return
```

- Type System
- Expressiveness
- Purity of Programming Model
- Lazy Semantics
- Academic Support

- Inflexibility of Programming Model
- Community
- Learning Curve

Of the functional languages in the book, Haskell was the most difficult to learn. The emphasis on monads and the type system made the learning curve steep. Once I mastered some of the key concepts, things got easier, and it became the most rewarding language I learned. Based on the type system and the elegance of the application of monads, one day we'll look back at this language as one of the most important in this book.

Haskell plays another role, too. The purity of the approach and the academic focus will both improve our understanding of programming. The best of the next generation of functional programmers in many places will cut their teeth on Haskell.