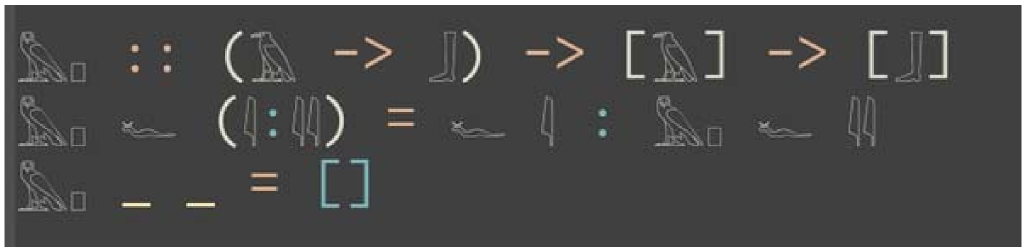
Haskell: From Basic to Advanced

Part 1 – Basic Language



By combining Haskell + Unicode, you can write perfectly functional programs in Hieroglyphics

Haskell buzzwords

- Functional
- Pure
- Lazy
- Strong static typing
- Type polymorphism
- Type classes
- Monads
- Haskell 98 / Haskell 2010
- GHC
  - Glasgow Haskell Compiler
- GADTs
  - Generalized Algebraic Data Types
- STM
- Hackage

History

- Named after the logician Haskell B. Curry
- Designed by a committee aiming to
  - consolidate (lazy) FP languages into a common one
  - develop a language basis for FP language research
- Well crafted and designed pure FP language
  - concise and expressive
  - strong theoretical basis (λ-calculus)
  - sophisticated type system
  - evaluation on demand, at most once (laziness)

Hello, World!

```
-- File: hello.hs
module Main where

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

Not the most representative Haskell program...

- '--' starts a one-line comment
- '::' denotes a type declaration
- '=' defines a function clause
- All but the last line are optional
- Source file names end in ".hs"

Quick sort over lists

```
-- File: qsort.hs
qsort [] = []
qsort (p:xs) =
  qsort [x | x <- xs, x < p] ++
  [p] ++
  qsort [x | x <- xs, x >= p]
```

- **[]** for the empty list
- **(h:t)** notation for a list with head **h** and tail **t**
- Very compact and easy to understand code
- Small letters for variables
- Simpler list comprehensions
- No parentheses or punctuations needed

```
%% Erlang version
qsort([]) -> [];
qsort([P|Xs]) ->
  qsort([X || X <- Xs, X < P]) ++
  [P] ++ % pivot element
  qsort([X || X <- Xs, X >= P]).
```

Another quick sort program

```
-- File: qsort2.hs
qsort [] = []
qsort (p:xs) = qsort lt ++ [p] ++ qsort ge
    where lt = [x | x <- xs, x < p]
          ge = [x | x <- xs, x >= p]
```

- Equivalent to the previous definition (shown below)
- Which version to prefer is a matter of taste

```
-- File: qsort.hs
qsort [] = []
qsort (p:xs) =
  qsort [x | x <- xs, x < p] ++
  [p] ++
  qsort [x | x <- xs, x >= p]
```

Running the Haskell interpreter

```
$ ghci
GHCi, version 7.4.1: http://www.haskell.org/ghc/ :? for help
Loading package ...  <SNIP>
Loading package base ... linking ... done.
Prelude> 6*7
42
Prelude> :quit
Leaving GHCi.
$
```

- The Glasgow Haskell interpreter is called ‘`GHCi`'
- The interactive shell lets you write any Haskell expressions and run them
- The “**`Prelude>`**” means that this library is available
- To exit the interpreter, type “**`:quit`**” (or “**`:q`**” or “**`^D`**”)

Loading and running a program

```
$ ghci
GHCi, version 7.4.1: http://www.haskell.org/ghc/ :? for help
Loading package ...  <SNIP>
Loading package base ... linking ... done.
Prelude> :load qsort.hs
[1 of 1] Compiling Main             ( qsort.hs, interpreted )
Ok, modules loaded: Main.
*Main> qsort [5,2,1,4,2,5,3]
[1,2,2,3,4,5,5]
```

- Use "**`:load`**" (or "**`:l`**") to load a file in the interpreter

Functions and values

```
len [] = 0
len (x:xs) = len xs + 1

nums = [17,42,54]
n = len nums
```

As we will soon see, functions *are* values!

- Functions are written as equations (no **`fun`** keywords)
- Their definitions can consist of several clauses
- Function application is written without parentheses
- We can define values and apply functions to them
- Local definitions using **`let`** expressions or **`where`** clauses

```
nums = [17,42,54]
n = let len [] = 0
        len (x:xs) = len xs + 1
    in  len nums
```

```
nums = [17,42,54]
n = len nums
  where len [] = 0
        len (x:xs) = len xs + 1
```

Layout matters!

- Note the spaces: all clauses of a function need to be aligned

```
nums = [17,42,54]
n = let len [] = 0
        len (x:xs) = len xs + 1
    in  len nums
```

- On the other hand, the following is not legal

```
nums = [17,42,54]
n = let len [] = 0
      len (x:xs) = len xs + 1
    in len nums
```

- One can also write

```
nums = [17,42,54]
n = let { len [] = 0; len (x:xs) = len xs + 1 }
    in  len nums
```

Pattern matching

```
len [] = 0
len (x:xs) = len xs + 1
```

- Function clauses are chosen by pattern matching
- Pattern matching also available using **`case`** expressions

```
len ls = case ls of
           [] -> 0
           x:xs -> len xs + 1
```

- Strong static typing ensures the above is equivalent to

```
len ls = case ls of
           x:xs -> len xs + 1
           _ -> 0
```

Pattern matching (cont.)

```
-- take first N elements from a list
take 0 ls = []
take n [] = []
take n (x:xs) = x : take (n-1) xs
```

- Pattern matching can involve ‘multiple’ arguments
- But no repeated variables in patterns (as in ML)
- Pattern matching can also be expressed with **case**

```
-- equivalent definition using case
take n ls =
  case (n, ls) of
    (0, _)     -> []
    (_, [])    -> []
    (n, x:xs) -> x : take (n-1) ls
```

Note: All branches of a **case** have to return the same type

Pattern matching and guards

- Pattern matching can also involve guards

```
-- a simple factorial function
fac 0 = 1
fac n | n > 0 = n * fac (n-1)
```

This clause will match only for positive numbers

```
Prelude> :l factorial.hs
[1 of 1] Compiling Main             ( factorial.hs, interpreted )
Ok, modules loaded: Main.
*Main> fac 3
6
*Main> fac 42
1405006117752879898543142606244511569936384000000000
*Main> fac (-42)
*** Exception: factorial.hs:(2,1)-(3.31):
        Non-exhaustive patterns in function fac
```

No “match non exhaustive” warnings; runtime errors instead

More on guards

- More than one clauses can contain guards

```
-- returns the absolute value of x
abs x | x >= 0 = x
abs x | x < 0 = -x
```

- We can abbreviate repeated left hand sides

```
-- returns the absolute value of x
abs x | x >= 0 = x
      | x < 0 = -x
```

- Haskell also has **`if-then-else`**

```
-- returns the absolute value of x
abs x = if x >= 0 then x else -x
```

Type annotations

```
len :: [a] -> Integer
len [] = 0
len (x:xs) = len xs + 1

nums :: [Integer]
nums = [17,42,54]

n :: Integer
n = len nums
```

- Every function and value has an associated type
- This type can be (optionally) supplied by the programmer in the form of an annotation
- Note the variable in the type of **`len`** (a polymorphic type)

Type notation

- Integer, String, Float, Double, Char, ... Base types
- [X] A list of values of type X
- X -> Y A function from X values to Y values
- (X,Y,Z) A 3-tuple with an X, a Y and a Z value
- ...

```
pair_sum :: (Integer,Integer) -> Integer
pair_sum (a,b) = a + b

triple :: (Integer,(String,Integer),[Char])
triple = (17,("foo",42),['b','a','r'])
```

Type inference

- A type annotation is a contract between the author and the user of a function definition
- In Haskell, writing type annotations is *optional*
  - the compiler will infer types and detect inconsistencies
  - in fact, it will infer the best possible type (principal type)
- Still, providing type annotations is recommended
  - to enhance readability of programs
  - especially when the intended meaning of functions is not "immediately obvious"
- But, as we will see, often Haskell infers better types than those we would normally write by hand

User defined types

We can create new types by enumerating constants and constructors (they need to start with uppercase)

```
data Color = Green | Yellow | Red

next Green = Yellow
next Yellow = Red
next Red = Green
```

```
data Shape = Rectangle Double Double
           | Circle Double

area (Rectangle x y) = x * y
area (Circle r) = 3.14159265 * r * r
```

A type used in another type (such as `Double` above) has to be wrapped in a constructor – why?

Constructors vs. pattern matching

- Constructors are a special kind of functions that construct values

  e.g. **`Rectangle 3.0 2.0`** constructs a **`Shape`** value

- Constructors have types!

  e.g. **`Rectangle :: Double -> Double -> Shape`**

- Pattern matching can be used to “destruct” values

  e.g. below we define a function that can extract the first (**`x`**) component of a **`Rectangle`** value

```
getX (Rectangle x y) = x
```

Recursive data types

- Type definitions can be recursive

```
data Expr = Const Double
          | Add Expr Expr
          | Neg Expr
          | Mult Expr Expr

eval :: Expr -> Double
eval (Const c) = c
eval (Add e1 e2) = eval e1 + eval e2
eval (Neg e) = - eval e
eval (Mult e1 e2) = eval e1 * eval e2
```

```
eval (Mult (Const 6.0) (Add (Const 3.0) (Const 4.0)))
  ⇒ ... ⇒ 42.0
```

Parameterized types

```
data Expr = Const Double
          | Add Expr Expr
          | Neg Expr
          | Mult Expr Expr
```

- Type definitions can also be parameterized

```
data Expr a = Const a
            | Add (Expr a) (Expr a)
            | Neg (Expr a)
            | Mult (Expr a) (Expr a)

type DoubleExpr = Expr Double
```

- Now **`Expr`** is a parameterized type:
  - It takes a type as “argument” and “returns” a type

Parameterized types (cont.)

- Another parameterized type definition

```
data Tree a = Empty | Node a (Tree a) (Tree a)

Empty :: Tree a
Node  :: a -> Tree a -> Tree a -> Tree a

depth :: Tree a -> Integer
depth Empty = 0
depth (Node x l r) = 1 + max (depth l) (depth r)
```

- Types can be parameterized on more type variables

```
type Map a b = [(a,b)]
```

```
data Pair a = Duo a a
```

constraints **Duo** to have two elements of the same type

Type synonyms

- Synonyms for types are just abbreviations
- Defined for convenience

```
type String = [Char]

type Name = String
data OptAddress = None | Addr String
type Person = (Name,OptAddress)
```

A note on names: The naming style we have been using is mandatory
• Type names and constructor names begin with an uppercase letter
• Value names (and type variables) begin with a lowercase letter

Higher order functions

- Functions are first class values
- They can take functions as arguments and return functions as results

Type variables

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs

nums = [17,42,54]
inc x = x + 1
more_nums = map inc nums
```

- Function application associates to the left

```
f x y = (f x) y
```

Currying

```haskell
add_t :: (Integer,Integer) -> Integer
add_t (x,y) = x + y

add_c :: Integer -> Integer -> Integer
add_c x y = x + y

add42 = add_c 42
```

- **add_t** takes a *pair* of integers as argument and returns their sum
- **add_c** takes one integer as argument and returns a function that takes another integer as argument and returns their sum (*curried* version)

Anonymous functions

- A **λ-abstraction** is an anonymous function
- Math syntax:

  λx.exp where *x* is a variable name and *exp* is an expression that may use *x*

- Haskell syntax:

  `\`*x* `->` *exp*

- Two examples:

```
inc42 x = x + 42    ≈  inc42 = \x -> x + 42
add x y = x + y     ≈  add = \x -> \y -> x + y
                    ≈  add = \x y -> x + y
```

Infix operators

- Infix operators (e.g. **+** or **++**) are just "binary" functions

```
x + y  ≈  (+) x y
```

- "Binary" functions can be written with an infix notation

```
add x y  ≈  x `add` y
```

- Apart from the built-in operators, we can define our own
  - Infix operators are built from non-alphanumeric characters

```
[] @@ ys = ys
(x:xs) @@ ys = x : (ys @@ xs)
```

  - Operator precedence and associativity can be specified with "*fixity declarations*"

Strictly, there are no binary functions in Haskell as all functions have only one argument...

Infix operators & partial application

Even infix operators can be applied partially

```
Prelude> map (42 +) [1,2,3]
[43,44,45]
Prelude> map (+ 42) [1,2,3]
[43,44,45]
Prelude> map ("the " ++) ["dog","cat","pig"]
["the dog","the cat","the pig"]
Prelude> map (++ " food") ["dog","cat","pig"]
["dog food","cat food","pig food"]
```

Notice that for a non-commutative operator order matters!
(as shown for **++** above or as shown for **/** below)

```
Prelude> map (/ 2) [1,2,3]
[0.5,1.0,1.5]
Prelude> map (2 /) [1,2,3]
[2.0,1.0,0.6666666666666666]
```

Function composition

- Function composition is easy (and built-in)

```
-- same as the built-in operator . (dot)
compose f g = \x -> f (g x)
```

```
*Main> compose fac length "foo"
6
*Main> (fac . length) "foobar"
720
```

- Composition is not commutative
- What is the type of function composition?

```
*Main> :type compose
compose :: (b -> c) -> (a -> b) -> a -> c
```

Haskell standard **Prelude**

- A library containing commonly used definitions
- Examples:

```
type String = [Char]
```

```
data Bool = False | True
```

```
True  && x = x
False && _ = False
```

```
[] ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)
```

- The core of Haskell is quite small
- In theory, everything can be reduced to λ-calculus

List comprehensions

- Lists are pervasive in Haskell (as in all FP languages...)
- List comprehensions are a convenient notation for list manipulation
- Recall

```
lt = [y | y <- xs, y < x]
```

which means the same as

```
lt = concatMap f xs
       where
         f y | y < x = [y]
             | otherwise = []
```

(**concatMap** is defined in the **Prelude**)

List comprehensions (cont.)

- List comprehensions can have multiple generators

```
-- finds all Pythagorian triples up to n
pythag :: Int -> [(Int,Int,Int)]
pythag n =
  [(x,y,z) | x <- [1..n], y <- [x..n],
             z <- [y..n], x^2 + y^2 == z^2]
```

```
*Main> pythag 13
[(3,4,5),(5,12,13),(6,8,10)]
*Main> pythag 17
[(3,4,5),(5,12,13),(6,8,10),(8,15,17),(9,12,15)]
```

- Note that any list-producing expression can be used as a generator, not just explicit lists
- Similarly, any Boolean expression can be used as a filter

The lists zip operation

- The function **`zip`** takes two lists as input (curried) and returns a list of corresponding pairs

```
zip (x:xs) (y:ys) = (x,y) : zip xs ys
zip [] ys = []
zip xs [] = []
```

- Two examples:

```
Prelude> zip [17,42,54] ['a','b','c']
[(17,'a'),(42,'b'),(54,'c')]
Prelude> zip [1,2,3,4] ['A'..'Z']
[(1,'A'),(2,'B'),(3,'C'),(4,'D')]
```

Abstractions using HO functions

- These two functions perform a similar traversal of the list, but apply different operations to elements

```
sum [] = 0
sum (x:xs) = x + sum xs

prod [] = 1
prod (x:xs) = x * prod xs
```

very common technique in FP languages

- We can abstract the traversal part and separate it from the operations

```
foldr op init [] = init
foldr op init (x:xs) = x `op` foldr op init xs

sum  = foldr (+) 0
prod = foldr (*) 1
```

foldr op init [x1,x2,...,x42] ⇒
(x1 `op` (x2 `op` ... (x42 `op` init) ...

More **foldr** fun

Using **foldr** we can obtain very concise definitions of many common list functions

```
and = foldr (&&) True
concat = foldr (++) []
```

```
xs ++ ys = foldr (:) ys xs
```

```
reverse = foldr (\y ys -> ys ++ [y]) []
```

```
maximum (x:xs) = foldr max x xs
```

Syntactic redundancy

Expression style	vs.	**Declaration style**
each function is defined as one expression	⇔	each function is defined as a series of equations
`let`	⇔	**`where`**
λ	⇔	arguments on the left hand side of **`=`**
`case`	⇔	function level pattern matching
`if`	⇔	guards

Terminology review

Higher-order function: a function that takes another function as argument and/or returns one as a result

Polymorphic function: a function that works with arguments of many possible types

Type scheme: a type that involves type variables

- the type of a polymorphic function is a type scheme

Parameterized type: a type that takes another type as “argument” and “returns” a type

- their constructors are often polymorphic functions