Haskell Summary

Functions

- A function takes 1 or more parameter and produces a result : <function name> <parameter list> = <function body > . Example:

```
double x = x + x
```

- The act of calling a function is known as applying the function to arguments. Examples :

```
double 3 = 3 + 3                double (double 2) = double 2 + double 2
         = 6                                      = (2 + 2) + double 2
                                                  = 4 + double 2
                                                  = 4+4  = 8 (5 steps)
```

- Step order does not usually influence the final result, but may impact on the number of steps and the fact that the stepping process terminates.

Functional Programming

It's a programming style (paradigm) in which functions are the (only) building blocks of a program (pure functional programming). A functional language supports and encourages functional style programming.

Properties of Functional languages

- Declarativeness : A program expresses what it does and not how it does it.
- Referential transparency : Variables can be replaced by their values and vice versa.
- Highorder functions : Functions taking other functions as arguments and returning new functions as results.
- Polymorphism : A single function can satisfy different input and output types.
- Conciseness : compact but very readable
- Strongly typed : with type inference
- list comprehensions : lists of the form [x|x satisfies a given property]
- Lazy evaluation :evaluates expressions only when results are actually needed
- Monadic effects : controls side effects without compromising unction purity
- Partial function instantiation : functions with partially specified arguments
- Class typing : grouping of types into classes with similar properties
- Supports reasoning about programs : functions can be manipulated as mathematical ones

Function application

Function application is denoted using spaces to separate the arguments, and a multiplication is denoted using *: Example: `f a b + c * d`
Also, functions have precedence over all other operators. Hence, we write `f a (b+1)` to mean f (a, b + 1); otherwise (f a b) + 1 is understood.
Operator identifiers use only a set of special characters: `!, #, $, %, &, *, +, ., /, <, =, >, ?, @, \, ^, |, -, ~` . Operators can also be used as functions: Example: `(+) 3 4` is the functional application of 3 + 4.
Naming conventions : Function and parameter identifiers must begin with a lower-case letter, but may then be followed by letters, digits, underscores, and forward single quotes. Function and parameter identifiers must not be one of Haskell's keywords.

Comments

"--" introduce a line comment extending to the end of the current line. Nested comments are delimited by `{-` and `-}` and may span multiple lines. Note that these comments are well formed (not as C's /* */).

Types and Classes

A type is a collection of related values combined with a set of operations. Example s

- type `Bool` has 2 logical values: `True` and `False`
- type `Bool -> Bool` is a function that maps arguments from `Bool` to results from `Bool`.

Applying a function to 1 or more arguments of the wrong type is called a type error. Example: `1 + False` yields a type error
Built-in types **Bool** (`True, False`) , **Char** (`'\a', '-', '\'', '\t', '\n'`) , **String** (`"abc", "a\"c"`) , **Int** (fixed-precision integers in the range -2^{31} to $2^{31}-1$) , **Integer** (arbitrary-precision integers) , **Float** (single precision floating-point numbers)

Type inference

is a compile time process that determines the type of well-defined expressions.

```
second xs = head (tail xs)
```
Solution: `second :: [a]->a`
```
swap (x,y) = (y,x)
```
Solution: `swap :: (a,b)->(b,a)`
```
pair x y = (x,y)
```
Solution: `pair :: a->b->(a,b)`
```
twice f x = f (f x)
```
Solution: `twice :: (a->a)->a->a`

List types

A list is a sequence of elements of the same type:

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

An empty list is written as []. A singleton list is a list of length 1.
A list can be built using for instance : `[1..10]`

The length of a list is not part of its type.
The type of the elements is unrestricted:

```
[[False,True],[True]] :: [[Bool]]
```

Lists can have an infinite length (because of lazy evaluation).

There are 2 particularly useful functions operating on lists:
`head` returns the first element of a non-empty list;
`tail` returns the list without its first element.

These functions are defined by

```
head ::= [a] -> a
tail ::= [a] -> [a]
```

Tuple types

A tuple is a sequence of elements of possibly different type:

```
(False,True) :: (Bool,Bool)
(False,'a','b') :: (Bool,Char,Char)
```

The type of a tuple encodes its size. The arity of a tuple is the number of components it holds. Tuples of arity 1 are not tuples but expressions enclosed in parentheses.
The type of the tuple's components is unrestricted:

```
([False],'a') :: ([Bool,Char)
```

Function types

A function is a mapping from values of one type to values of another:

```
not :: Bool -> Bool
isDigit :: Char -> Bool
```

The parameters and result types are unrestricted:

```
add :: (Integer, Integer) -> Integer
add (x,y) = x + y
zeroto :: Int -> [Int]
zeroto n = [0..n]
```

Curried functions

Functions with several parameters can also return functions as results:

```
add2 :: Integer -> Integer -> Integer
add2 x y = x + y
```

add2 takes an integer x and returns a function (`add2 x`) which in turn takes an integer y and returns the result x + y .
Example:

```
add2 5 :: Integer -> Integer
```

add and add2 produce the same result but add takes both arguments at the same time whereas add2 takes them one at a time.
Functions taking arguments one at a time are called curried functions.

Curried functions introduce more flexibility than functions on tuples, because new functions can be made by partially applying a curried function. Example:

```
dropChar :: Int -> [Char] -> [Char]
dropChar 0 string = string
dropChar i [] = []
dropChar i (el:els) = dropChar (i-1) els
```

but

```
drop3Chars :: [Char] -> [Char]
drop3Chars = dropChar 3
```

is a specialized dropChar function.

The arrow associates to the right:
`T1 -> T2 -> T3` means `T1 -> (T2 -> T3)`

Consequently, function applications associate to the left:
`f a b c` means `((f a) b) c`

Polymorphic Types

A function is polymorphic if its type contains 1 or more type variables.
Example:

```
length :: [a] -> Int
```

Read as "For any type a, length takes a list of values of type a and returns an integer."

Type variables can be instantiated to different types in different circumstances:

```
length [1,2]
length "abc"
```

Type variables must begin with a lower-case letter and are usually called a, b, c, etc.

Class and overloaded functions

A class is simply a collection of types with similar properties.

Constrained type variables can be instantiated to any type satisfying the constraints:

A polymorphic function is overloaded if its type contains one or more class constraints.
Example:

```
sum :: Num a ==> [a] -> a
```

Read as "For any numeric type a, sum takes a list of values of type a and returns a value of type a".

`sum [1,2,3]` produces 6
`sum [1.1,2.2]` produces 3.3
but `sum ['a','b']` yields a type error

Class constraints are written as C a where C is the class name and a is a type variable; a then becomes an instance of class C.

Basic Haskell Classes

Eq - equality types

- Contains `(==), (/=) :: a -> a -> Bool`
- All built-in types are instances of Eq.
- Type variable a can also define list and tuple types.

Ord - ordered types

- Contains `(<), (>=), (>), (<=) :: a -> a -> Bool`
  `max, min :: a -> a -> a`
- All built-in types are instances of Ord. The Ord class derives from Eq.
- Type variable a can also define list and tuple types (lexical order applies),
  `('a',2)<('b',1) is True` `"ab" < "abc" is True`

Show - showable types (turns values into strings)

- Contains `show :: a -> String`
- Examples
  `show False` produces `"False"`
  `show 'a'` produces `"'a'"`
  `show [1,2,3]` produces `"[1,2,3]"`
  `show "ab"` produces `""ab""`

Read - readable types (turns strings into values)

- Contains `read :: String -> a`
- All built-in types are instances of Read.
- Examples
  `read "False"::Bool`
  converts the string "False" into a Bool and produces the value False.
  `not (read "True")`
  converts the string "True" into a compat. type suitable for not, a Bool.

Num - numeric types

- Contains
  ```
  (+), (*), (-) :: a -> a -> a
  negate, abs, signum :: a -> a
  fromInteger :: Integer -> a
  ```
- Examples
  `negate 3` produces -3 `negate (-3)` produces 3
  `abs (-3)` produces 3 `signum 0` produces 0
  `signum (-3)` produces -1 `signum 3` produces 1
- Note: Integer literals are of any numeric type:
  `2 :: (Num a) => a`

Integral - integral types

- Contains types that are instances of Num, but in addition whose values are integers:
  ```
  quot, rem :: a -> a -> a
  quotRem :: a -> a -> (a, a)
  div, mod :: a -> a -> a
  divMod :: a -> a -> (a, a)
  ```
- Examples
  `mod (-2) 3` produces 1 `(-2) 'mod' 3` produces 1
  `div 3 2` produces 1 `div (-3) 2` produces -2
  `quot 3 2` produces 1 `quot (-3) 2` produces -1

Fractional - fraction types

- Contains types that are instances of Num, but are also non-integral:
  ```
  (/) :: a -> a -> a
  recip :: a -> a
  ```
- Float is an instance of this class. Examples:
  `recip 2.0 gives 0.5`
  `2.5/0.5 gives 5.0`
- Literals of real numbers are instances of class Fractional:
  `2.3 :: (Fractional a) => a`

RealFrac - real types

- Holds functions to convert real numbers to integerals:
  ```
  truncate, round :: (Integral b) => a -> b
  ceiling, floor :: (Integral b) => a -> b
  properFraction :: (Integral b) => a -> (b, a)
  ```
- Derives from classes Read and Integral. Examples:
  `round 2.5 gives 2 ,` `round 2.51 gives 3`
  `floor 2.5 gives 2 ,` `floor (-2.5) gives -3`
  `properFraction (-2.5) gives (-2,-0.5)`

Operators

9		not, negate
	Left	!!
	Right	.
8	Right	^,^^,**
7	Left	*, /, \`div\`, 'quot', 'rem', 'mod'
6		- (unary minus)
	Left	+, -
5	Right	:,++
4		=, /=, <, <= , > , >=
3	Left	&&
2	Left	||

- negate and not are unary functions.
- `(!!) :: [a] -> Int -> a` returns the ith element of a list; element positions are numbered from 0. Example: `[1,2,3] !! 1` returns 2.
- ^, ^^, and ** are exponentiation operators differing by their types:
  `(^) :: (Num a, Integral b) => a -> b -> a`
  (in the expression x^y, y must be > 0)
  `(^^) :: (Fractional a, Integral b) => a -> b -> a`
  `(**) :: (Floating a) => a -> a -> a`
- `(:) :: a -> [a] -> [a]` is the list constructor.
  Example: `1:2:[]` returns `1:[2] = [1,2]`.
- `(++) :: [a] -> [a] -> [a]` is the list concatenation operator.
- && and || are the Boolean ∧ and ∨ operators. In an expression, evaluation stops as soon as the final result is known:
  `True || x` returns True regardless of x,
  `False && f x` returns False without evaluating `f x`
- Finally, Haskell provides a conditional expression, which may be useful when defining functions.
  A conditional expression having the form
  `if e1 then e2 else e3`
  returns the value of e2 if the value of el is True, and e3 if el is False.

Defining functions

In its simplest form, a function is defined as

```
<function> ::= <id> {<parameter >} = <expression>
<parameter > ::= <id>
```

Examples:

```
abs n = if n >= 0 then n else -n
```

```
signum n = if n > 0 then 1
           else if n == 0 then 0
           else -1
factorial n = if n <= 0 then 1
              else n * factorial (n-1)
```

Guarded conditions

As an alternative to conditional expressions, defined using guarded conditionals:

```
abs n | n >= 0 = n
      | otherwise = -n
```

- The symbol | should be read as "such that."
- Guards are evaluated from top to bottom.
- The catch all otherwise is defined as True.

A function with guarded conditions is defined as

```
<function> ::= <id> {<parameter >} <guarded expressions>
<guarded expressions> ::= <guard> {<guard>}
<guard> ::= | (<Boolean condition>|otherwise) = <expression>
```

- Functions with guards are more readable than function bodies
  ```
  signum n | n > 0 = 1
           | n == 0 = 0
           | otherwise = -1
  ```

Pattern matching

- A pattern is a mapping between a function's parameter and its argument value.
- In Haskell, patterns can be applied to literals, tuples and lists.
- For example, function not is defined in Prelude as
  ```
  not False = True
  not True = False
  ```
  When not is applied, the argument is evaluated to first match pattern False. If these indeed match, the associated expression is returned (value True); otherwise the matching pattern process continues with the next pattern and False is returned.
- Pattern clauses are evaluated from top to bottom until a match is found.
- The underscore symbol _ is a *wildcard* pattern that matches any argument value.
- A tuple of patterns is also a pattern:
  ```
  sumPair (a,b)    = a + b
  fst (a,_) = a    -- defined in Prelude
  snd (_,b) = b    -- ditto
  ```
- Every non-empty list is internally constructed by repeated use of the cons operator (:) that adds an element to the head of a list:
  `[1,2,3] means 1:2:3:[]`
- Functions on lists can be defined as (x:xs) patterns:
  `head (x:_) = x` -- defined in Prelude
  `tail (_:xs) = xs` - ditto
- Remarks:
  - Because head and tail are defined as above, these's no accounting for empty lists: `head []` and `tail []` produce an error
  - Pattern (x:xs) must be parenthesized because of the application precedence.

Defining Operators

Operators are defined like functions but have infix notion. For example the Boolean ∧ can be defined as

```
True && True = True        True && False = False
False && True = False      False && False = False
```

However to take advantage of lazy evaluations of the operands, Haskell defines ∧ as

```
True && b = b              False && _ = False
```

This definition is more efficient, because it avoids evaluating the second argument if the first argument is False.

- A declaration of an infix operator together with an indication of its binding power (precedence level) and its associativity is called a fixity declaration.
- There are three kinds of fixity, non-, left- and right-associativi ty (`infix`, `infixl`, and `infixr`, respectively), and ten precedence levels, 0 to 9 (level 9 binds more tightly than level 0).
- If the precedence level is omitted, level 9 is assumed.
- An operator lacking a fixity declaration is assumed to be `infixl 9`.

```
<fixity declaration> ::= <infix> [pre] <opid>{,<opid>}
<pre> ::= (0|1|2|3|4|5|6|7|8|9)
<infix> ::= (infixl|infixr|infix)
```

- Examples: `infixl 3 &&` `infixl 2 ||`

List comprehensions

In mathematics, the comprehension notation defines new sets from existing ones. For example, {x 2 |x ∈ {1..5}} produces {1, 4, 9, 16, 25}.
In Haskell, this example is written as `[x^2 | x<-[1..5]]`
Remarks:

- The expression `x<-[1..5]` is a generator that states how values for x are produced.
- Comprehensions can have multiple generators separated by commas.
  For example, `[(x,y) | x<-[1,2,3], y<-[4,5]]`
  yields `[(1,4),(1,5),(2,4),(2,5),(3,4),(3,5)]`

Multiple generators are akin the nested loops with nesting levels read from left to right.
A generator may also depend on variables introduced by a previous generator:

```
[(x,y) | x<-[1..3], y<-[x..3]]
```

gives `[(1,1),(1,2),(1,3),(2,2),(2,3),(3,3)]`.
List comprehensions can also use predicates (guards) to filter or restrict values produced by generators:

```
[x | x <- [1..10], mod x 2 == 0]
```

gives the list of all numbers x such that x is an element of the list [1..10] and x is even.
From existing list : `[x | x <- xs, mod x 2 == 0]`

Higher-Order Functions

A function is called higher-order if it takes a function as an argument or returns a function as a result.

Example :

```
twice :: (a -> a) -> a -> a
twice f x = f (f x)
```

`twice` is higher-order because it takes a function as its first argument.

Lambda Expressions

Functions can be constructed without naming them by using lambda expressions.
For example,
λx.B(x)
defines a nameless function that takes a parameter x and returns the result given by the expression B(x).
Lambda expressions can be used to avoid naming functions that are only referenced once.

For example:

```
odds n = map f [0..n-1]
  where
    f x = x * 2 + 1
```

can be simplified into

```
odds n = map (\x -> x * 2 + 1) [0..n-1]
```

Sections

A binary operator can be converted into a curried function by enclosing the name of the operator in parentheses. For example:
`1+2` can be written as `(+) 1 2`
This convention also allows one of the operands of the operator to be included in the parentheses. For example:
`(1+) 2` and `(+2) 1` produce the same result.
In general, if ⊕ is an operator, then functions of the form (⊕), (x⊕) and (⊕y) are called sections, and have the following meaning:
(⊕) = λx.(λy.x ⊕ y) (x⊕) = λy.x ⊕ y (⊕y) = λx.x ⊕ y

Useful functions can sometimes be constructed in a simple way using sections. Examples:
`(1+)` - successor function
`(1/)` - reciprocation function
`(*2)` - doubling function
`(/2)` - halving function
Lambda expressions and sections come in handy when defining higher-order functions.

List Processing Functions

The higher-order library function called `map` applies a function to very element of a list: `map :: (a -> b) -> [a] -> [b]`

```
map f [] = []
map f (x:xs) = f x : map f xs
```

Examples: `map (+1) [1,3,5,7]` returns `[2,4,6,8]`
`map isDigit ['a','1','b']` returns `[False,True,False]`
`map` can also be defined in a particularly simple manner using a list comprehension: `map f xs = [f x | x <- xs]`
Another example: `map (map (+1)) [[1,2,3],[4,5]]`
yields `[map (+1) [1,2,3], map (+1) [4,5]]`
which then produces `[[2,3,4],[5,6]]`

- Decide if all elements of a list satisfy a predicate:
  `all :: (a -> Bool) -> [a] -> Bool`
  Example: `all even [2,4,6,8]` returns `True`
- Decide if any element of a list satisfies a predicate:
  any :: (a -> Bool) -> [a] -> Bool
  Example: any odd [2,4,6,8] returns False

The higher-order library function `filter` selects every element from a list that satisfies a predicate:

```
filter :: (a -> Bool) -> [a] -> [a]
filter p [] = []
filter p (x:xs) | p x = x : filter p xs
                | otherwise = filter p xs
```

Example: `filter even [1..10]` produces `[2,4,6,8,10]`
Function `filter` can be defined using a list comprehension:

```
filter p xs = [x | x <- xs, p x]
```

- Select elements from a list while they satisfy a predicate:
  `takeWhile :: (a -> Bool) -> [a] -> [a]`
  Example: `takeWhile isLower "abc def"` returns `"abc"`
- Remove elements from a list while they satisfy a predicate:
  `dropWhile :: (a -> Bool) -> [a] -> [a]`
  Example: `dropWhile isLower "abc def"` returns `" def"`

The foldr Function

A number of functions on lists can be defined using the following simple pattern of recursion:

```
f [] = v
f (x:xs) = x ⊕ f xs
```

f maps the empty list to some value v, and any non-empty list to some operator ⊕ applied to its head and f of its tail. Examples:

```
sum [] = 0
sum (x:xs) = x + sum xs (v = 0 and ⊕ = +)
product [] = 1
product (x:xs) = x * product xs (v = 1 and ⊕ = *)
and [] = True
and (x:xs) = x && and xs (v = True and ⊕ = &&)
```

The higher-order library function `foldr` (fold right) encapsulates this common pattern of recursion, with the function ⊕ and the value v as arguments:

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

Examples:

```
sum = foldr (+) 0
product = foldr (*) 1
and = foldr (&&) True
```

The foldl Function

It is also possible to define recursive functions on lists using an operator that is assumed to associate to the left.

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

`foldl` applies to recursion pattern that have the form:

```
f v [] = v
f v (x:xs) = f (v ⊕ x) xs
```

Example:

```
  foldl (+) 0 [1,2,3]
= foldl (+) 0 (1 : (2 : (3 : [])))
= ((0 + 1) + 2) + 3
```

`foldl` is trickier to use than `foldr`.

Function composition

Although it may seem that a function composition operator (∘ in math and . in Haskell) should play a major role in functional programming, this operator only contributes as a shortcut when defining functions: f ∘ g (read as "f composed with g") is identical to defining (f ∘ g)x by f (g x).
Note that . is right associative and so the last example needs no parenthesizing.

Examples:

```
odd n = not (even n)
twice f x = f (f x)
```

can all be rewritten as

```
odd = not . even
twice f = f . f
```