Haskell Quick Reference

Basic Syntax

```
\ x y z → expr
let var = value in expr
expr where var = value
if a then b else c
case a b c of
  d e f → expr
  ...

a >>= \s → b >> c ≡ do a; s ← b; c

(f x) y = f x y
x op y = (op) x y
(op e) = \x → x op e
(e op) = \x → e op x
-e = negate (e)
(x::Bool)
```

Type Declarations

```
x :: type
f :: x → y
f :: (Eq a) => a → b
type T = (A,B)
data A = A Int Int deriving (Eq,Show)
data A a b = A a b deriving (Eq,Show)
data A = A !Int deriving (Eq,Show)
data U = A | B | C
data A
```

Classes

```
class Num a where
  (+) :: a → a → a
instance Num Int where
  x + y = addInt x y
```

Definitions

```
x = 3
f = \ x → 3*x
f x = 3*x
f x | x>0 = log x
f 0 = 1; f 1 = 1; f n = f (n-1) + f (n-2)
f x | x<=1 = 1 | otherwise = f (n-1) + f (n-2)
f ~(x:xs) = ...

f s@(x:xs) = ...
f (_:xs) = ...
```

Functions

```
id, const
f . g ≡ \x → f (g x)
flip f x y = f y x
seq a b = b
f $ x = f x   f $ g $ h x = f (g (h x))
($ 0) ; zipWith ($) fs xs
f $! x = x `seq` f x
until p f x
asTypeOf = const
error “something”, undefined
```

Bool

```
data Bool = False | True
&&, ||, not, otherwise
```

Char

```
==, <=
toEnum, fromEnum, enumFrom, ...
minBound, maxBound
type String = [Char]
minBound:::Char
(toEnum 117)::Char
```

Numbers

```
Integer, Int ≡ Integral
Ratio a ≡ RealFrac
Float, Double ≡ RealFloat
Complex a  ≡ Floating
```

- +, -, *, negate, abs, signum, fromInteger
- toRational
- quot, rem, div, mod, quotRem, divMod, toInteger
- /, recip, fromRational
- pi, exp, log, sqrt, **, logBase, sin, cos, tan, asin, acos, atan, sinh, cosh, tanh, asinh, acosh, atanh

I/O

- putChar :: Char → IO ()
- putChar, putStr, putStrLn, print
- getChar, getLine, getContents, interact, radIO, readLn
- writeFile, appendFile, readFile
- ioError, catch
- stdin, stdout, stderr
- withFile, openFile, hClose
- IOMode = ReadMode | WriteMode | AppendMode | ReadWRiteMode
- readFile, writeFile, appendFile
- hFileSize, hSetFileSize, hIsEOF, isEOF
- BufferMode = NoBuffering | LineBuffering | BlockBuffering (Just Int)
- hSetBuffering, hGetBuffering, hFlush, hGetPosn, hSetPosn, hSeek, hTell, hIsOpen, hIsClosed, hIsReadable...
- hIsTerminalDevice, hSetEcho, hGetEcho, hShow, hWaitForInput, hReady, hGetChar, hGetLine, hLookAhead
- hGetContents, hPutChar, hPutStr, ...
- withBinaryFile, openBinaryFile, hSetBInaryMode, ...
- TextEncoding ‣ latin1, utf8, ...
- localeEncoding, mkTextEncoding, hSetNewlineMode ‣ LF | CRLF

Lists

```
data [a] = [] | a : [a] deriving (Eq,Ord)
[] 1:[] 1:2:3:[] = [1,2,3]
```

- head, tail, init, last, null, length
- !! ≡ at
- map, filter
- ++, concat, concatMap
- foldl, foldl1, foldr, foldr1
- scanl, scanl1, scanr, scanr1
- iterate, repeat, replicate, cycle
- take, drop, splitAt, takeWhile, dropWhile
- span, break
- lines, words, unlines, unwords
- reverse
- and, or, sum, product, maximum, minimum
- zip, zip3, zipWith, zipWith3, unzip, unzip3

Tuples

```
(1,2,3) = (,,) 1 2 3
fst, snd
curry         f (x,y) = f x y
uncurry  f x y = f (x,y)
```

Other

```
data Maybe a = Nothing | Just a
```

- maybe n f nothing = n
- maybe n f (Just x) = f x
- fmap f Nothing= Nothing
- fmap f (Just x) = Just (f x)
- (Just x) >>= k = k x
- Nothing >>= k = Nothing
- return = Just
- fail s = Nothing

```
data Either a b = Left a | Right b
```

- either f g (Left x) = f x
- either f g (Right y) = g y

```
Ordering = LT | EQ | GT
```

Deriving

Eq
- ==, /=

Ord
- compare, <=, ..., max, min

Bounded
- minBound, maxBound

Enum
- toEnum, fromEnum

Read, Show
- reads, read, readPrec, readList
- shows, show, showList, showPrec, ...

Monad
- >>, >>=, return, fail, =<<
- sequence, sequence_, mapM, mapM_

Module Syntax

```
module Main where
  import A
  import B
  main = A.f >> B.f
module A where
  f = ...
module B where
  f = ...
module M(A,b,module M) where ...
import B(f)
import B hiding (C)
import qualified C(f,g)
import qualified Bar as B

— ADT: constructor not exported
module Stack(S,push,pop,empty) where
  data S a = Empty | Stk a (S a)
  push = ...; pop = ...; empty = ...
```