Basic Haskell Cheat Sheet

Structure

```
func :: type -> type
func x = expr

fung :: type -> [type] -> type
fung x xs = expr

main = do code
          code
          ...
```

Function Application

```
f x y       ≡ (f x) y          ≡ ((f) (x)) (y)
f x y z     ≡ ((f x) y) z      ≡ (f x y) z
f $ g x     ≡ f (g x)          ≡ f . g $ x
f $ g $ h x ≡ f (g (h x))      ≡ f . g . h $ x
f $ g x y   ≡ f (g x y)        ≡ f . g x $ y
f g $ h x   ≡ f g (h x)
```

Values and Types

has type	*expr*	:: *type*
boolean	`True || False`	:: `Bool`
character	`'a'`	:: `Char`
fixed-precision integer	`1`	:: `Int`
integer (arbitrary sz.)	`31337`	:: `Integer`
	`31337^10`	:: `Integer`
single precision float	`1.2`	:: `Float`
double precision float	`1.2`	:: `Double`
list	`[]`	:: `[a]`
	`[1,2,3]`	:: `[Integer]`
	`['a','b','c']`	:: `[Char]`
	`"abc"`	:: `[Char]`
	`[[1,2],[3,4]]`	:: `[[Integer]]`
string	`"asdf"`	:: `String`
tuple	`(1,2)`	:: `(Int,Int)`
	`([1,2],'a')`	:: `([Int],Char)`
ordering relation	`LT, EQ, GT`	:: `Ordering`
function (λ)	`\x -> e`	:: `a -> a`

Values and Typeclasses

given context, has type	*expr*	:: *context* => *type*
Numeric (+,-,*)	`137`	:: `Num a => a`
Fractional (/)	`1.2`	:: `Fractional a => a`
Floating	`1.2`	:: `Floating a => a`
Equatable (==)	`'a'`	:: `Eq a => a`
Ordered (<=, >=, >, <)	`731`	:: `Ord a => a`

Declaring Types and Classes

type synonym	`type MyType = Type`
	`type UserId = Integer`
	`type UserName = String`
	`type User = (UserId,UserName)`
	`type UserList = [User]`

data (single constructor)	`data MyData = MyData Type Type Type` `deriving (Class,Class)`

data (multi constructor)	`data MyData = Simple Type |`
	`Duple Type Type |`
	`Nople`

typeclass	`class MyClass a where`
	`foo :: a -> a -> b`
	`goo :: a -> a`
	`...`

typeclass instance	`instance MyClass MyType where`
	`foo x y = ...`
	`goo x = ...`
	`...`

Operators (grouped by precedence)

List index, function composition	`!!, .`
raise to: Non-neg. Int, Int, Float	`^, ^^, **`
multiplication, fractional division	`*, /`
integral division ($\Rightarrow -\infty$), modulus	`` `div` ``, `` `mod` ``
integral quotient ($\Rightarrow 0$), remainder	`` `quot` ``, `` `rem` ``
addition, subtraction	`+, -`
list construction, append lists	`:, ++`
list difference	`\\`
comparisons:	`>, >=, <, <=, ==, /=`
list membership	`` `elem` ``, `` `notElem` ``
boolean and	`&&`
boolean or	`||`
sequencing: bind and then	`>>=, >>`
application, strict apl., sequencing	`$, $!, `` `seq` ``

NOTE: Highest precedence (first line) is 9, lowest precedence is 0. Those aligned to the right are right associative, all others left associative: except comparisons, list membership and list difference which are non-associative. Default is `infixl 9`.

Defining fixity

non associative fixity	`infix 0-9 `op``
left associative fixity	`infixl 0-9 +--+`
right associative fixity	`infixr 0-9 -!-`
default, implied when no fixity given	`infixl 9`

Functions ≡ Infix operators

```
f a b   ≡ a `f` b
a + b   ≡ (+) a b
(a +) b ≡ ((+) a) b
(+ b) a ≡ \x -> ((+) x b)) a
```

Common functions

Misc

```
id        :: a -> a                 id x ≡ x   -- identity
const     :: a -> b -> a    (const x) y ≡ x
undefined :: a                undefined ≡ ⊥ (lifts error)
error     :: String -> a       error cs ≡ ⊥ (lifts error cs)
not       :: Bool -> Bool      not True ≡ False
flip      :: (a -> b -> c) -> (b -> a -> c)
                            flip f $ x y ≡ f y x
```

Tuples

```
fst   :: (a, b) -> a            fst (x,y) ≡ x
snd   :: (a, b) -> b            snd (x,y) ≡ y
curry :: ((a, b) -> c) -> a -> b -> c
                      curry (\(x,y) -> e) ≡ \x y -> e
uncurry :: a -> b -> c -> ((a, b) -> c)
                      uncurry (\x y -> e) ≡ \(x,y) -> e
```

Lists

```
null    :: [a] -> Bool              null [] ≡ True  -- empty?
head    :: [a] -> a           head [x,y,z,w] ≡ x
tail    :: [a] -> [a]         tail [x,y,z,w] ≡ [y,z,w]
init    :: [a] -> [a]         init [x,y,z,w] ≡ [x,y,z]
reverse :: [a] -> [a]        reverse [x,y,z] ≡ [z,y,x]
take    :: Int -> [a] -> [a]  take 2 [x,y,z] ≡ [x,y]
drop    :: Int -> [a] -> [a]  drop 2 [x,y,z] ≡ [z]
length  :: [a] -> Int         length [x,y,z] ≡ 3
elem    :: a -> [a] -> Bool   y `elem` [x,y] ≡ True  -- ∈?
repeat  :: a -> [a]                repeat x ≡ [x,x,x,...]
cycle   :: [a] -> [a]              cycle xs ≡ xs++xs++...
```

Special folds

```
and     :: [Bool] -> Bool          and [p,q,r] ≡ p && q&&r
or      :: [Bool] -> Bool           or [p,q,r] ≡ p || q||r
sum     :: Num a => [a] -> a       sum [i,j,k] ≡ i + j + k
product :: Num a => [a] -> a   product [i,j,k] ≡ i * j * k
concat  :: [[a]] ->[a]       concat [xs,ys,zs] ≡ xs++ys++zs
maximum :: Ord a => [a] -> a  maximum [10,0,5] ≡ 10
minimum :: Ord a => [a] -> a  minimum [10,0,5] ≡ 0
```

Higher-order / Functors

```
map    :: (a->b) -> [a] -> [b]
                        map f [x,y,z] ≡ [f x, f y, f z]
filter :: (a -> Bool) -> [a] -> [a]
                  filter (/=y) [x,y,z] ≡ [x,z]
foldl  :: (a -> b -> a) -> a -> [b] -> a
                      foldl f x [y,z] ≡ (x `f` y) `f` z
foldr  :: (a -> b -> b) -> b -> [a] -> b
                      foldr f z [x,y] ≡ x `f` (y `f` z)
```

Numeric

```
abs       :: Num a => a -> a          abs -10 ≡ 10
even, odd :: Num a => a -> Bool      even -10 ≡ True
gcd, lcm  :: Integral a => a -> a -> a
                                     gcd 4 2 ≡ 2
recip     :: Fractional a => a -> a   recip x ≡ 1/x
pi        :: Floating a => a               pi ≡ 3.1415...
sqrt, log :: Floating a => a -> a      sqrt x ≡ x**0.5
exp, sin, cos, tan, asin, acos, atan  :: Floating a => a -> a
truncate, round :: (RealFrac a, Integral b) => a -> b
ceiling, floor  :: (RealFrac a, Integral b) => a -> b
```

Strings

```
lines   :: String -> [String]
                lines "ab\ncd\ne" ≡ ["ab","cd","e"]
unlines :: [String] -> String
           unlines ["ab","cd","e"] ≡ "ab\ncd\ne\n"
words   :: String -> [String]
                  words "ab cd e" ≡ ["ab","cd","e"]
unwords :: [String] -> String
          unwords ["ab","cd","ef"] ≡ "ab cd ef"
```

Read and Show classes

```
show :: Show a => a -> String    show 137 ≡ "137"
read :: Show a => String -> a    read "2" ≡ 2
```

Ord Class

```
min     :: Ord a => a -> a -> a         min 'a' 'b' ≡ 'a'
max     :: Ord a => a -> a -> a        max "b" "ab" ≡ "b"
compare :: Ord a => a -> a -> Ordering  compare 1 2 ≡ LT
```

Libraries / Modules

```
importing              import PathTo.Lib
importing (qualified)  import PathTo.Lib as PL
importing (subset)     import PathTo.Lib (foo,goo)
declaring              module Module.Name
                         ( foo
                         , goo
                         )
                       where
                       ...
./File/On/Disk.hs      import File.On.Disk
```

Tracing and monitoring (unsafe) — Debug.Trace

```
Print string, return expr    trace string $ expr
Call show before printing    traceShow expr $ expr
Trace function call values   fun x y | traceShow (x,y) False = undefined
                             fun x y = ...
```

IO – Must be "inside" the IO Monad

```
Write char c to stdout                          putChar c
Write string cs to stdout                       putStr cs
Write string cs to stdout w/ a newline          putStrLn cs
Print x, a show instance, to stdout             print x
Read char from stdin                            getChar
Read line from stdin as a string                getLine
Read all input from stdin as a string           getContents
Bind stdin/stdout to foo (:: String -> String)  interact foo
Write string cs to a file named fn              writeFile fn cs
Append string cs to a file named fn             appendFile fn cs
Read contents from a file named fn              readFile fn
```

Pattern Matching

Simple Pattern Matching

```
Number 3        3        Character 'a'   'a'
Empty string    ""       Ignore value    _
```

List Pattern Matching

```
empty list                        []
head x and tail xs                (x:xs)
tail xs (ignore head)             (_:xs)
list with 3 elements a, b and c   [a,b,c]
list where 2nd element is 3       (x:3:xs)
```

Patterns for Tuples and Other Types

```
pair values a and b               (a,b)
ignore second element of tuple    (a,_)
triple values a, b and c          (a,b,c)
just constructor                  Just a
nothing constructor               Nothing
user-defined type                 MyData a b c
ignore one of the "components"    MyData a _ c
match first tuple on list         ((a,b):xs)
```

As-pattern

```
match entire tuple s its values a,b          s@(a,b)
match entire list a its head x and tail xs   a@(x:xs)
entire data p and "components"               p@(MyData a b c)
```

List Comprehensions

Take *pat* from *list*. If *boolPredicate*, add element *expr* to list:

```
[expr | pat <- list, boolPredicate, ...]
```

```
[x | x <- xs]                ≡ xs
[f x | x <- xs, p x]         ≡ map f $ filter p xs
[x | x <- xs, p x, q x]      ≡ filter q $ filter p xs
[f x y | x <- xs, y <- ys]   ≡ zipWith f xs ys
[x+y | x <- [a,b], y <- [i,j]] ≡ [a+i, a+j, b+i, b+j]
```

Expressions / Clauses

if expression ≈ **guarded equations**

```
if boolExpr                  foo ... | boolExpr  = exprA
  then exprA                         | otherwise = exprB
  else exprB
```

nested if expression ≈ **guarded equations**

```
if boolExpr1                 foo ... | boolExpr1 = exprA
  then exprA                         | boolExpr2 = exprB
    else if boolExpr2                | otherwise = exprC
           then exprB
           else exprC
```

case expression ≈ **function pattern matching**

```
case x of                    foo pat1 = exprA
  pat1 -> exprA              foo pat2 = exprB
  pat2 -> exprB              foo _    = exprC
  _    -> exprC
```

2-variable case expression ≈ **function pattern matching**

```
case (x,y) of                foo pat1 patA = exprA
  (pat1,patA) -> exprA       foo pat2 patB = exprB
  (pat2,patB) -> exprB       foo _    _    = exprC
  _           -> exprC
```

let expression ≈ **where clause**

```
let nameA=exprA                foo ... = mainExpression
    nameB=exprB                  where nameA=exprA
    ...                                nameB=exprB
in mainExpression                      ...
```

do notation ≈ **desugarized do notation**

```
do statement                 statement >>
   pat <- exp                exp >>= \pat ->
   statement                   statement >>
   pat <- exp                  exp >>= \pat ->
   ...                           ...
```

```
statement separator   ;     -- or line break
statement grouping    { }   -- or layout/indentation
```

GHC - Glasgow Haskell Compiler (and Cabal)

```
compiling program.hs                    $ ghc program.hs
running                                 $ ./program
running directly                        $ run_haskell program.hs
interactive mode (GHCi)                 $ ghci
GHCi load                               > :l program.hs
GHCi reload                             > :r
GHCi activate stats                     > :set +s
GHCi help                               > :?
Type of an expression                   > :t expr
Info (oper./func./class)                > :i thing
install package pkg                     $ cabal install pkg
update package list                     $ cabal update
list/search for packages matching pat   $ cabal list pat
information about package pkg           $ cabal info pkg
```