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

Binding Types

has type	*expr* `::` *type*
boolean	`True || False :: Bool`
character	`'a' :: Char`
integer (32-bit)	`1 :: Int`
	`3 + 2 :: Int`
integer (arbitrary sz.)	`31337 :: Integer`
	`31337^10 :: Integer`
single precision float	`1.2 :: Float`
double precision float	`1.2 :: Double`
list	`[] :: [a]`
	`['a','b','c'] :: [Char]`
	`"abc" :: [Char]`
	`[[1,2],[3,4]] :: [[Integer]]`
tuple	`(1,2) :: (Int,Int)`
	`([1,2],'a') :: ([Int],Char)`
string	`"asdf" :: String`
functions	`foo :: a -> a`
	`double :: Int -> Int`

Binding Classes (Typeclasses)

Numeric (+,-,*,/)	`137 :: Num a => a`
Floating	`1.2 :: Floating a => a`
Fractional	`1.2 :: Fractional a => a`
Equatable (==)	`'a' :: Eq a => a`
Ordered (<=, >=, >, <)	`731 :: Ord a => a`
	`sort :: Ord a => [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`
	`  ...`

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	`:`, `++`
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 boolean comparisons and list membership, 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

Lists (and Strings (which are lists...))

size / length of `xs`	`length xs`
invert / reverse of `xs`	`reverse xs`
head / first element of `xs`	`head xs`
checks for `x` in `xs`	`x `elem` xs`
	`elem x xs`
sorts `xs`	`sort xs :: Ord a => [a]`
pairs of elements from `xs` and `ys`	`zip xs ys :: [a] -> [b] -> [(a, b)]`

Tuples

first of pair `p`	`fst p`
second of pair `p`	`snd p`
swap pair `p`	`swap p`

Higher-order / Functors

apply `f` for each `x` in `xs` returning new list	`map f xs :: (a -> b) -> [a] -> [b]`
fold - (`z` + left) `to` right	`foldl f z xs :: (a -> b -> a) -> a -> [b] -> a`
fold - right `to` (left + `z`)	`foldr f z xs :: (a -> b -> b) -> b -> [a] -> b`
filter all `xs` satisfying `p xs`	`filter p xs :: (a -> Bool) -> [a] -> [a]`

IO – Must be "inside" the IO Monad

Write char `c` to stdout	`putChar c`
Write string `cs` to stdout	`putStr cs`
... `cs` ... with a newline	`putStrLn cs`
Print `x`, a `show` instance[2], 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`
Make foo process the input	`interact foo :: (String -> String) -> IO ()`
Write char `c` to channel/file `h`	`hPutChar h c`
Write string `cs` to channel/file `h`	`hPutStr h cs`
... `cs` ... with a newline ... to `h`	`hPutStrLn h cs`

Pattern Matching

List Pattern Matching

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

Other Types Pattern Matching

pair values a and b	`(a,b)`
triple values a, b and c	`(a,b,c)`
just constructor	`Just a`
nothing constructor	`Nothing`
user-defined type	`MyData a b c`

Wildcard Pattern "Matching"

ignore value	`_`
ignore first elements of list	`(_:xs)`
ignore second element of tuple	`(a,_)`
ignore one of the "componenet"	`MyData a _ c`

Nested Pattern

match first tuple on list	`((a,b):xs)`
match list inside tuple	`(xs,y:ys,zs)`

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

pairs where sum=4

```
[(x,y) |
  x <- [0..4],
  y <- [0..4],
  x + y == 4]
== [(0,4),(1,3),(2,2),...]
```

Expressions (Eval. control)

statement separator
statement grouping

```
;    -- or line break
{ }  -- or layout/indentation
```

if expression

```
if expr :: Bool
  then expr :: a
  else expr :: a
```

case expression

```
case expr of
  pat -> expr
  pat -> expr
  ...
  _ -> expr
```

let expression

```
let name=expr
    name=expr
    ...
in expr
```

where notation

```
expr
where name=expr
      name=expr
      ...
```

do notation

```
do statement
   pat <- exp
   statement
   pat <- exp
   ...
```

pattern matching *(case sugar)*

```
f :: a -> b -> c
f pat pat = expr
f _   pat = expr
f pat _   = expr
f _   _   = expr
```

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
```

QuickCheck — Test.Quickcheck

declaring property

```
prop_something :: a -> Bool
prop_something :: a -> Property
```

verifying property

```
quickCheck prop_something
```

SmallCheck — Test.SmallCheck

verifying property

```
smallCheck depth prop_something
```

HUnit — Test.HUnit

equality assertion

```
expected ~=? actual
```

testlist

```
mytestlist =
  TestList [ expec ~=? actual
           , expec ~=? actual
           ...
           , expec ~=? actual ]
```

running tests

```
runTestTT mytestlist
```

GHC - Glasgow Haskell Compiler

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 program.hs`
GHCi activate stats	`> :set +s`
GHCi turn off stats	`> :unset +s`
GHCi help	`> :?`
Type of an expression	`> :t expr`
Info (oper./func./class)	`> :i thing`