

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
fixed-precision integer 1 :: Int
integer (arbitrary sz.) 31337 :: Integer
               3133710 :: 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]
Bounded (minBound,maxBound) minBound :: Int
```

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 \equiv 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  map f xs
returning new list       :: (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 instance2, 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 “component”	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	;	-- or line break
statement grouping	{ }	-- or layout/indentation

if expression	if <i>expr</i> :: Bool
	then <i>expr</i> :: a
	else <i>expr</i> :: a

case expression	case <i>expr</i> of
	<i>pat</i> -> <i>expr</i>
	<i>pat</i> -> <i>expr</i>
	...
	_ -> <i>expr</i>

let expression	let <i>name=expr</i>
	<i>name=expr</i>
	...
	in <i>expr</i>

where notation	<i>expr</i>
	where <i>name=expr</i>
	<i>name=expr</i>
	...

do notation	do <i>statement</i>
	<i>pat</i> <- <i>exp</i>
	<i>statement</i>
	<i>pat</i> <- <i>exp</i>
	...

pattern matching	f :: a -> b -> c
(<i>case sugar</i>)	f pat pat = expr
	f _ pat = expr
	f pat _ = expr
	f _ _ = expr

Libraries / Modules

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

QuickCheck

Test.Quickcheck

declaring property	<i>prop.something</i> :: a -> Bool
	<i>prop.something</i> :: a -> Property
verifying property	quickCheck <i>prop.something</i>

SmallCheck

Test.SmallCheck

verifying property	smallCheck <i>depth prop.something</i>
--------------------	--

HUnit

Test.HUnit

equality assertion	<i>expected</i> ~=? <i>actual</i>
testlist	<i>mytestlist</i> =
	TestList [<i>expec</i> ~=? <i>actual</i>
	, <i>expec</i> ~=? <i>actual</i>
	...
	, <i>expec</i> ~=? <i>actual</i>]
running tests	runTestTT <i>mytestlist</i>

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