Cheatography

Haskell Cheat Sheet

by xonnex via cheatography.com/196187/cs/41216/

Class Constraints

Eq – equality types

Contains types whose values can be compared for equality and inequality

methods: (==), (/=)

Ord – ordered types

Contains types whose values are totally ordered

methods: (<), (<=), (>), (>=), min, max

Show – showable types

Contains types whose values can be converted into strings of characters

method show :: a -> String

Num – numeric types

Contains types whose values are numeric

methods: (+), (-), (*), negate, abs, signum

Integral – integral types

Contains types that are numeric but of integral value

methods: div, mod

Fractional – fractional types

Contains types that are numeric but of fractional value

methods: (/), recip

Predefined Functions

cos

sin

fst	first argument
snd	second argument
show	display (will display "\246" instead of "ö")
putStrLn	IO display (will display "ö")

import Data.Char

isDigit 'a'

isUpperCase 'a'

Predefined Constants

pi

Pattern Matching

fstInt :: (Int, Int) -> Int

fstInt (x, y) = x

fstInt (1, 3) 1

sayNumber :: Int -> String -> String

sayNumber 1 s = "One " ++ s

sayNumber n s = "Many " ++ s ++ "s"

constants like 0, [] or an enum names like n

wildcard "_" (matches always but binds no name to the matched value)

structures like lists (x:xs) or tuples (a,b)

Commands

cabal run	Runs cabal project
ghci	Open interactive shell
ghci fileName.hs	Open shell and load file

Functions

data Color = Red | Yellow | Green deriving (Show)

data ToDo = Stop | Wait | Go deriving (Show)

atTrafficLight :: Color -> ToDo

atTrafficLight Red = Stop

atTrafficLight Yellow = Wait

atTrafficLight Green = Go

"->" whenever the arrow is shown, we have ourselves a function

ghci

:r	reload file
:q	quit
:t var	show type of var
:i +	show type of operator
:{	start multiline
:}	end multiline
(var1, var2, ...) :: (Type1, Type2, ...)	Validates type
{-# OPTIONS_GHC -Wall #-}	Shows info in case something is missing

Bool

True

False

a && b

a || b

not a

Lists

[1,2,3,4]

[(1,2),(3,4)]

ones = 1 : ones

head [1,2,3]	1
tail [1,2,3]	[2,3]
init [1,2,3]	[1,2]
last [1,2,3]	3
take 2 [1,2,3]	[1,2]
uncons [1,2,3]	Just (1, [2,3])

map :: (a -> b) -> [a] -> [b]

map (+1) [1, 2, 3]	[2,3,4]

filter :: (a -> Bool) -> [a] -> [a]

filter odd [1, 2, 3]	[1,3]
null []	~> True -- Checks whether list is empty (performant)
length []	~> 0 -- Checks length (need to go through the whole list)

[a,b,c] = a : (b : (c : []))

[]	nil
(:)	cons operator

stdMatch :: Show a => [a] -> String

stdMatch [] = "Matched empty list"

Haskell Cheat Sheet

by xonnex via cheatography.com/196187/cs/41216/

Lists (cont)

```
stdMatch (x:xs) = "Matched list with head " ++ show x
```

```
ml :: Show a => [a] -> String
ml [x] = "Matched list with one element" ++ show x
ml [x,y] = "Matched list with two elements"
```

sequence of elements of the same type
infinite amount of elements
immutable
"++" concat two lists

String

reverse "abc"	"cba"
['a','b','c'] == "abc"	True
"Foo" ++ " " ++ "-Bar"	"Foo Bar"

String = [Char]

Chars

```
'a', ,
' '
'\n'
```

Where Bindings

```
amountToText :: Int -> String
amountToText amount
 | amount >= high = "Many"
 | amount >= mid = "Medium"
 | otherwise = "Low"
 where
 high = 10
 mid = 5
```

Conditional Expressions

```
if a == b
 then "Eq"
 else "Not Eq"
```

Where Bindings

```
amountToText :: Int -> String
amountToText amount
 | amount >= high = "Many"
 | amount >= mid = "Medium"
 | otherwise = "Low"
 where
 high = 10
 mid = 5
```

Guards

```
abs :: (Num a, Ord a) => a -> a
abs n
 | n < 0 = -n
 | otherwise = n
```

Function Composition

```
g . f = \x -> g (f x)
```

Lambda Expressions

```
\x -> x + 1
```

\p q -> e	same as \p -> \q -> e

Let Bindings

```
cylinder :: Float -> Float -> Float
cylinder r h =
 let sideArea = 2 *pi* r * h
 topArea = pi * r ^ 2
 in 2 * topArea + sideArea
```

Double

Floating Point Number 64 bit

Case Expressions

```
case expression of
 pattern -> result
 pattern -> result

describeList :: [a] -> String
describeList xs = "The list is " ++ case xs of
 [] -> "empty."
 [x] -> "a singleton list."
 xs -> "a longer list."
```

Integer

maxBound :: Int	max int
45	literals will always default to Integer

long in Java 2^64

Record Types

```
data Person = MkPerson { name :: String, age :: Int } deriving (Show)
me = MkPerson "XonneX" 99
```

name me	"XonneX"
age me	99

Type Synonyms

```
type Coord = (Int, Int)
xCoord :: Coord -> Int
xCoord (x, y) = x
time :: (Int, Int)
time = (23, 59)
```

Type Synonyms (cont)

xCoord time	compiles

The keyword type can be used to introduce a new name (a synonym) for an existing type. This does not create a new type, only a new name!

Tuples

```
(False, 8, "Hallo") :: (Bool, Int, String)
((True, 8), (12, "Hallo")) :: ((True, 8), (12, "Hallo"))
```

Enumerations

```
data Color = Red | Yellow | Green
```

show Green	Would fail as no toString method is not implemented

```
data Color = Red | Yellow | Green deriving (Show)
```

show Green	Displays "Green" as toString method is implemented

Operators

```
(|+|) :: Int -> Int -> Int
a |+| b = abs a + abs b
```

1 |+| (-2)	3