



# Haskell-ucs-1 - Zusammenfassung Programmierung und Modellierung

Programmierung und Modellierung (Ludwig-Maximilians-Universität München)

# 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)    ≡ f g . h $ x
```

## Values and Types

has type	<i>expr</i>	:: <i>type</i>
boolean	True    False	:: Bool
character	'a'	:: Char
fixed-precision integer	1	:: Int
integer (arbitrary sz.)	31337	:: Integer
	31337 <sup>10</sup>	:: 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 -> b
maybe (just something or nothing)	Just 10 Nothing	:: Maybe Int :: Maybe a

## Values and Typeclasses

given context, has type	<i>expr</i>	:: <i>constraint</i> => <i>type</i>
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 <i>MyType</i> = <i>Type</i> type PairList a b = [(a,b)] type String = [Char] -- from Prelude
data (single constructor)	data <i>MyData</i> = <i>MyData Type Type</i> deriving ( <i>Class, Class</i> )
data (multi constructor)	data <i>MyData</i> = <i>Simple Type</i>   <i>Duple Type Type</i>   <i>Nople</i>
data (labelled fields)	data <i>MDt</i> = <i>MDt { fieldA</i>   <i>fieldB :: TyAB</i>   <i>fieldC :: TyC }</i>
newtype	newtype <i>MyType</i> = <i>MyType Type</i> deriving ( <i>Class, Class</i> )
typeclass	class <i>MyClass</i> a where foo :: a -> a -> b goo :: a -> a
typeclass instance	instance <i>MyClass MyType</i> 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.  
Operator listings aligned left, right, and center indicate left-, right-, and non-associativity.

	non associative	infix 0-9 'op'
Defining fixity:	left associative	infixl 0-9 + -
	right associative	infixr 0-9 - ! -
	default (when none given)	infixl 9

## Functions ≡ Infix operators

```
f a b      ≡ a 'f' b
a + b      ≡ (+) a b
(a +) b    ≡ ((+) a) b
... ..    ≡ ... ..
```

## Expressions / Clauses

<b>if expression</b>	≈	<b>guarded equations</b>
if <i>boolExpr</i> then <i>exprA</i> else <i>exprB</i>		<i>foo ...</i>   <i>boolExpr</i> = <i>exprA</i>   otherwise = <i>exprB</i>
<b>nested if expression</b>	≈	<b>guarded equations</b>
if <i>boolExpr1</i> then <i>exprA</i> else if <i>boolExpr2</i> then <i>exprB</i> else <i>exprC</i>		<i>foo ...</i>   <i>boolExpr1</i> = <i>exprA</i>   <i>boolExpr2</i> = <i>exprB</i>   otherwise = <i>exprC</i>
<b>case expression</b>	≈	<b>function pattern matching</b>
case <i>x</i> of <i>pat1</i> -> <i>exA</i>   <i>pat2</i> -> <i>exB</i>   - -> <i>exC</i>		<i>foo pat1</i> = <i>exA</i> <i>foo pat2</i> = <i>exB</i> <i>foo _</i> = <i>exC</i>
<b>2-variable case expression</b>	≈	<b>function pattern matching</b>
case ( <i>x,y</i> ) of ( <i>pat1,patA</i> ) -> <i>exprA</i> ( <i>pat2,patB</i> ) -> <i>exprB</i> - -> <i>exprC</i>		<i>foo pat1 patA</i> = <i>exprA</i> <i>foo pat2 patB</i> = <i>exprB</i> <i>foo _ _</i> = <i>exprC</i>
<b>let expression</b>	≈	<b>where clause</b>
let <i>nameA</i> = <i>exprA</i>   <i>nameB</i> = <i>exprB</i> in <i>mainExpression</i>		<i>foo ...</i> = <i>mainExpression</i> where <i>nameA</i> = <i>exprA</i>   <i>nameB</i> = <i>exprB</i>
<b>do notation</b>	≈	<b>desugared do notation</b>
do <i>patA</i> <- <i>action1</i>   <i>action2</i>   <i>patB</i> <- <i>action3</i>   <i>action4</i>		<i>action1</i> >>= \ <i>patA</i> ->   <i>action2</i> >>   <i>action3</i> >>= \ <i>patB</i> ->   <i>action4</i>

## Pattern Matching (fn. declaration, lambda, case, let, where)

<b>fixed</b>	number 3	3	character 'a'	'a'
	ignore value	_	empty string	""
<b>list</b>	empty	[]		
	head x and tail xs	(x:xs)		
	tail xs (ignore head)	(_:xs)		
	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)		
<b>tuple</b>	pair values a and b	(a,b)		
	ignore second element of tuple	(a,_)		
	triple values a, b and c	(a,b,c)		
<b>mixed</b>	first tuple on list	((a,b):xs)		
<b>maybe</b>	just constructor	Just a		
	nothing constructor	Nothing		
<b>custom</b>	user-defined type	MyData a b c		
	ignore second field	MyData a _ c		
<b>as-pattern</b>	tuple s and its values a and b	s@(a,b)		
	list a, its head x and tail xs	a@(x:xs)		

## Prelude functions

(A few types have been simplified to their list instances, e.g.: foldr)

### Misc

```
id      :: a -> a           id x == x    -- identity
const   :: a -> b -> a      (const x) y == x
undefined :: a             undefined == ⊥ (lifts error)
error   :: [Char] -> 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
```

### Lists

```
null :: [a] -> Bool          null [] == True  -- []?
length :: [a] -> Int         length [x,y,z] == 3
elem :: a -> [a] -> Bool      y `elem` [x,y] == True  -- ∈?
head :: [a] -> a             head [x,y,z,w] == x
last :: [a] -> a             last [x,y,z,w] == w
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]
takeWhile, dropWhile :: (a -> Bool) -> [a] -> [a]
          takeWhile (/= z) [x,y,z,w] == [x,y]
zip :: [a] -> [b] -> [(a,b)]
          zip [x,y,z] [a,b] == [(x,a),(y,b)]
```

### Infinite Lists

```
repeat :: a -> [a]          repeat x == [x,x,x,x,x,...]
cycle   :: [a] -> [a]        cycle xs == xs++xs++xs++...
          cycle [x,y] == [x,y,x,y,x,y,...]
iterate :: (a -> a) -> a -> [a]
          iterate f x == [x,f x,f (f x),...]
```

### Higher-order / Functors

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

### 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
maximum :: Ord a => [a] -> a    maximum [9,0,5] == 9
minimum :: Ord a => [a] -> a    minimum [9,0,5] == 0
concat :: [[a]] -> [a]        concat [xs,ys,zs] == xs++ys++zs
```

## 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
```

## Numeric

```
abs      :: Num a => a -> a     abs (-9) == 9
even, odd :: Integral a => a -> Bool even 10 == True
gcd, lcm :: Integral a => a -> a gcd 6 8 == 2
recip    :: Fractional a => a -> a recip x == 1/x
pi       :: Floating a => a     pi == 3.14...
sqrt, log :: Floating a => a -> a sqrt x == x**0.5
exp, sin, cos, tan, asin, acos :: 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
(qualified)      import Some.Module
(subset)         import qualified Some.Module as SM
(hiding)         import Some.Module hiding (foo,goos)
(typeclass instances) import Some.Module ()

declaring
module Module.Name
( foo, goos )
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 f x y | traceShow (x,y) False = undefined
call values f 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/out 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
```

## 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
[x+y | x <- [a,b], y <- [i,j]] == [a+i, a+j, b+i, b+j]
[x | boolE] == if boolE then [x] else []
```

## 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
Installed GHC packages $ ghc-pkg list [pkg_name]
```

```
Activating some pragma {-# LANGUAGE Pragma #-}
Same, via GHC call $ ghc -XSomePragma ...
```

```
install package pkg $ cabal install pkg
update package list $ cabal update
list packages matching pat $ cabal list pat
information about package $ cabal info pkg
help on commands $ cabal help [command]
run executable/test/bench $ cabal run/test/bench [name]
initialize sandbox $ cabal sandbox init
add custom sandbox source $ cabal sandbox add-source dir
```