Haskell Cheatsheet

A concise reference for Haskell syntax, data types, functions, and common operations. This cheat sheet provides a quick guide to help you write and understand Haskell code efficiently.

Basic Syntax and Types

Syntax Fundamentals

Function Definition	`functionName :: Type1 -> Type2 -> ReturnType`
	`functionName arg1 arg2 = expression`
	
	Example:
	`add :: Int -> Int -> Int`
	`add x y = x + y`

Comments	`-- Single-line comment` `{- Multi-line comment -}`
Layout	Haskell uses indentation to define blocks. Consistent indentation is crucial.

Let Bindings	`let`
	`  variable = expression`
	`in`
	`  resultUsingVariable`

Where Bindings	`functionName arg1 arg2 = result`
	`  where`
	`  variable = expression`

Case Expressions	`case expression of`
	`  pattern1 -> result1`
	`  pattern2 -> result2`
	`  _ -> defaultResult`

Basic Data Types

Int	Integer values with a fixed range. Example: `5`, `-10`
Integer	Integer values with arbitrary precision. Example: `12345678901234567890`
Float/Double	Floating-point numbers. Example: `3.14`, `2.718`
Bool	Boolean values: `True` or `False`
Char	Single Unicode characters. Example: `'a'`, `'\n'`
String	List of characters. Example: `"Hello, World!"`

Type Signatures

Explicit type signatures are recommended for clarity.

```
variable :: Type
functionName :: Type1 -> Type2
```

Example:

```
x :: Int
x = 5

add :: Int -> Int -> Int
add x y = x + y
```

Functions and Control Flow

Function Application

Basic Application	Functions are applied by simply placing arguments after the function name.
	`functionName arg1 arg2`
	
	Example:
	`add 3 5  -- Result: 8`

Parentheses	Parentheses are used to control precedence. `functionName (arg1 + arg2)`
Composition	Function composition using `.` `(f . g) x  -- Equivalent to f (g x)`

Control Flow

If-Then-Else

```
if condition then expression1
else expression2
```

Example:

```
if x > 0 then "Positive" else
"Non-positive"
```

Guards

Guards provide an alternative to if-then-else for defining functions.

```
functionName arg1
  | condition1  = expression1
  | condition2  = expression2
  | otherwise   =
defaultExpression
```

Example:

```
absVal x
  | x >= 0    = x
  | otherwise = -x
```

Case Expressions

Pattern matching with `case`.

```
case expression of
  pattern1 -> result1
  pattern2 -> result2
  _        -> defaultResult
```

Lambda Expressions

Anonymous functions defined using `\`.

```
\arg1 arg2 -> expression
```

Example:

```
(\x y -> x + y) 3 5  -- Result: 8
```

Data Structures

Lists

List Syntax

```
[element1, element2,
element3]
```

Example:

```
[1, 2, 3, 4, 5]
```

Cons Operator

The `(:)` operator adds an element to the beginning of a list.

```
head : tail
```

Example:

```
1 : [2, 3]  -- Result: [1,
2, 3]
```

List Comprehension

```
[expression | variable <-
list, condition]
```

Example:

```
[x * 2 | x <- [1..5], x
`mod` 2 == 0]  -- Result:
[4, 8]
```

Common Functions

`head`, `tail`, `length`, `map`, `filter`, `foldl`, `foldr`

Tuples

Tuple Syntax

```
(element1, element2,
element3)
```

Example:

```
(1, "hello", True)
```

Accessing Elements

`fst` (for 2-tuples), `snd` (for 2-tuples)
For larger tuples, pattern matching is typically used.

Data Type Declarations

```
data MyType = Constructor1 Type1 |
Constructor2 Type2 Type3
```

Example:

```
data Color = Red | Green | Blue
data Shape = Circle Float | Rectangle Float
Float
```

Typeclasses and Monads

Typeclasses

Defining a Typeclass	`class MyTypeclass a where` `  myFunction :: a -> ReturnType`
Instances	`instance MyTypeclass MyType where` `  myFunction arg = expression`
Common Typeclasses	`Eq`, `Ord`, `Show`, `Read`, `Num`, `Functor`, `Applicative`, `Monad`

Monads

Monad Typeclass	`class Monad m where`
	`  return :: a -> m a`
	`  (>>=)  :: m a -> (a -> m b) -> m b`

Do Notation	Syntactic sugar for working with Monads.
	`do`
	`  x <- action1`
	`  y <- action2 x`
	`  return (x + y)`

Common Monads	`Maybe`, `IO`, `List`, `Either`

IO Monad

Used for performing input/output operations.

```
main :: IO ()
main = do
  line <- getLine
  putStrLn ("You entered: " ++ line)
```