Haskell Cheatsheet

Tooling & Project Basics GHC / REPL

- Compile: `ghc Main.hs`
- Run: `runghc Main.hs`
- REPL: `ghci` (reload: `:r`, quit: `:q`)
- Type of expr: `:t expr` Info: `:i Name`

Cabal / Stack

- Cabal init/build: `cabal init`, `cabal build`, `cabal run`
- Stack: `stack new`, `stack build`, `stack run`, `stack ghci`

Basics: Values, Types, Bindings Hello `main ::`

```
IO () main = putStrLn "Hello"
```

Bindings

- Immutable by default: `x = 42`
- Local: `let x = 1 in x + 2`
- Where: `f x = y + 1 where y = x * 2`

Type annotations

- `x :: Int`
- Functions: `add :: Int -> Int -> Int`
- Polymorphic: `id :: a -> a`

Common types

- `Int`, `Integer`, `Float`, `Double`
- `Bool`, `Char`, `String` (`String = [Char]`)
- Lists: `[a]` Tuples: `(a,b)`
- `Maybe a`, `Either e a`

Operators & precedence

- Function application has high precedence: `f x y`
- ()*lowersprecedence* :
- `Compose: (.): (f . g) x == f (g x)`

Lists & Strings List literals

- `xs = [1,2,3]`
- Cons: `1 : [2,3]` (`:` prepends)
- Concat: `xs ++ ys`
- Ranges: `[1..10]`, `[0,2..10]`

Common list funcs (Prelude)

- `head`, `tail`, `init`, `last` (partial; can crash)
- `length`, `null`, `reverse`, `take`, `drop`
- `map`, `filter`, `foldr`, `foldl'`
- `zip`, `unzip`, `any`, `all`, `elem`

List comprehension `[x*x | x <- [1..10], even x]`

Strings

- String is `[Char]`: list ops work
- Better perf: `Text` (package `text`), `ByteString` (`bytestring`)

Functions & Pattern Matching Definitions

- `add x y = x + y`
- Guards:

```
abs' x | x >= 0 = x | otherwise = -x
```

Pattern matching

- On args:

```
len [] = 0 len (_:xs) = 1 + len xs
```

Case `case xs of [] -> 0; (y:) -> y`

Lambda

- `\x -> x + 1`
- Sections: `(+1)`, `(1+)`

Let / where

- `let a = 1; b = 2 in a + b`
- `f x = g x where g y = y + 1`

Algebraic Data Types (ADT) Data `data Color =`

`Red | Green | Blue`

Parametric types `data Box a = Box a`

Record syntax `data User = User { userId :: Int, name :: String }`

Type synonyms vs newtypes

- Alias: `type UserId = Int` (no runtime distinction)
- Newtype: `newtype UserId = UserId Int` (distinct type, zero-cost)

Deriving

- `deriving (Eq, Ord, Show, Read)`

Typeclasses & Instances Typeclass idea

- "Interface" of functions with laws (informal but important)
- Examples: Eq, Ord, Show, Functor, Monad

Define a typeclass `class Pretty a where pretty :: a -> String`

Instance `instance Pretty Color where pretty Red = "red"`

Common typeclasses

- Semigroup (`<>`), Monoid (`mempty`)
- Functor (`fmap`), Applicative (`<*>`), Monad (`>>=`)
- `Foldable`, `Traversable`

Maybe, Either, Error-ish Patterns Maybe

- data Maybe a = Nothing | Just a
- Safe lookup: lookup k xs :: Maybe v

Either

- data Either e a = Left e | Right a
- Great for error messages: Either String a

Working with them

- maybe def f mx
- either fe fa ex
- fmap maps inside Maybe/Either

Do-Notation & Monads IO basics

- getLine :: IO String
- putStrLn :: String -> IO ()

Do-notation (sequence) main = do s <- getLine putStrLn ("You: " ++ s)

Maybe in do

- do works with any Monad: Maybe, Either e, lists, etc.

Bind vs fmap

- fmap :: (a->b) -> m a -> m b
- (>>=) :: m a -> (a -> m b) -> m b

Higher-Order Patterns Pointfree-ish

- sumSquares = sum . map $(^2)$
- Use sparingly; readability matters

Folds

- foldr good for lists / laziness
- foldl' (from Data.List) strict left fold (avoid space leaks)

Strictness (practical)

- Haskell is lazy; can build thunks (memory)
- Use foldl' / strict fields / seq when needed

Modules & Imports **Module header** module My.Mod

(foo, Bar(..)) where

Imports

- import Data.List (sort, nub)
- Qualified: import qualified Data.Map as M
- Hide: import Prelude hiding (lookup)

Common Libraries (Ecosystem)

- Text: text (Data.Text)
- Bytes: bytestring
- Maps/Sets: containers (Data.Map, Data.Set)
- JSON: aeson
- Parsing: megaparsec / attoparsec
- Testing: hspec, tasty, property tests: QuickCheck

Mini Reference

- :: "has type" -> function type => constraints
- () unit [] empty list (:) cons (++) concat
- <> isfmap; <*>$applicativeapply$; >>=$monadicbind$
- Prefer total functions; avoid head/!! unless you prove safety