Haskell Pure functional language

START HERE

`ghci`
Interactive REPL

`:r`
Reload changed modules

`ghc -O2 -o <out> Main.hs`
Optimised build

`cabal run`
Build and run

`runghc <file.hs>`
Run a script

`cabal repl`
GHCi for the project

`:t <expr>`
Show a type in GHCi

`ghcup tui`
Manage toolchains

ghcup installs ghc, cabal, stack, hls
cabal or stack resolve packages, build, test
ghc compiles modules, ghci is its REPL

The toolchain: ghcup installs the compilers, a build tool drives them

CONCEPTS

Everything is an expression and functions are pure. Side effects live in IO, which main returns.

Use cabal or stack for projects and plain ghc for single files. Stack uses curated snapshots, cabal uses a solver.

A module named Main must export main :: IO (). Layout (indentation) delimits blocks.

GHC

`ghc <file.hs>`
Compile a program
Output goes next to the source.

`ghc -o <out> <file.hs>`
Name the executable

`ghc -O2 <file.hs>`
Optimise heavily
-O1 is the usual -O; the default is -O0.

`ghc -Wall <file.hs>`
Enable common warnings

`ghc -e '<expr>'`
Evaluate an expression and exit

`ghc -i<dir> Main.hs`
Add a module search directory

`ghc -threaded -rtsopts Main.hs`
Threaded runtime with RTS flags

`runghc <file.hs>`
Run a source file directly

GHCI

`ghci`
Start the REPL

`:l <file>`
Load a module

`:r`
Reload the loaded modules

`:t <expr>`
Show the type of an expression

`:i <name>`
Show info on a name or class

`:k <type>`
Show the kind of a type

`:q`
Quit GHCi
Ctrl+D also works.

GHCI (cont.)

`:doc <name>`
Show documentation for a name

`:set -X<Ext>`
Enable an extension

`let x = 5`
Bind a name at the prompt

CABAL

`cabal init`
Create a package

`cabal build`
Build the targets

`cabal run <exe>`
Build and run an executable
Program arguments go after the target.

`cabal test`
Build and run test suites

`cabal repl`
GHCi with the package loaded

`cabal update`
Refresh the package index

`cabal install <pkg>`
Install an executable

`cabal clean`
Remove dist-newstyle

STACK AND GHCUP

`stack new <name>`
Create a project

`stack build`
Build the project
Fetches the snapshot's GHC if needed.

`stack run`
Build and run

`stack test`
Build and run tests

`stack ghci`
Open GHCi for the project

`stack install <pkg>`
Install an executable

`ghcup list`
List available tools and versions

`ghcup install ghc <ver>`
Install a GHC version

`ghcup set ghc <ver>`
Select the active GHC

TYPES AND CLASSES

`f :: Int → Int`
Type signature

TYPES AND CLASSES (cont.)

`data T = A | B Int`
Sum type with a constructor field

`data P = P { x :: Int }`
Record with field accessors

`newtype N = N Int`
Zero-cost wrapper type

`class C a where`
Declare a typeclass

`instance C T where`
Implement a class for a type

`deriving (Show, Eq, Ord)`
Auto-generate instances

`f :: (Show a) ⇒ a → String`
Constrain a type variable

`Maybe a`
Value that may be missing
Just x or Nothing.

`Either e a`
Success or failure
Left is the error by convention.

`[a]`
List of a

`(a, b)`
Tuple

`IO a`
Action that yields an a

SYNTAX

`f 0 = 1; f n = n * f (n-1)`
Pattern matching clauses

`case m of Just x → x`
Pattern match an expression

`f x | x > 0 = 1 | otherwise = 0`
Guards

`\x → x + 1`
Lambda

`let y = 2 in y * 2`
Local binding

`f = g where g = 1`
Bindings scoped to a definition

`do { x ← act; return x }`
Do notation for monads

`[x*2 | x ← xs, x > 1]`
List comprehension

`f . g`
Compose functions

`f $ g x`
Apply with low precedence

SYNTAX (cont.)

`import qualified Data.Map as M`
Qualified import

COMMON FUNCTIONS

`map f xs`
Apply f to each element

`filter p xs`
Keep elements matching p

`foldr f z xs`
Right fold

`foldl' f z xs`
Strict left fold
From Data.List.

`zip xs ys`
Pair up two lists

`fmap f m / f <$> m`
Map inside a functor

`mapM_ print xs`
Run an action per element

`either f g e`
Fold an Either

`show x`
Convert to a string

`putStrLn s`
Print a line

`getLine`
Read a line of input

RECIPES

Hello program

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

-- ghc -O2 -Wall -o hello
Hello.hs
```

Handle a Maybe

```
safeDiv :: Int → Int → Maybe
Int
safeDiv _ 0 = Nothing
safeDiv a b = Just (a `div` b)

main = print (safeDiv 4 2 >>=
safeDiv 10)
```

Typeclass instance

```
data Color = Red | Green
deriving (Eq, Show)
class Pretty a where pretty :: a
→ String
instance Pretty Color where
  pretty Red = "red"
  pretty Green = "green"
```

New cabal project

```
mkdir demo && cd demo
cabal init --non-interactive
cabal build && cabal run
cabal repl
```