Haskell Pure functional language

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

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

ghc -O2 *<file.hs>*
Optimise heavily

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

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

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

cabal test
Build and run test suites

cabal repl
GHCi with the package loaded

CABAL (cont.)

cabal update
Refresh the package index

cabal install *<pkg>*
Install an executable

STACK AND GHCUP

stack new *<name>*
Create a project

stack build
Build the project

stack run
Build and run

stack test
Build and run tests

stack ghci
Open GHCi for the project

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

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

TYPES AND CLASSES (cont.)

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

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

Either e a
Success or failure

[a]
List of 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

SYNTAX (cont.)

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

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

mapM_ print xs
Run an action per element

show x
Convert to a string

putStrLn s
Print a line