Haskell Pure functional language

CONCEPTS

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

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

`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

`: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

TYPES AND CLASSES

`f :: Int → Int`
Type signature

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

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

`Maybe a`
Value that may be missing

SYNTAX

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

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

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

`f . g`
Compose functions

COMMON FUNCTIONS

`map f xs`
Apply f to each element

`filter p xs`
Keep elements matching p