

Strings. → data structures used to contain text.

↓

usually represented as a sequence of characters.

: types → way of categorizing values

∴ → 'has the type' → encountered often in Haskell, typically when dealing with type signature.

↓

line of code that defines the types for a value, expression, or function

: type "Hello"

"Hello" :: [Char] → []: Syntactic sugar for a list in Haskell

↓

for a list of characters

← String is a type alias

↓

refers to the use of a high level name for a type, usually for convenience, that has a different name underneath.

not limited to strings.
can be used to print any type

: simple print "Hello world" is used to print string in

REPL. prints with quotation around the string.

specific to REPL → print

→ `putStrLn` : print to screen, specific to strings, and move to new line.

↓
prints the string without the quotation mark

for source file :

module Print where

← `main :: IO ()`

`main = putStrLn "hello world"`

default
action when
we build
an executable

prompt can be
set by us as well
using

:set prompt

`{r c}`
↓
string to set
as prompt

→
not a fn. but often a series of instructions
to execute, which can include applying
functions and producing side effects.

↓
Also, when building a project with Stack,
having a main executable is obligatory.

IO type → input/output → special type in Haskell

↓

when the result of the
program involves effects beyond
evaluating a function or
expression.

← printing to the
screen is an effect,
so printing the o/p

of the module must be wrapped in this IO type.

do : special syntax that allows for sequencing actions.



i.e. used to sequence the actions that constitute our program.

→ string concatenation :

'++' and concat

↓
list of list given
as argument to
concatenate

used to declare the type
world :: String
world = "world"

↓
not necessary,
good habit to have
though.

↓
for top level
declarations.

→ top level vs local definitions



not nested within anything else and their scope is throughout the module, whereas local defn. means that it has been nested within some other expression and is not visible outside the expression.

→ $(++) :: [a] \rightarrow [a] \rightarrow [a]$

a can be type numbers, or a can be characters.
a can even be a list. Hence 'a' is polymorphic



But this signature does apply the constraint that both if lists should be of same type

and output list is of the same type.

→ typeclass provide definitions of operations, or functions that can be shared across set of types.

→ list functions

⇒ `::` → cons operator → used to combine a single element to a list at the front

⇒ `head` lists → returns the first element

↓
tail is the complementary function for head

⇒ `take n list` → takes n elements off the front off the list into a new list

↓
drop: the complementary i.e. returns the part take neglects.

⇒ indexing done using `!!`

"Papuchaon" !! 0 → 'P'

"Papuchaon" !! 4 → 'c'

many of these functions although a part of Prelude are considered unsafe, since they don't cover the case where they might be handed an empty string.

↓
instead they throw out an
error message or exception

↓
Due to their unsafe nature, they are
not really used for any big or complex
project.