•) running ghc in interactive mode. Arithmetic operators, can be tried on the same line, and normal precedence rules are obeyed. Use of paranthesis is again recommended.

•) surrounding negative numbers with paranthesis.

•) True, False and usual boolean operators.

•) Here the arithmetic operators used are also functions. These functions which are called by sandwiching b/w two parameters are called infix fns. other fns. are prefix fns. In Haskell, fns are called by writing the fn. name, a space and then the parameters, separated by spaces.

↓

takes anything that has a defined successor and returns that successor. ← succ 8

9

↓

min 2 3 → to work with many parameters.

2

→ function application has the highest precedence of them all

but succ (9 * 10) → returns 91

↓

first this, then the succ fn.

•) can use `div` based calling for fns. with 2 parameters to call as infix fn.

•) bar (bar 3) → chaining function calls.

•) Defining a function, basics : done in the same way in a similar way that they are called.

↓

fn. name followed by parameters separated by spaces. followed by '=' after which we defined what the fn. does. (like an actual fn.)

↙ need to begin with lowercase letter

•) functions in Haskell don't have to be defined in a given order.

↓

•) conditionals in Haskell fns.

```
doubleSmallNumber x = if x > 100
                      then x
                      else x * 2
```

↓ can be written in one line, but this is more readable.

→ Here the else statement is mandatory to be defined

↓ in Haskell every expression & function must return something.

↓ Haskell's if statement is an expression (piece of code that returns a value). By making else mandatory, Haskell made the if statement, an expression.

→ when a function does not take any parameter, we usually say its a defn. (or a name)

(•) Lists : Homogenous data structure i.e. it stores several elements of the same type. (denoted by [])

(*) We can use let in ghci to define a name, This is equivalent to simply defining the variable in a script i.e.

let a = 1 ⟷ a = 1

→ '++' is used to concatenate lists. Requires complete parsing of the list on the left, so can be slow for really long list. But putting something at the beginning of a list using ':' operator is instantaneous.

(: allows for element to list addition)

→ Accessing an element is done using '!!'

[2, 3, 4, 5] !! 1 → 3

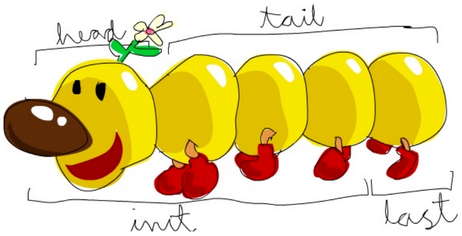
→ Lists within list can be of different lengths but can't be of different types. Lists can be compared if their content is comparable. This comparision is done in lexicographical order

→ Some basic functions on lists : a) head

b) tail (list - head)

c) last

d) init (list - last)



e) length

* f) null (checks if a list is empty)

g) reverse

h) take (takes a no. and a list, and extracts that many elements from the beginning of the list)

i) drop

J) maximum

K) minimum

l) sum

m) product

n) elem (takes an element and a list, and returns a boolean if the element is part of the list or not)

→ Ranges are the way of making lists that are arithmetic sequences of elements that can be enumerated.. To create a list of ranges → [1..20] : both inclusive

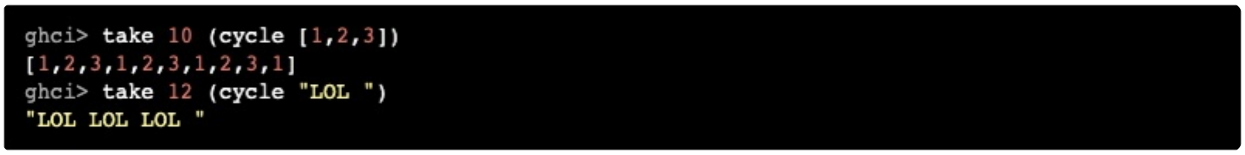
↓

step can also be defined

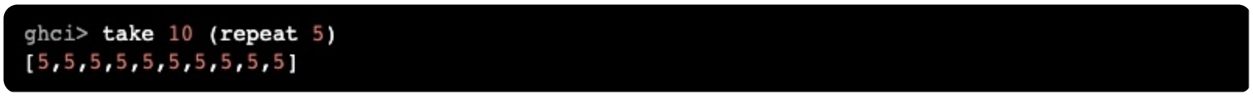
[1, 3]. . . . . 19]

Separating the first two elements with a comma. → only one step can be defined

→ cycle and repeat return '∞' lists.



```
ghci> take 10 (cycle [1,2,3])
[1,2,3,1,2,3,1,2,3,1]
ghci> take 12 (cycle "LOL ")
"LOL LOL LOL "
```



```
ghci> take 10 (repeat 5)
[5,5,5,5,5,5,5,5,5,5]
```

→ replicate can also be used to repeat an element a certain no. of times.

(•) List comprehension → analogous to set comprehensions in maths, wherein specific sets are built out of more general sets. for example

$$S = \{ \underbrace{2 \cdot x}_{\text{output function}} \mid x \in \underbrace{N}_{\text{input set}}, \overbrace{x \leq 10}^{\text{predicate}} \}$$

↓

ultimately means that S is a set that contains the double of all natural nos that satisfy the predicate.