Haskell

is a **purely functional programming language**.
In purely functional programming you don't tell the computer what to do as such but rather you tell it what stuff *is*.
You also can't set a variable to something and then set it to something else later.
So in purely functional languages, a function has no side-effects.
The only thing a function can do is calculate something and return it as a result. it actually has some very nice consequences: if a function is called twice with the same parameters, it's guaranteed to return the same result. That's called referential transparency

Haskell is **lazy**. That means that unless specifically told otherwise, Haskell won't execute functions and calculate things until it's really forced to show you a result.
it allows you to think of programs as a series of **transformations on data**.

Haskell is **statically typed**. When you compile your program, the compiler knows which piece of code is a number, which is a string and so on. That means that a lot of possible errors are caught at compile time.
Haskell uses a very good type system that has **type inference**. That means that you don't have to explicitly label every piece of code with a type because the type system can intelligently figure out a lot about it.

Haskell is **elegant and concise**. Because it uses a lot of high level concepts, Haskell programs are usually shorter than their imperative equivalents.

ghci -> run ghc's interactive
:set prompt "ghci> " -> because it can get longer when you load stuff into the session

in terminal command is -> runhaskell filename.hs

/=
not equal(!=)

div 7 2 = 3

7/2 = 3.5

Whereas **+** works only on things that are considered numbers, **==** works on any two things that can be compared.
Note: you can do **5 + 4.0**

***** is a function that takes two numbers and multiplies them. This is what we call an *infix* function

Most functions that aren't used with numbers are *prefix* functions.Functions are usually prefix so from now on we won't explicitly state that a function is of the prefix form

In Haskell, functions are called by writing the function name, a space and then the parameters, separated by spaces.

Function application (calling a function by putting a space after it and then typing out the parameters) has the highest precedence of them all.
max 6 4 + 1 is equivalent to max (6 4) + 1

If a function takes two parameters, we can also call it as an infix function by surrounding it with backticks.
` -> alt +(;,)
92 `div` 10

something like **bar (bar 3)**, it doesn't mean that **bar** is called with **bar** and **3** as parameters. It means that we first call the function **bar** with **3** as the parameter to get some number and then we call **bar** again with that number. In C, that would be something like **bar(bar(3))**.

- haskell do not do type conversion implicitly
- comparison between user defined types is not provided default (only in primitive and cartesian product among them)
- (A 6)==(A 7) is error
- if you add deriving (Show, Eq) it is not a error
- mapping values cannot be printed and Eq is not defined
- functions/ mapping are threated like value
- only difference is printing and comparison operator
- function names start with lower case

HASKELL code

function definition=
name arg1 arg2 arg3 = <expression>

```
in_range min max nm =  nm >= min && nm <= max
main = print $ in_range 1 4 2
```
the output is True
```
main = print $ in_range 1 4 7
```
the output is False

basic types=
name :: <type>

```
x :: Integer
x = 1
y :: Bool
y = True
z :: Float
z = 3.14
```

*functions have their own types=
list of types of arguments and return type

```
in_range :: Integer -> Integer -> Integer -> Bool
in_range min max nm =  nm >= min && nm <= max
```

to save the result of some expression=
let-in=
```
in_range min max x =
  let in_lower = min <= x
      in_upper = max >= x
  in
  in_lower && in_upper
```
where=
```
in_range min max x = in_lower && in_upper
  where
    in_lower = min <= x
    in_upper = max >= x
```

if=

```
in_range min max x =
    if in_lower then in_upper else False
    where
         in_lower = min <= x
          in_upper = max >= x
```

infix functions=

```
add a b = a + b
add 10 30     and    10 `add` 30      are same
```

there are no loops in Haskell, there is just recursion=

```
name <args> = … name <args’>

factor n =
        if n <= 1 then
                1
        else
                n * factor (n-1)
```

guards=

```
factor2 n
   | n <= 1 = 1
   | otherwise = n * factor2 (n-1)
```

pattern matching=

```
is_zero 0 = True
is_zero _ = False
```

accumulators (auxiliary function: yardımcı fonksiyon)=

tail recursive functions are better than non-tail recursive functions. no-tail ones can cause a stack over-flow, because in every recursive call you have to put a new stack frame, and it is limited. in tail recursive you call the function only ones.

tail recursive in c=

```
fac n:
acc=1;
while(1){
        if(n<=1)
                return acc;
```

```
        else{
                n—;
                acc*=n;
        }
}
```

<u>in Haskell=</u>

```
factor3  n = aux n 1
   where
      aux n acc
         | n <= 1 = acc
         | otherwise = aux (n-1) (n*acc)
```

<u>lists=</u>

<u>generating list=</u>

```
liste :: Int -> Int -> [Int]
liste n m
    | n > m = []
    | n == m = [m]
    | m > n = n : liste (n+2) m
main = print $ liste 4 16  will be printed as [4,6,8,10,12,14,16]
```

- with adding “import Data.List” we can use functions on lists that are already implemented.

ghci
:load der —(der.hs)
:t funcname

:set prompt "ghci> "

list comprehension haskell
[x| x <- [2,3,4,5,6,7], x<4] —— [2,3]
[x<4| x <- [2,3,4,5,6,7]] ——[True,True,False,False,False,False]

function type
```
ghci> let { fact 0 = 1 ; fact n = n * fact (n-1) }
ghci> fact 3
6
ghci> { fact 0 = 1 ; fact n = n * fact (n-1) }
ghci> fact 5
120
ghci> {sm [] = 0; sm (x:xs) = x + sm (xs)}
ghci> sm [3,4,5]
12
ghci> :t sm
sm :: Num p => [p] -> p
ghci> { examFunc f g s [] = g s ; examFunc f g s (a:b) = g ( f a ( examFunc f g s b)) }
ghci> :t examFunc
examFunc :: (t1 -> t2 -> t3) -> (t3 -> t2) -> t3 -> [t1] -> t2
ghci>
```