A glance at Haskell

fyusuf-a

December 2, 2021

School 42Paris

Learn haskell in 10 minutes

Program 1

```haskell
whoAmI :: Int -> Int -> Int
whoAmI x y = x + y
```

Program 1

```
add :: Int -> Int -> Int
add x y = x + y
```

Program 1

```
add :: Int -> Int -> Int
add x y = x + y
```

add 5 3 =

Program 1

```haskell
add :: Int -> Int -> Int
add x y = x + y
```

add 5 3 = 5 + 3 = 8

Program 1

```
add :: Int -> Int -> Int
add x y = x + y
```

add 5 3 = 5 + 3 = 8

add 10 ::

Program 1

```haskell
add :: Int -> Int -> Int
add x y = x + y
```

add 5 3 = 5 + 3 = 8

add 10 :: Int -> Int

Program 1

```
add :: Int -> Int -> Int
add x y = x + y
```

add 5 3 = 5 + 3 = 8

add 10 :: Int -> Int

(add 10) 3 =

Program 1

```
add :: Int -> Int -> Int
add x y = x + y
```

add 5 3 = 5 + 3 = 8

add 10 :: Int -> Int

(add 10) 3 = 13

Program 2 – recursivity in Haskell

```
whoAmI 0 = 1
whoAmI n = n * whoAmI (n - 1)
```

Program 2 – recursivity in Haskell

```
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

$$factorialn = n! = n \times (n-1) \times \cdots \times 1$$

Side note – precedence

factorial 2 - 2 =

factorial (2 - 2) =

Side note – precedence

factorial 2 - 2 = 2 * 1 - 2

factorial (2 - 2) =

Side note – precedence

factorial 2 - 2 = 2 * 1 - 2 = 0

factorial (2 - 2) =

Side note – precedence

factorial 2 - 2 = 2 * 1 - 2 = 0

factorial (2 - 2) = factorial 0

Side note – precedence

factorial 2 - 2 = 2 * 1 - 2 = 0

factorial (2 - 2) = factorial 0 = 1

Program 3

```
whoAmI :: [Char] -> [Char]
whoAmI [] = []
whoAmI (c:str) = c:'f':whoAmI str
```

Program 3

```
orcish :: [Char] -> [Char]
orcish [] = []
orcish (c:str) = c:'f':orcish str
```

Program 3

```
orcish :: [Char] -> [Char]
orcish [] = []
orcish (c:str) = c:'f':orcish str
```

orcish "hello" =

Program 3

```
orcish :: [Char] -> [Char]
orcish [] = []
orcish (c:str) = c:'f':orcish str
```

orcish ”hello” = ”hfeflflfof”

Program 4

```
yell :: [Char] -> [Char]
yell x = x ++ "!"
```

yell (orcish ”hello”) =

Program 4

```
yell :: [Char] -> [Char]
yell x = x ++ "!"
```

yell (orcish ”hello”) =

Program 4

```
yell :: [Char] -> [Char]
yell x = x ++ "!"
```

yell (orcish ”hello”) = ”hfeflflfof!”

Program 5

```
whoAmI :: (a -> a) -> (a -> a)
whoAmI f x = f (f (f x))
```

Program 5

```haskell
enthusiastically :: (a -> a) -> (a -> a)
enthusiastically f x = f (f (f x))

very :: (a -> a) -> (a -> a)
very = enthusiastically
```

Program 5

```
enthusiastically :: (a -> a) -> (a -> a)
enthusiastically f x = f (f (f x))

very :: (a -> a) -> (a -> a)
very = enthusiastically
```

Program 5

```
enthusiastically :: (a -> a) -> (a -> a)
enthusiastically f x = f (f (f x))

very :: (a -> a) -> (a -> a)
very = enthusiastically
```

Program 5

```
enthusiastically :: (a -> a) -> (a -> a)
enthusiastically f x = f (f (f x))

very :: (a -> a) -> (a -> a)
very = enthusiastically
```

enthusiastically yell ”hello” =

Program 5

```
enthusiastically :: (a -> a) -> (a -> a)
enthusiastically f x = f (f (f x))

very :: (a -> a) -> (a -> a)
very = enthusiastically
```

enthusiastically yell ”hello” = ”hello!!!”

Program 5

```
enthusiastically :: (a -> a) -> (a -> a)
enthusiastically f x = f (f (f x))

very :: (a -> a) -> (a -> a)
very = enthusiastically
```

enthusiastically yell ”hello” = ”hello!!!”

yell (orcish ”hello”) =

Program 5

```
enthusiastically :: (a -> a) -> (a -> a)
enthusiastically f x = f (f (f x))

very :: (a -> a) -> (a -> a)
very = enthusiastically
```

enthusiastically yell "hello" = "hello!!!"

yell (orcish "hello") = "hfeflflfof!"

True Orcish is subtle

very orcish ”hello” =

True Orcish is subtle

very orcish ”hello” = ”hfffffffefffffffflffffffflfffffffofffffff”

True Orcish is subtle

very orcish ”hello” = ”hfffffffeffffffflffffffflfffffffofffffff”

orcish (enthusiastically yell ”hello”) =

True Orcish is subtle

very orcish ”hello” = ”hfffffffefffffffflffffffflffffffforfffffff”

orcish (enthusiastically yell ”hello”) = ”hfeflflfof!f!f!f”

True Orcish is hardcore

True Orcish is hardcore

True Orcish is hardcore

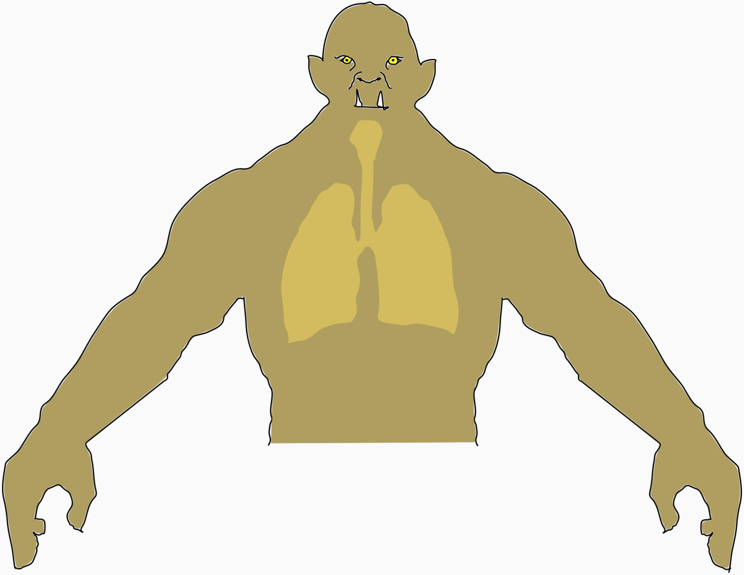
True Orcish is hardcore

very very orcish ”hello” =

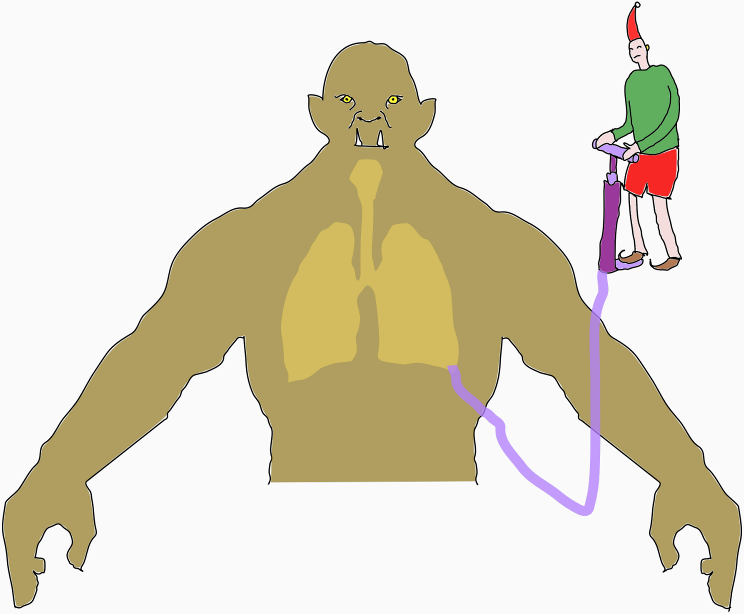
True Orcish is hardcore

very very orcish ”hello” =
”hffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff

Cross section of an orc



Sociology of the orcish society



Haskell is strongly-typed

Why compile-time checks are important

```
import sys;

print("Hello")
if len(sys.argv) == 2:
    undefined_function()
```

Why compile-time checks are important

```
import System.Environment
import Control.Monad

main = do
  putStrLn "Hello"
  args <- getArgs
  when (length args == 2)
    undefined_function
```

Why compile-time checks are important

```
hello.hs:8:5: error:
    Variable not in scope: undefined_function :: IO ()
  |
8 |     undefined_function
  |     ^^^^^^^^^^^^^^^^^^
```

The power of type inference

```haskell
e :: c -> d
e = undefined

f :: a -> [a]
f = undefined

h :: [a] -> (a -> b) -> [b]
h = undefined

-- headache :: ???
headache = h (f e)
```

The power of type inference

```haskell
e :: c -> d
e = undefined

f :: a -> [a]
f = undefined

h :: [a] -> (a -> b) -> [b]
h = undefined

-- headache :: ???
headache = h (f e)
```

```
*Main> :t headache
headache :: ((c -> d) -> b) -> [b]
*Main>
```

Purity in Haskell

A benign C code

```
int justAdd(int x, int y) {
        launchNuclearStrike();
        return (x + y);
}


int main() {
  int x = 15;
  int y = 24;
  printf("%d+%d=%d\n", x, y, justAdd(x, y));
}
```

A redflag in Haskell

```
launchNuclearStrike :: IO ()
launchNuclearStrike = undefined

justAdd :: Int -> Int -> IO Int
justAdd x y = do
  launchNuclearStrike
  return (x + y)

main :: IO ()
main = do
  let x = 15
  let y = 24
  result <- justAdd x y
  putStrLn (show x ++ "+" ++
            show y ++ "=" ++ show result)
```

The Ultimate Question of Life, the Universe and Everything

Haskell's answer : "Separate pure and impure code and minimize impure code."

But why?

Testing becomes easier, and...

But why?

Testing becomes easier, and...

one can have lazy evaluation!

Haskell is lazy

Yet another riddle

```
whoAmI :: Int -> [a] -> [a]
whoAmI n [] = []
whoAmI 0 l = []
whoAmI n (x:xs) = x:whoAmI (n - 1) xs
```

Yet another riddle

```
take :: Int -> [a] -> [a]
take n [] = []
take 0 l = []
take n (x:xs) = x:take (n - 1) xs
```

Yet another riddle

```
take :: Int -> [a] -> [a]
take n [] = []
take 0 l = []
take n (x:xs) = x:take (n - 1) xs
```

take 0 [4, 5, 1] =

Yet another riddle

```
take :: Int -> [a] -> [a]
take n [] = []
take 0 l = []
take n (x:xs) = x:take (n - 1) xs
```

take 0 [4, 5, 1] = []

Yet another riddle

```
take :: Int -> [a] -> [a]
take n [] = []
take 0 l = []
take n (x:xs) = x:take (n - 1) xs
```

take 0 [4, 5, 1] = []

take 3 [] =

Yet another riddle

```
take :: Int -> [a] -> [a]
take n [] = []
take 0 l = []
take n (x:xs) = x:take (n - 1) xs
```

take 0 [4, 5, 1] = []

take 3 [] = []

Yet another riddle

```
take :: Int -> [a] -> [a]
take n [] = []
take 0 l = []
take n (x:xs) = x:take (n - 1) xs
```

take 0 [4, 5, 1] = []

take 3 [] = []

take 1 [2, 4] =

Yet another riddle

```
take :: Int -> [a] -> [a]
take n [] = []
take 0 l = []
take n (x:xs) = x:take (n - 1) xs
```

take 0 [4, 5, 1] = []

take 3 [] = []

take 1 [2, 4] = 2:take 0 [4] = 2:[] = [2]

Every breath you take

```
take :: Int -> [a] -> [a]
take n [] = []
take 0 l = []
take n (x:xs) = x:take (n - 1) xs
```

take 2 [1, 4, 5] =

Every breath you take

```
take :: Int -> [a] -> [a]
take n [] = []
take 0 l = []
take n (x:xs) = x:take (n - 1) xs
```

take 2 [1, 4, 5] = 1:take 1 [4, 5]

Every breath you take

```
take :: Int -> [a] -> [a]
take n [] = []
take 0 l = []
take n (x:xs) = x:take (n - 1) xs
```

take 2 [1, 4, 5] = 1:take 1 [4, 5]

= 1:[4] = [1, 4]

The list of every natural numbers

```haskell
natConstructor :: Int -> [Int]
natConstructor x = x:natConstructor (x+1)

nat :: [Int]
nat = natConstructor 0
```

The list of every natural numbers

```
natConstructor :: Int -> [Int]
natConstructor x = x:natConstructor (x+1)

nat :: [Int]
nat = natConstructor 0
```

take 1 nat =

The list of every natural numbers

```
natConstructor :: Int -> [Int]
natConstructor x = x:natConstructor (x+1)

nat :: [Int]
nat = natConstructor 0
```

take 1 nat = [0]

The list of every natural numbers

```
natConstructor :: Int -> [Int]
natConstructor x = x:natConstructor (x+1)

nat :: [Int]
nat = natConstructor 0
```

take 1 nat = [0]

take 5 nat =

The list of every natural numbers

```
natConstructor :: Int -> [Int]
natConstructor x = x:natConstructor (x+1)

nat :: [Int]
nat = natConstructor 0
```

take 1 nat = [0]

take 5 nat = [0, 1, 2, 3, 4]

∞ ???

```
natConstructor :: Int -> [Int]
natConstructor x = x:natConstructor (x+1)

nat :: [Int]
nat = natConstructor 0
```

```
natConstructor :: Int -> [Int]
natConstructor x = x:natConstructor (x+1)

nat :: [Int]
nat = natConstructor 0
```

take 2 nat =

∞ ???

```haskell
natConstructor :: Int -> [Int]
natConstructor x = x:natConstructor (x+1)

nat :: [Int]
nat = natConstructor 0
```

take 2 nat = take 2 (natConstructor 0)

∞ ???

```
natConstructor :: Int -> [Int]
natConstructor x = x:natConstructor (x+1)

nat :: [Int]
nat = natConstructor 0
```

take 2 nat = take 2 (natConstructor 0)

= take 2 (0:natConstructor 1)

∞ ???

```
natConstructor :: Int -> [Int]
natConstructor x = x:natConstructor (x+1)

nat :: [Int]
nat = natConstructor 0
```

take 2 nat = take 2 (natConstructor 0)

= take 2 (0:natConstructor 1)

= 0:take 1 (natConstructor 1) =

∞ ???

```
natConstructor :: Int -> [Int]
natConstructor x = x:natConstructor (x+1)

nat :: [Int]
nat = natConstructor 0
```

take 2 nat = take 2 (natConstructor 0)

= take 2 (0:natConstructor 1)

= 0:take 1 (natConstructor 1) = 0:1:take 0 (natConstructor 2)

∞ ???

```
natConstructor :: Int -> [Int]
natConstructor x = x:natConstructor (x+1)

nat :: [Int]
nat = natConstructor 0
```

take 2 nat = take 2 (natConstructor 0)

= take 2 (0:natConstructor 1)

= 0:take 1 (natConstructor 1) = 0:1:take 0 (natConstructor 2)

= 0:1:[] =

```
natConstructor :: Int -> [Int]
natConstructor x = x:natConstructor (x+1)

nat :: [Int]
nat = natConstructor 0
```

take 2 nat = take 2 (natConstructor 0)

= take 2 (0:natConstructor 1)

= 0:take 1 (natConstructor 1) = 0:1:take 0 (natConstructor 2)

= 0:1:[] = [0, 1]

Haskell is concise

Haskell expressivity (1/2)

```
sort [] = []
sort (p:xs) = sort lesser ++ p:sort greater
    where lesser = [x | x <- xs, x < p]
          greater = [x | x <- xs, x >= p]
```

Haskell expressivity (2/2)

```
-- Sort all the characters in all strings,
-- across word boundaries!
>>> ("one", "two", "three") & partsOf (each . traversed)
      %~ sort
("eee","hno","orttw")

-- Flip the 2nd bit of each number to a 0
>>> [1, 2, 3, 4] & traversed . bitAt 1 %~ not
[3,0,1,6]
```

Real life Haskell

A scraper of Hacker News' API

Java : 293 lines and 7547 characters

Haskell : 155 lines 5541 characters

A scraper of Hacker News' API

Java : 293 lines and 7547 characters

$\simeq$ 26 characters per line

Haskell : 155 lines 5541 characters

$\simeq$ 35 characters per line

A scraper of Hacker News' API

Java : 293 lines and 7547 characters

$\simeq$ 26 characters per line

Haskell : 155 lines 5541 characters

$\simeq$ 35 characters per line

```
$ ./benchmark.sh 10
Mean of fyusuf-a for contest (10 tries): 3.24s
Mean of ******** for contest (10 tries): 5.00s
```

Haskell is only for mathematicians ?

Haskell is only for mathematicians ?

- Facebook: anti-spam filter and lex-pass, a tool for manipulating a PHP database;

Haskell is only for mathematicians ?

- Facebook: anti-spam filter and lex-pass, a tool for manipulating a PHP database;
- Google: Ganeti, a tool for managing clusters of virtual servers;

Haskell is only for mathematicians ?

- Facebook: anti-spam filter and lex-pass, a tool for manipulating a PHP database;
- Google: Ganeti, a tool for managing clusters of virtual servers;
- Microsoft: Bond, their production data serialization framework;

Haskell is only for mathematicians ?

- Facebook: anti-spam filter and lex-pass, a tool for manipulating a PHP database;
- Google: Ganeti, a tool for managing clusters of virtual servers;
- Microsoft: Bond, their production data serialization framework;
- countless others...

Use cases

Haskell is a general-purpose language and has great libraries for almost anything...

Use cases

Haskell is a general-purpose language and has great libraries for almost anything...

... but from my meager experience, it seems more difficult to do numeric calculations and mobile development.

Thanks

- Chris Penner, for the examples of expressivity using lenses;
- FrungyKing, for his Youtube video and his jokes on Swedish;
- Junior 42Paris for the organization.