FUNCTIONAL PROGRAMMING

Haskell

FUNCTIONAL PROGRAMMING

- Opinions differ, and it is difficult to give a precise definition, but generally speaking:
  - Functional programming is <u>style</u> of programming in which the basic method of computation is the application of functions to arguments;
  - A functional language is one that <u>supports</u> and <u>encourages</u> the functional style.

PROCEDURAL VS OBJECT-ORIENTED VS FUNCTIONAL

- **Procedural Programming** is a kind of programming paradigm where everything will be in the form of instruction. This is could be **anything like loop statements, if-else clause, functions and even sub-routines.** These instructions will tell what the computer should do. Procedural Programming revolves around variables and instructions.
  Example: C, C++, PHP, Python
- **Object Oriented Programming is nothing but a structured procedural programming** paradigm. Everything problem is modeled into an object. Object binds together **data and behavior.** Through behavior it is possible to perform actions on data. Because of Object Oriented Programming, it is possible to structure everything logically. Object Oriented Programming revolves around objects.
  Example: C++, Java, PHP, C#, Ruby, Python
- Functional Programming is a programming paradigm in which the **problem is treated as a sequence of functions.** Data flows freely among these functions. In this paradigm, it is possible to pass a function as an argument to another function and a function can return another function. Functional Programming revolves around functions.
  Example: JavaScript, C++, Haskell, Ruby

Summing integers 1 to 10 in **C/C++/Java**

```
total = 0;
for (i = 1; i ≤ 10; ++i)
   total = total+i;
```

Computation method is **variable assignment**

Summing integers 1 to 10 in **Haskell**

```
sum [1..10]
```

Computation method is **function application**

WHY FUNCTIONAL PROGRAMMING?

- Again, there are many possible answers to this question, but generally speaking:

  - The abstract nature of functional programming leads to considerably <u>simpler</u> programs;

  - It also supports a number of powerful new ways to <u>structure</u> and <u>reason about</u> programs.

START

- Download

https://www.haskell.org/downloads/

```
main = do
putStrLn "Hello World"
```

- Prelude> **:set prompt "ghci> "**

  ghci>

- putStrLn "Hello World"

SIMPLE ARITHMATIC

- ghci> **2 + 2**

4

- ghci> **31337 * 101**

3165037

- ghci> **7.0 / 2.0**

3.5

- ghci> **2 + 2**

4

- ghci> **(+) 2 2**

4

- ghci> **-3**

-3

- ghci> **2 + -3**

<interactive>:1:0:

precedence parsing error

cannot mix `(+)' [infixl 6] and prefix `-' [infixl 6] in the same infix expression. If we want to use the unary minus near an infix operator, we must wrap the expression that it applies to in parentheses:

- ghci> **2 + (-3)**

-1

- ghci> **3 + (-(13 * 37))**

-478

- ghci> **pi**

3.141592653589793

LISTS

- A list is surrounded by square brackets; the elements are separated by commas:

ghci> **[1, 2, 3]**

[1,2,3]

- If we write a series of elements using *enumeration notation*, Haskell will fill in the contents of the list for us:
- ghci> **[1..10]**

[1,2,3,4,5,6,7,8,9,10]

- ghci> **[1.0,1.25..2.0]**

[1.0,1.25,1.5,1.75,2.0]

- ghci> **[1,4..15]**

[1,4,7,10,13]

- ghci> **[10,9..1]**

[10,9,8,7,6,5,4,3,2,1]

OPERATORS IN LISTS

- There are two ubiquitous operators for working with lists. We concatenate two lists using the (++) operator:
- ghci> **[3,1,3] ++ [3,7]**

[3,1,3,3,7]

- ghci> **[] ++ [False,True] ++ [True]**

[False,True,True]

- More basic is the (:) operator, which adds an element to the front of a list (this is pronounced "cons" [short for "construct"]):
- ghci> **1 : [2,3]**

[1,2,3]

- ghci> **1 : []**

[1]

OPERATORS IN LISTS

- In fact, a text string is simply a list of individual characters. Here’s a painful way to write a short string, which ghci gives back to us in a more familiar form:

- ghci> **let a = ['l', 'o', 't', 's', ' ', 'o', 'f', ' ', 'w', 'o', 'r', 'k']**

ghci> **a**

"lots of work,,

- ghci> **a == "lots of work"**

True

TYPES

- ghci> **:type 'a'**

 'a' :: Char

- ghci> **"foo"**

"foo"

- ghci> **:type it**

it :: [Char]

- The :type command will print type information for any expression we give it (including it, as we see here). It won’t actually evaluate the expression; it checks only its type and prints that.

- ghci> **3 + 2**

5

- ghci> **:type it**

it :: Integer

CALLING FUNCTIONS

- ghci> succ 8

9

The succ function takes one parameter that can be anything that has a well-defined successor, and returns that value. The successor of an integer value is just the next higher number.

- Now let's call two prefix functions that take multiple parameters, min and max:
- ghci> min 9 10

9

- ghci> min 3.4 3.2

3.2

- ghci> max 100 101

101

CALLING FUNCTIONS (CONTINUED…)

- Function application has the highest precedence of all the operations in Haskell. In other words, these two statements are equivalent.
- ghci> succ 9 + max 5 4 + 1

16

- ghci> (succ 9) + (max 5 4) + 1

16

- If a function takes two parameters, we can also call it as an infix function by surrounding its name with backticks (`). For instance, the div function takes two integers and executes an integral division, as follows:
- ghci> div 92 10

9

- ghci> 92 `div` 10

9

DEFINE FUNCTIONS

- Open up your favorite text editor and type in the following:

doubleMe x = x + x

- Save this file as *fc.hs*. Now run ghci, making sure that *fc.hs* is in your current directory.
- Now we can play with our new function:
- ghci> load fc.hs

[1 of 1] Compiling Main (fc.hs, interpreted)

Ok, modules loaded: Main.

- ghci> doubleMe 9

18

- ghci> doubleMe 8.3

16.6

DEFINE FUNCTIONS (CONTINUED…)

- doubleUs x y = x * 2 + y * 2 (Write and save in a text editor)

- ghci> doubleUs 2.3 34.2

73.0

- Multiple functions. Here, for example, doubleMe and doubleUs
- ghci> doubleUs 28 88 + doubleMe 123

478

DEFINE FUNCTIONS (MAIN=DO)

- main=do

- main = do

```
putStrLn "Greetings!  What is your name?"
inpStr <- getLine
putStrLn $ "Welcome to Haskell, " ++ inpStr ++ "!"
```

DEFINE FUNCTIONS (CONTINUED....)

- In the following example, we are taking a complex mathematical expression. We will show how we can find the roots of a polynomial equation [x^2 - 8x + 6] using Haskell.

```
roots :: (Float, Float, Float) -> (Float, Float)
roots (a,b,c) = (x1, x2) where
  x1 = e + sqrt d / (2 * a)
  x2 = e - sqrt d / (2 * a)
  d = b * b - 4 * a * c
  e = - b / (2 * a)
main = do
  putStrLn "The roots of our Polynomial equation are:"
  print (roots(1,-8,6))
```

IF..THEN..ELSE STATEMENTS…

- ***if condition***

then do This

else do That

- ***If condition then…***

else if condition then…

else if condition then..

.

.

else statement

```
myScore x =
    if x > 90 then "You got a A"
    else if 80 < x && x < 90 then "you got a B"
    else if 70 < x && x < 80 then "You got a C"
    else if 60 < x && x < 70 then "you got a D"
    else "You got a F"
```

PRACTICE

- Write a program in haskell that finds an integer number either odd or even.

- Write a program in haskell that calculates the periphery and area of a circle with the given radius.

- Write a program in haskell that calculates the Euclidian distance between two coordinates (x1,y1) and (x2,y2).

- Write a program in haskell that finds a year leap year or not.