Functional Programming

The most striking feature of purely functional programming is that there is no state.

This means that our variables are not variable, i.e.,
cannot change their values!

In other words, they are immutable and only represent some constant value.

The execution of a program only involves the evaluation of functions.

This sounds weird -- what are the advantages and disadvantages of functional programming?

The advantage of having no state is that functions have no side effects.

Therefore, we can be sure that whenever we evaluate a function with the same inputs, we will get the same output, and nothing in our system changed due to this evaluation.

This prevents most of the bugs that commonly occur in imperative programming.

It also allows for automatic multithreading.

You will learn about other advantages when you study Haskell more closely.

The main problem with strictly preventing side effects is that user input and output during program execution become impossible.

To enable such user interaction, we have to sometimes allow state changes. It is then important to separate such “impure” code from the rest of the program.

There are many functional languages, with some being as old as the earliest imperative ones.

Examples are: LISP, Scheme, Haskell, Erlang, R, Clojure, Scala, OCaml, and F#.

Functional programming is not the best solution to every problem, just like object-oriented programming is not, either.

In the context of theory of computation, you will see how functional programming allows you easily translate mathematical definitions into concise and clear programs.

Even if you rarely or never use Haskell again afterwards, it will give you a different perspective on programming and may change the way you program.

Haskell

Download the Haskell Platform at: https://www.haskell.org/

There are no perfect Haskell IDEs, but some good plugins for editors such as Atom, Sublime Text, or IntelliJ.

For our purposes, a simple text editor plus console a good enough.

For Windows, Notepad++ with the NppExec plugin gives you syntax highlighting and integrated coding, testing, and profiling.

Example scripts for Notepad++:

Compiling:
ghc -O -o main.exe $(FULL_CURRENT_PATH)

Interactive testing/debugging:
ghci $(FULL_CURRENT_PATH)

Profiling:
ghc -O -prof -auto-all $(FULL_CURRENT_PATH)
$(NAME_PART) +RTS -p -RTS

If you rarely or never use Haskell again afterwards, it will give you a different perspective on programming and may change the way you program.
Haskell

Free Haskell tutorials:
http://learnyouahaskell.com/
http://book.realworldhaskell.org/

I recommend that you read Chapters 1 and 2 of “Learn you a Haskell” and experiment with the language a bit. As I said before, it is not mandatory for you to learn Haskell, and it will not be relevant to homework or exams. It is just an opportunity that I am offering to you.

Demo Session (I)

Here is the protocol of our demo session with GHCi:

*Main> 3 + 4
7
*Main> 2^1000
107150860718627726732094842504060000018010561404811705533
607443750383703510511249361224931983788156958581275
9467291755314682518714528569231404359845775746985748
0393456777482423098542107460506237114187795418215304
6474983581941267398767559165543946070762914571196477
68654216766042983165262438837205668069376
*Main> "hello"
"hello"

Demo Session (II)

*Main> :t "hello"
"hello" :: [Char]
*Main> :t 3
3 :: Num a => a
*Main> :t 3.5
3.5 :: Fractional a => a
*Main> [3, 2, 1]
[3,2,1]
*Main> 4:[3, 2, 1]
[4,3,2,1]
*Main> 4:3:5:[]
[4,3,5]

Demo Session (III)

*Main> (2, 3)
(2,3)
*Main> (2, 3) == (3, 2)
False
*Main> [2, 3] == [3, 2]
False
*Main> (5, 'a')
(5,'a')
*Main> :t head
head :: [a] -> a
*Main> head [4, 6, 1]
4

Demo Session (IV)

*Main> :t tail
tail :: [a] -> [a]
*Main> tail [4, 6, 1]
[6,1]
*Main> let mult a b = a*b
*Main> mult 6 7
42
*Main> :t mult
mult :: Num a => a -> a -> a
*Main> head "hello"
'h'

Demo Session (V)

*Main> (mult 6) 8
48
*Main> let supermult = mult 6
*Main> supermult 5
30
*Main> :t supermult
supermult :: Num a => a -> a
*Main> even 7
False
*Main> :t filter
filter :: (a -> Bool) -> [a] -> [a]
 Demo Session (VI)

```haskell
*Main> :t even
even :: Integral a => a -> Bool
*Main> [1..10]
[1,2,3,4,5,6,7,8,9,10]
*Main> filter even [1..10]
[2,4,6,8,10]
*Main> :t map
map :: (a -> b) -> [a] -> [b]
*Main> map even [1..10]
[False,True,False,True,False,True,False,True,False,True]
```

 Demo Session (VII)

```haskell
*Main> map supermult [1..10]
[6,12,18,24,30,36,42,48,54,60]
*Main> :{
*Main| fc 0 = 1
*Main| fc n = n*fc (n - 1)
*Main| :}
*Main> fc 3
6
*Main> fc 10
3628800
*Main> fc 30
265285981219105863084800000000
```

 Demo Session (VIII)

```haskell
*Main> [1..10]
[1,2,3,4,5,6,7,8,9,10]
*Main> head [1..10]
1
*Main> head [1..100000000000000000000000000000000]
1
*Main> head [1..]
1
Bottom line: Haskell is lazy!
```

 Function Application

In Haskell, function application has precedence over all other operations. Since the compiler knows how many arguments each function requires, we can do the following:

```haskell
func1 x = x + 2
func2 x y = x*x + y*y
func3 a b = func1 a + func2 b a
```

No parentheses are necessary – the known signatures of func1 and func2 define how to compute func3.

 If – Then - Else

Remember that execution of purely functional Haskell code involves only function evaluation, and nothing else. Therefore, there is an if – then – else function, but it always has to return something, so we always need the "else" part. Furthermore, it needs to have a well-defined signature, which means that the expressions following "then" and "else" have to be of the same type.

```haskell
iq :: Int -> [Char]
iq n = if n > 130
    then "Wow!"
    else "Bah!"
```

You can put everything in a single line if you like, as in this example:

```haskell
max' :: Int -> Int -> Int
max' x y | x > y then x else y
```
Some Functions on Lists

head \( xs \): Returns the first element of list \( xs \)
tail \( xs \): Returns list \( xs \) with its first element removed
length \( xs \): Returns the number of elements in list \( xs \)
reverse \( xs \): Returns a list with the elements of \( xs \) in reverse order
null \( xs \): Returns true if \( xs \) is an empty list and false otherwise

Some Functions on Tuples

fst \( p \): Returns the first element of pair \( p \)
snd \( p \): Returns the second element of pair \( p \)
This only works for pairs, but you can define your own functions for larger tuples, e.g.:
\[
\begin{align*}
\text{fst3 :: (a, b, c) -> a} \\
\text{fst3 (x, y, z) = x}
\end{align*}
\]
You can always replace variables whose values you do not need with an underscore:
\[
\text{fst3(x, _, _) = x}
\]

Pattern Matching

You can define separate output expressions for distinct patterns in the input to a function. This is also the best way to implement recursion, as in the factorial function:
\[
\begin{align*}
\text{fact 0 = 1} \\
\text{fact n = n*fact (n - 1)}
\end{align*}
\]
Similarly, we can define recursion on a list, for example, to compute the sum of all its elements:
\[
\begin{align*}
\text{sum' [] = 0} \\
\text{sum' (x:xs) = x + sum' xs}
\end{align*}
\]

Guards

In pattern matching, you have to specify exact patterns and values to distinguish different cases.
If you need to check inequalities or call functions in order to make a match, you can use guards instead:
\[
\begin{align*}
\text{iqGuards :: Int -> [Char]} \\
\text{iqGuards n} \\
| \ n > 150 = \text{"amazing!"} \\
| \ n > 100 = \text{"cool!"} \\
| \ \text{otherwise} = \text{"oh well..."}
\end{align*}
\]
Currying

As you know, you can turn any infix operator into a prefix operator by putting it in parentheses:

\((+ \ 3)\ 4\)

7

Now currying allows us to place the parentheses differently:

\((+\ 3)\ 4\)

7

By “fixing” the first input to \((+)\) to be 3, we created a new function \((+\ 3)\) that receives only one (further) input.

We can check this:

\(\text{:(t (+ 3)}\)

\((+ 3) :: \text{Num a => a -> a)}\)

This \((+ 3)\) function can be used like any other function, for example:

map \((+ 3)\) \([1..5]\)

\([4, 5, 6, 7, 8]\)

Or:

map \((\text{max\ 5})\) \([1..10]\)

\([5, 5, 5, 5, 6, 7, 8, 9, 10]\)

Lambda Expressions

Sometimes we just need a small local function that is not used anywhere else in the program.

Then it is convenient to use a lambda expression, which is a way of defining an anonymous function directly in the place where we use it.

They have the following form:

\(\lambda\ \text{input} \rightarrow \text{output}\)

For example:

\((\lambda\ x \rightarrow 3*x)\ 4\)

12

More examples for lambda expressions:

\(\text{zipWith (+)}\) \([1..10]\) \([11..20]\)

\([12,14,16,18,20,22,24,26,28,30]\)

\(\text{zipWith (\(\lambda\ x \ y \rightarrow x^2 + y^2\))} [11..20]\)

\([122,148,178,212,250,292,338,388,442,500]\)

\(\text{map (\(\lambda\ x \rightarrow (x, x^2, x^3))\)} [1..10]\)

\([(1, 1, 1),(2, 4, 8),(3, 9, 27),(4, 16, 64),(5, 25, 125)]\)