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.

Demo Session (I)

Here is the protocol of our demo session with GHCi:

```
Prelude> 3 + 4
7
Prelude> 2 `1000
107150860718626732094825049060001810561404811705533
60744375038370351051249361224931983788156958581275
9466729175531468251873452856923140435984577546985748
039345677776824308542107460505623711428779543215304
6474983581942673987655351655439460770629124571196477
686542167660429831652624386837205668069376
Prelude> "hello"
"hello"
```

Demo Session (II)

```
Prelude> :t "hello"
"hello" :: [Char]
Prelude> :t 3
3 :: Num a => a
Prelude> :t 3.5
3.5 :: Fractional a => a
Prelude> [3, 2, 1]
[3,2,1]
Prelude> 4:[3, 2, 1]
[4,3,2,1]
Prelude> 4:3:5:[]
[4,3,5]
```
Demo Session (III)

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

Demo Session (IV)

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

Demo Session (V)

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

Demo Session (VI)

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

Demo Session (VII)

Prelude> map supermult [1..10]
[6,12,18,24,30,36,42,48,54,60]
Prelude> :
Prelude| let fc 0 = 1
Prelude| fc n = n*fc (n - 1)
Prelude| :
Prelude| fc 3
6
Prelude> fc 10
3628800
Prelude> fc 30
265252859812191058636308480000000

Demo Session (VIII)

Prelude> [1..10]
[1,2,3,4,5,6,7,8,9,10]
Prelude> head [1..10]
1
Prelude> 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:

\[
\begin{align*}
\text{func1} \ x &= x + 2 \\
\text{func2} \ x \ y &= x^x + y^y \\
\text{func3} \ a \ b &= \text{func1} \ a + \text{func2} \ b \ a \\
\end{align*}
\]

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.

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:

\[
\begin{align*}
\text{fact } 0 &= 1 \\
\text{fact } n &= n \times \text{fact } (n - 1)
\end{align*}
\]

Similarly, we can define recursion on a list, for example, to compute the sum of all its elements:

\[
\begin{align*}
\text{sum } \emptyset &= 0 \\
\text{sum } (x:xs) &= x + \text{sum } xs
\end{align*}
\]

Pattern Matching
We can even do things like this:

\[
\begin{align*}
\text{testList } \emptyset &= \text{"empty"} \\
\text{testList } (x:[]) &= \text{"single-element"} \\
\text{testList } (x:xs) &= \text{"multiple elements. First one is "} + \text{show } x
\end{align*}
\]

The last line demonstrates how we can use pattern matching to not only specify a pattern but also access the relevant input elements in the function expression. Since we do not use xs in that line, we could as well write

\[
\begin{align*}
\text{testList } (x:_:) &= \text{"multiple elements. First one is "} + \text{show } x
\end{align*}
\]

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:

\[
\begin{align*}
\text{iqGuards } :: \text{Int} \rightarrow [\text{Char}] \\
\text{iqGuards } n &= \begin{cases}
\text{"amazing!"} & \text{if } n > 150 \\
\text{"cool!"} & \text{if } n > 100 \\
\text{"oh well..."} & \text{otherwise}
\end{cases}
\end{align*}
\]

Recursion
Since variables in Haskell are immutable, our only way of achieving iteration is through recursion. For example, the reverse function receives a list as its input and outputs the same list but with its elements in reverse order:

\[
\begin{align*}
\text{reverse } :: [a] \rightarrow [a] \\
\text{reverse } [] &= [] \\
\text{reverse } (x:xs) &= \text{reverse } xs ++ [x]
\end{align*}
\]

Recursion
Another example: The function zip takes two lists and outputs a list of pairs with the first element taken from the first list and the second one from the second list. Pairs are created until one of the lists runs out of elements.

\[
\begin{align*}
\text{zip } :: [a] \rightarrow [b] \rightarrow [(a,b)] \\
\text{zip } [] &= [] \\
\text{zip } [] &= [] \\
\text{zip } (x:xs) (y:ys) &= (x,y):\text{zip } xs ys
\end{align*}
\]
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.

Currying

We can check this:

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

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

\[\text{map}\ (+\ 3)\ \text{[1..5]}\]
\[[4, 5, 6, 7, 8]\]

Or:

\[\text{map}\ \text{(max 5)}\ \text{[1..10]}\]
\[[5, 5, 5, 5, 5, 6, 7, 8, 9, 10]\]

Lambda Expressions

More examples for lambda expressions:

\[\text{zipWith}\ \text{\((\lambda x\ y \rightarrow x^2 + y^2)\)}\ \text{[1..10]}\ \text{[11..20]}\]
\[[122, 148, 178, 212, 250, 292, 338, 388, 442, 500]\]

\[\text{map}\ \text{(\((\lambda x \rightarrow (x, x^2, x^3))\)}\ \text{[1..5]}\]
\[\{(1, 1, 1), (2, 4, 8), (3, 9, 27), (4, 16, 64), (5, 25, 125)\}\]

The $ Operator

The $ operator is defined as follows:

\[f\ \text{$\ x = f\ x}\]

It has the lowest precedence, and therefore, the value on its right is evaluated first before the function on its left is applied to it.

As a consequence, it allows us to omit parentheses:

\[\text{negate}\ \text{(sum (map sqrt [1..10]))}\]

can be written as:

\[\text{negate\$ sum\$ map\ sqrt\ [1..10]}\]

Function Composition

Similarly, we can use function composition to make our code more readable and to create new functions. As you know, in mathematics, function composition works like this:

\[(f \circ g) (x) = f(g(x))\]

In Haskell, we use the \"\cdot\" character instead:

\[\text{map}\ \text{(\((\lambda xs \rightarrow \text{negate}\ (\text{sum}\ (\text{tail}\ xs))\))}\ [[1..5],[3..6],[1..7]]\]

Can be written as:

\[\text{map}\ \text{(\((\lambda x\ \rightarrow\ \text{negate}\ \text{. sum}\ \text{. tail})\))}\ [[1..5],[3..6],[1..7]]\]

Data Types

Data types can be declared as follows:

\[\text{data}\ \text{Bool}\ =\ \text{False} \mid \text{True}\]

\[\text{data}\ \text{Shape}\ =\ \text{Circle}\ \text{Float}\ \text{Float}\ \text{Float} \mid \text{Rectangle}\ \text{Float}\ \text{Float}\ \text{Float}\ \text{Float}\]

\[\text{deriving}\ \text{Show}\]

Then we can construct values of these types like this:

\[x = \text{Circle}\ 3\ 4\ 5\]

The "deriving Show" line makes these values printable by simply using the show function (object-to-string conversion) as it is defined for the individual objects (here: floats).
Data Types

We can use pattern matching on our custom data types:

surface :: Shape -> Float
surface (Circle _ _ r) = 3.1416 * r ^ 2
surface (Rectangle x1 y1 x2 y2) = (abs $ x2 - x1) * (abs $ y2 - y1)

surface $ Circle 10 20 10
314.16

Records

If we want to name the components of our data types, we can use records:

data Car = Car {company :: [Char], model :: [Char], year :: Int} deriving Show
myCar = Car {company="Ford", model="Mustang", year=1967}

company myCar
"Ford"

Input/Output with “do”

Purely functional code cannot perform user interactions such as input and output, because it would involve side effects. Therefore, we sometimes have to use impure functional code, which needs to be separated from the purely functional code in order to keep it (relatively) bug-safe. In Haskell, this is done by so-called Monads. To fully understand this concept, more in-depth study is necessary.

However, in this course, we do not need to perform much input and output. We can use a simple wrapper (or “syntactic sugar”) for this — the “do” notation.

Input/Output with “do”

In a do-block, we can only use statements whose type is “tagged” IO so that they cannot be mixed with purely functional statements.

Example for a program performing input and output:

main = do
    putStrLn "Hello, what's your name?"
    name <- getLine
    putStrLn ("Hey " ++ name ++ ", you rock!")