MORE ELEMENTS OF HASKELL BY EXAMPLE

WIM VANDERBAUWHEDE

A few more basic (and not-so-basic) elements of Haskell through comparison with other languages. We will not go into detail on the Haskell constructs, just show the similarities with constructs from languages you may know.

Blocks. In JavaScript functions typically are blocks of code:

```
    function roots(a,b,c) {
        det2 = b*b-4*a*c;
        det  = sqrt(det);
        rootp = (-b + det)/a/2;
        rootm = (-b - det)/a/2;
        return [rootm,rootp]
    }
```

In Haskell, we would write this function as follows:

```
    roots a b c =
        let
            det2 = b*b-4*a*c;
            det  = sqrt(det);
            rootp = (-b + det)/a/2;
            rootm = (-b - det)/a/2;
        in
            [rootm,rootp]
```

Note that the `let ... in ...` construct is an *expression*, so it returns a value. That's why there is no need for a `return` keyword.

Conditions. In Python we could write a function with a condition as like this:

```
def max(x,y):
    if x > y:
        return x
    else:
        return y
```

Of course Haskell also has an if-then construct:

```
max x y =
    if x > y
        then x
        else y
```

Again the `if ... then ... else ...` construct is an *expression*, so it returns a value.

Case statement. Many languages provide a `case` statement for conditions with more than two choices. For example, Ruby provides a `case` expression:

```
Red = 1
Blue = 2
Yellow = 3

color = set_color();
action = case color
    when Red then action1()
    when Blue then action2()
    when Yellow then action3()
end
```

In `Haskell`, the case works and looks similar:

```
data Color = Red | Blue | Yellow

color = set_color
action = case color of
    Red -> action1
    Blue -> action2
    Yellow -> action3
```

Note however how we use the type as the value to decide on the case, where in other languages we need to define some kind of enumeration.

Generics/Templates. In `Java` and `C++` there are generic data types (aka template types), such as:

```
Map<String,Integer> set = new HashMap<String,Integer> ;
```

In `Haskell`, you would write this as follows:

```
set :: Data.Map.Map String Integer
set = Data.Map.empty
```

The main difference is of course that `set` in `Haskell` is not an object but an immutable variable, so where in Java you would say:

```
set.put("Answer",42)
```

In `Haskell` you would say:

```
set' = insert "Answer" 42 set
```

Because in `Haskell` variables are immutable, the return value of the `insert` call is bound to a new variable rather than updating the variable in place as in `Java`.