Contents

Functional Programming is Different **1**

Workflow **1**

Programming Toolbox **2**

Haskel Language Structure **2**

Function Definition **2**

Patterns in Mathematics 3

Cases by Pattern Matching 3

Partial Functions **4**

Pattern Matching on Structures **4**

Order in Pattern Matching **5**

Definition by Cases **5**

Guarded Alternatives 5

Functional Programming is Different

- ***Balance of effort***: It's not only common but *desirable* that you spend much more time thinking about how to structure your code than writing the code
- ***Workflow***: A lot of you wll struggle initially with the workflow of writing functional programs, because the approach to programming you've learned from imperative languages is build & fix, and iterate on fixing until the program works

Workflow

- *What do I have?* Identify the parts of the problem that need to be represented with datatypes

- *What do I want?* Model the solution to the problem as an instance of some datatype
- *How do I get there?* Imagine a function mapping the problem to the solution (both represented as an instance of some data type)
- Try to write this function as a sequence of steps
- Use this same process to construct each of the steps
- Eventually a step in some language or library primitive or a one-liner

Programming Toolbox

- Development enironment (OS/shell/editor)
- Source control - `git`
- Automated testing - `stack test`
- Build systems - `stack`
- Different kinds of safety (tests/types)
- Syntax Cheatsheet

Haskel Language Structure

- Haskell is built on top of a simple functional language (Haskell "Core")
- A lot of syntactic sugar is added
  - `"ad"` for `['a', 'b']` for `'a':'b':[]`
- Large collection of standard libraries and functions are predefined and preloaded (the Haskell "Prelude")
- Vast number of libraries available

Function Definition

We have seen that functions are defined in Haskell as sets of equations:

```
sum[] = 0
sum(n:ns) = n + sum(ns)
```

Let's look in more detail at how these are written, and how the system selects which equation applies in any given case

Patterns in Mathematics

In mathematics, we often characterise something by laws it obeys, and these laws often look like patterns of templates

$0! = 1$

$n! = n \times (n-1)!, n > 0$

$len(<>) = 0$

$len(l_1 \cap l_2) = len(l_1) + len(l_2)$

Pattern matching is inspired by this (but with some pragmatic differences)

Cases by Pattern Matching

When there is more than one equation defining a function there must be some way to select between them.

One particularly convenient way to do this is *pattern matching*

```
myfun 0 = 0
myfun 1 = 1
myfun n = myfun(n-2)
```

These three equations give a definition for a function. When the function is applied to an integer Haskell selects on of the equations by matching the actual parameter against the patterns.

- Consider application of `myfun` to `3`

```
> myfun 3
```

- Haskell tries to match `3` against `0`, and fails
- Haskell tries to match `3` against `1`, and fails
- Haskell tries to match `3` against `n`, and succeeds, binding `n` to `3`

```
myfun(3-2)
myfun 1     -- subtraction gets done
```

- Haskell tries to match `1` against `0`, and fails
- Haskell tries to match `1` against `1`, and succeeds

Patterns can be used to give an elegant expression to certain functions, for instance we can define a function over two `Bool` arguments like this:

```
and True True = True
and _    _    = False
```

The special pattern "_" is a wildcard and will match any value without binding it to a name. It is usually used to indicate that the value is not needed on the right-hand side

Partial Functions

- We can select equations using quite complex patterns. This set is non-exhaustive, and so the `sum3` function is *partial*

```
sum3 [] = 0
sum3 [x] = x
sum3 [x, y] = x+y
sum3 [x, y, z] = x+y+z
```

It is an error to apply a partial function outside the domain it is define for (division is another example of a partial function)

`sum3 [1, 2, 3, 4]` gives a runtime error.

Pattern Matching on Structures

Here is a partial function from the Prelude:

```
> head [1, 2, 3]
1
```

It povides the first element of a list. We can define it using pattern matching:

```
head(x:xs) = x
```

The pattern in this case is based on something called a *constructor*.

The pattern match relies on the fact that lists are built out of individual elements using "`[]`" (empty list) and "`:`" ("cons"[2]). It's only for convenience that we use the comma-list notation.

```
> 1:[]
[1]
> 2:3:[]
[2, 3]
> 1:[2:3]
[1, 2, 3]
```

So in the pattern match for `head`

```
`head(x:xs) = x
```

An argument like `[1, 2, 3]` would result in `x` being associated with `1` and `xs` being associated with `[2, 3]`

Order in Pattern Matching

- Patterns are matched in order until a match occurs, or all fail to match (runtime error)
  - We don't re-evaluate arguments from scratch for later patterns
  - An early general match (`x:xs`) masks a later more specific one (`1:xs`)
- This "first-come first-served" approach handles overlapping patterns gracefully
- Haskell can warn about overlapping and some missing patterns

Definition by Cases

Often we want different equations for different cases for different cases. Mathematically we sometimes write something like this:

$$signum(x) = \begin{cases} 0\, if\ x = 0 \\ -1\, if\ x < 0 \\ 1\, if\ x > 0 \end{cases}$$

This indicates that there are three different cases in the definition of `signum`, and gives a rule for choosing which case should applied in any particular application

Guarded Alternatives

Another way to make choices it to use *guarded alternatives*:

```
signum x | x < 0  = -1
         | x == 0 = 0
         | x > 0  = 1
```

- The guards are boolean expressions
- If a guard is true the corresponding right-hand side will be selected as the definition for the function

Each guard is tested in turn, and the first one to match selects an alternative. This means that it is OK to have a guard that would always be true, as long as it is the *last* alternative.

So the previous definition could have been written like this

```
signum x | x < 0  = -1
         | x == 0 = 0
         | True   = 1
```

- For readability the name `otherwise` is allowed as a synonym for `True`
  - `True and False` is not the same as `1 and 0`

```
signum x | x < 0     = -1
         | x == 0    = 0
         | otherwise = 1
```

We can use guards to select special cases in functions. This function is `True` when the year number is a leap year

```
leapyear :: Int -> Bool
leapyear y | mod y 400 == 0  = True  -- 2000 was
           | mod y 100 == 0  = False -- 1900 wasn;t
           | mod y 4   == 0  = True  -- 2016 is
           | otherwise       = Fase  -- 2017 won't be
```

A more compact form

```
leapyear y | y `mod` 100 == 0 = y `mod` 400 == 0
           | otherwise        = y `mod` 4 == 0
```

In Haskell, any function of two arguments may be written infix it t is surrounded by backquotes, which is why `mod` is ok

Guards and patterns can be combined

```
startswith _ [] = False
startswith c (x:xs) | x == c    = True
                    | otherwise = False
```

First the patterns are matched; when an equation is found the guards are evaluated in order in the usual way.

If no guard matches then we return to the pattern matching stage and try to find another equation