Contents

Introduction

- We shall use the GHC compiler
  - Either version 7.10.3 or 8.0.1
- Coursework wil be based on the use of the `stack` tool

What is a functional programming language?

- Basic notion of computation: the application of fuctions to arguments
- Basic idea of program: writing function definitions
- Functional languages are declarative: *what* rather than *how*
- This course covers he Haskell language - A well-known and widely supported function programming language

What is not a functional language?

- *Imperative* programming languages
  - C, C++, Java, Perl, Python, Assembler, etc.
  - Basic notion of computation: issuing commands to change variable values
- Other *Declarative* languages
  - e.g. Prolog (multi-directional, logic-based)
  - Basic notion of computation: compute whether a constraint can be satisfied

Why functional?

- In terms of programming:
  - **Concise**. Shorter, clearer, more maintainable code
  - **Understandable**. Emphasis on *what*, not *how* results in programs that are easier to understand.
  - **Reliable**. Many kinds of common error become impossible.
  - **Powerful abstractions**. There is a smaller "semantic gap" between the programmer and the language.
- Disadvantages:
  - Lose some ability to do detailed low-level tuning of algorithms (bit twiddling is harder in Haskell than in C).
  - Slower (but often not by much) - for general purpose code.
- In broader terms:
  - Programming is a craft like any other.
  - Good tools are a force multiplier for productivity.
  - Building systems in a function style yields safer, mode maintainable, and more reliable code *faster*.

  - Function programming opens up interesting possibilities:
    * Automated, property-based testing - testing your code is important, so testing should be fast and painless.
    * Truly modular systems - pure programs have reliable, repeatable behaviours and well-defined interfaces.
    * Engineers don't need to keep the entire application structure in their head.
    * Properly decoupled development - a pure, strongly typed interface is a contract that the compiler enforces.

Why Haskell?

- Active community
- Great performace (Competitive with Java, C++, etc.)
- Extension library ecosystem (hackage.haskell.org)
- Good tools & editor support (cabal, ghci, ghc-mod, stack)

What does it look like?

```
> 3+1
4
> "this" == "that"
False
> reverse "that"
"taht"
```

- These examples were executed in GHCi, a Haskell interpreter.
- Symbols `+`, `==` and `reverse` are standard builtin functions in Haskell.

Definined Haskell values

- Function definitions are written as equations
  - name on the left hand side
  - value of expression on the right hand side
- `phrase = "able was i ere i saw elba"`
- This is a name definition, not an assignment statement.
  - `phrase` becomes a shorthand for the string.
  - Definitions are immutable
  - More like `#DEFINE`

Defining Haskell functions

Functions are *parameterised* values, which will allow us to write rather more useful definitions.

```
double x = x + x
```

The functional can be applied to a number to produce another number.

```
> double 2
4
> double (double 2)
8
```

They can also be defined in terms of other functions.

```
quadruple x = double (double x)
```

- Function Application is ubiquitous in Haskell
- So applying function `fun` to argument `arg` is written `fun arg`.
  - The normal mathematical convention would be `fun(arg)`
- Function with multiple arguments have them seperated by spaces: `fun arg1 arg2 arg3`
- Function application in Haskel is so commmon it has the simplest syntax possible.

Lists

A key datastructure in Haskell is the *List*:

- A list of integers: `[-3, -2, -1, 0, 1, 2, 3]`
- A list of characters: `['h', 'e', 'l', 'l', 'o']`
  - For character lists (String) we use *syntactic sugar*: `"hello"`
- We can think of a list as being either:
  - Empty - `[]`
  - An element stuck onto the front of a list - `x:xs`
- So the notation `[1, 2, 3]` is syntactic sugar for `1:(2:(3:[]))`

Function of Lists

- We shall define a function to compute the length of a list
- For lists we can use *pattern-matching*
- An empty list has length 0, `length [] = 0`
- An non-empty list has length one greater than its "tail"
  - `length (x:xs) = 1 + length xs`
  - Recursion is the natural way to describe repeated computation
- Without pattern mtching we could have been forced to write something like:

```
length xs = if null xs
            then 0
            else 1 + length (tail xs)
```

- Define a function to compute the sumer of a (numeric) list

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

Higher Order Functions

- We can define functions that
  - take other functions as arguments
  - return functions as results
- Consider `length`, `sum`, and `prod`:
  - They had a specific value for empty list (call it `e`)
  - They had a specific function to combine the "head" element with the recursive result (call it `op`)
- Let us wrap this up as a special function that takes `e` and `op` as arguments:

```
fold e op [] = e
fold e op (x:xs) = op x (fold e op xs)
```

- We have captured the recursion pattern common to all three functions

Using fold

- For `sum`, we use `0` for the empty list, and add to combine values in the recursive case
    - `sum = fold 0 (+)`
    - Notice we can pass the "+" function in as an argument
- For `prod`, we use `1` for the empty list, and multiplu to combine values in the recursive case
    - `prod = fold 1 (*)`
- For `length`, we use `0` for the empty list, and add one to combine values in the recursive case

```
length = fold 0 incsnd
incsnd x y = y + 1
```

Here we need to define `incsnd` to ignore its first argument and increment the second

Types in Haskell

- Haskell is a strongly types language
- Every value has a type:
- `Int`
- `Integer` (big int - grows in size)
- `Char`
- `[Int]` list of `Int`
- Haskell can infer types itself (Type Inference)

Type Polymorphism

- What is the type of `length`?

```
> length [1, 2, 3]
3
> length ['a', 'b', 'c', 'd']
4
> length [[], [1, 2], [3, 2, 1], [], [6, 7, 8]]
5
```

- `length` works for a list of elements of arbitrary type: `length :: [a] -> Int`

- Here `a` denotes a type variable, so the above reads as “`length` takes a list of type `a` and returns an `Int`”
- Similar notion of “generics” in O-O languages, but builtin without fuss.

Laziness

- What’s wrong with the following (recursive) definitions?
  - `from n = n : (function (n+1))`
- Nothing! (Provided we do not try to evaluate all of it (???))
- Builtin function `take` takes a number and list as arguments: `take n list` - return first `n` elements of `list`.
- What is `take 10 (from 1)`?

```
> take 10 (from 1)
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

- Haskell is a *lazy* languages, so values are evaluated only when needed.

Programming Compactness

- A key advantage of Haskell is its compactness
- Sorting the empty list gives the empty list: `qsort [] = []`
- Sorting a non empty list uses the head as pivot, and partitions the rest into elements less than or greater than the pivot

```
qsort (x:xs)
  = qsort [y | y <- xs, y < x]
    ++ [x]
    ++ qsort [z | z <- xs, z >= x]
```

- We have used Haskell list comprehensions `[y | y <- xs, y < x]`
  - Build list of ys, where `y` is drawn from `xs`, such that `y < x`