What is a functional programming language?

- Basic notion of computation: the application of functions to arguments.
- Basic idea of program: writing function definitions
- Functional languages are declarative: *what* rather than *how*.
- This course covers the Haskell language
  A well-known and widely supported functional 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
  - Basic idea of program:
    writing assignment (`a := b`)
    and control flow (`while`, `for`, `if`, `switch`) statements
- other *Declarative* languages
  - e.g. Prolog (multi-directional, logic-based)
  - Basic notion of computation:
    compute whether a constraint can be satisfied
  - Basic idea of program: asking a specific question
    `prime? 343`

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 ("No segfaults!").
- **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.

Why functional?

In broader terms:

- Programming is a craft like any other.
- Good tools are a force multiplier for productivity.
- Building systems in functional style yields safer, more maintainable, and more reliable code *faster*.
- Functional 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 performance (competitive with Java, C++, etc.)
- Extensive library ecosystem (hackage.haskell.org)
- Good tools & editor support (cabal, ghci, ghc-mod, stack)

What does it look like?

Here are some basic Haskell functions being applied:

```
> 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

Defining Haskell values

- Function definitions are written as an equations
  - name on the left hand side
  - value or expression on the right
- `phrase = "able was i ere i saw elba"`
- This is a name definition, not an assignment statement.

On loading this definition, we can use it:

```
> phrase
"able was i ere i saw elba"
```

Defining Haskell functions

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

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

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

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

Of course, functions can be defined in terms of other functions:

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

A Note on Haskell function notation

- Function Application is ubiquitous in Haskell
- So applying function `fun` to argument `arg` is written

  `fun arg`

  The normal mathematical/prog.-lang. convention would be

  `fun(arg)`
- Function with multiple arguments have them separated by spaces:

  `fun arg1 arg2 arg3`

  (instead of `fun(arg1,arg2,arg3)`)
- Function application in Haskell is so common it has the simplest syntax possible.

Functional Highlights

We are now going to do a whistle-stop tour of some of the forthcoming highlights of functional programming, including

- Pattern-Matching in Definitions
- Higher-Order Functions
- Types
  - Inference
  - Polymorphism
  - Classes
- Laziness
- Program Compactness
- etc ...

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 (e.g. `Strings`) we use *syntactic sugar*:

  `"hello"`
- We can think of a list as being either:
  - Empty— `[]`
  - An element stuck ("consed") onto the front of a list— `x:xs`
- So the notation `[1,2,3]` is syntactic sugar for `1:(2:(3:[]))`.

Functions on Lists (I)

- 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 matching we would have been forced to write something like:

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

Functions on Lists (II)

- Flushed with success . . .
- Define a function to compute the sum of a (numeric) list
- ```
  sum []     = 0
  sum (n:ns) = n + sum ns
  ```
- Define a function to compute the product of a (numeric) list
- ```
  prod []     = 1
  prod (n:ns) = n * prod ns
  ```
- Notice a pattern ?

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 (+)
  ```

  Note we can pass the "+" function in as an argument
- For `prod`, we use 1 for the empty list, and multiply 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 typed language
- Every value has a type:
  - `Int` (machine-width integer)
  - `Integer` (big-int — grows in size)
  - `Char` (characters)
  - `[Int]` list of `Int`
  - `[Int] -> Int` – function from `Int`-list to `Int`.
- But where are types declared?
- Haskell can infer types itself (Type Inference)
  It is rare that the programmer is required to specify these !

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 lists of elements of arbitrary type
  `length ::  [a] -> Int`
- Here '`a`' denotes a type variable, so the above reads as "`length` takes a list of (arbitrary) type `a` and returns an `Int`".
- A similar notion to "generics" in O-O languages, but builtin without fuss.

Laziness

- What's wrong with the following (recursive) definition ?
```
from n =  n : (from (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* language, so values are evaluated only when needed.
- How ? — later on in the course ...

Program 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 `y`s, where `y` is drawn from `xs`, such that `y < x`"
- Try that in Java !

Whistle ... Stop!

- Enough said !
- Haskell is powerful, and quite different to most mainstream languages
- It allows very powerful programs to be written in a concise manner
- It has many real-world features as well that we shall uncover:
  - e.g., File I/O

Haskell for CS3016

- We shall use the GHC compiler
  https://www.haskell.org/downloads
  Version 7.10.3
- Coursework will be based on the use of the stack tool
  https://www.stackage.org (using lts-6.19).
  https://docs.haskellstack.org/en/stable/README/
- Install stack and let it install ghc, at least as far as this course is concerned (see Lab00, to come).

Course Timetable (2016–17)

- Timetable:
  - Mon 2pm LB 0.1/ICTLabI&II : Lecture/Labs
  - Thu 2pm LB 0.4/ICTLab I : Lecture/Labs
    This week will be a Lecture in LB 0.4
  - Fri 3pm LB 0.1 : Lecture/Tutorial
- Class Management: Blackboard
- Assessment
  - Exam : 75%, introducing some multiple choice.
  - Continuous Assessment : 25%
- Notice: there will be no classes during the week of 27th November–1st December..