

# Session 1: Introduction

COMP2221: Functional programming

---

Lawrence Mitchell\*

\*`lawrence.mitchell@durham.ac.uk`

*Haskell*

- Functional programming: what is it?
- Course philosophy & organisation
- Why do we want programming languages anyway?
- Some taster examples

First practicals start in week 2. Problem sheets are hosted on the course webpage at <https://teaching.wence.uk/comp2221>.

# A simple example, computing $n!$

## Imperative style

```
factorial = 1  
for i in range(1, n+1):  
    factorial = factorial * i
```

*Assign*  
*loop*  
*reassign*

## Functional style

```
def factorial(n):  
    if n == 1:  
        return 1  
    else:  
        return n * factorial(n-1)
```

$$F_n = \begin{cases} 1 & n = 1 \\ nF_{n-1} & \text{otherwise} \end{cases}$$

$$\left. \begin{array}{l} n \geq 1 \\ n \in \mathbb{N} \end{array} \right\}$$

$$F_n : \mathbb{N}^+ \rightarrow \mathbb{N}^+$$

$$F_n(2) \rightarrow 2$$

Which implementation maps more naturally onto a computer?

Which implementation is more convenient for the programmer?

# What is a functional language?

As with most things, there are multiple opinions on precise definitions but broadly:

- A *style of programming* where the building block of computation is application of functions to arguments;
- ⇒ a functional language is one that *supports* and *encourages* programming in this style.

But isn't every programming language about functions and applying them to arguments?

# Side effects

## Definition (Side effect)

Modify some (internal/hidden) state as well as returning a value

$y1 = f(1)$   
 $y2 = f(1)$

Will  $y1 == y2$ ?

How could it not?

*will  $y1$  always equal  $y2$ ?*

# Side effects

## Definition (Side effect)

Modify some (internal/hidden) state as well as returning a value

```
y1 = f(1)
y2 = f(1)
```

Will `y1 == y2`?

How could it not?

If `f` has some internal state that affects the answer:

```
state = 0
def f(n):
    global state
    state += 1
    return n + state
```

```
print(f(1)) => "2"
print(f(1)) => "3"
```

Difficult to reason  
about the program.

Global state.  
Random # generator.

$f(1) == f(1)$  always  
in pure functional lang

# A functional approach

- Forbid variable assignment and side effects in the language.  
“Pure functional” *referential transparency.*
- ✓ Makes *reasoning* about code simpler (for humans and compilers).
- ✗ A new programming *paradigm*: takes some time to get used to.

Why not C/Java/Python? / *JavaScript?*

- ✓ It is *possible* to write in a functional style in these languages...
- ✗ but the language does not enforce it.
- ✗ Moreover, the language-level support is weak
- ✓ In contrast, Haskell is a purely functional (side effects not allowed!), and built from scratch for functional programming

*the 'research lang.' for FP.*  
*→ lots of modern langs are introducing Haskell concepts.*

# Goals of this course

- *Understand* Haskell and functional applications and write your own code.

⇒ practice via practicals

- Provide academic background: revealing underlying programming *paradigms*
- Discuss pros and cons of the functional style (performance, correctness, ease of implementation, ...) in different application scenarios.
- Link into related areas such as equational reasoning, automated proof systems, and parallel programming.



# Building block summary

- Prerequisites: none
- Content
  - Look at toy problem from both a functional and imperative point of view
  - Define some basic terms; functional style, side effects, functional programming language
- Expected learning outcomes
  - student *knows* the definition of functional programming and side effects
  - student can *explain* side effects with some examples
  - student can *apply* definition of side effects to determine if some code fragment is side effectful

# Underlying book

- Course follows (first half of) Graham Hutton's Haskell book, *Programming in Haskell* (2016)
- Slides for the first 10 chapters are available at <http://www.cs.nott.ac.uk/~pszgmh/pih.html>
- Sometimes, we'll use them directly
- Sometimes, I'll ask you to look through them in your own time
- Course will make links with other material/programming languages (Java/C/Python)  $\Rightarrow$  seen in other submodules

## Lectures

- 10 lectures
- Split into small(ish) pieces
- Learning outcomes on slides
- Typically start with brief recap at start of each lectures

## Practicals / homework

- As well as theoretical aspects, programming requires practice
- Although not compulsory, the formative practical sessions are important: *do attend*
- via Zoom (see DUO for details).

## Assessment

- By exam (no coursework)
- *knowledge* and *comprehension*: how do things work in Haskell, why do they work, ...
- *application*: what does some code do; can you write code to solve problem X...
- *evaluation*: what are the concepts; what properties does some solution have...
- Past papers available: some changes from previous years due to online aspect this year. I'll make a sample paper available too.
- We will go through one such paper in a term 3 revision lecture

# Changes from last year

- More exposition on *type classes*, especially as a way of defining abstract interfaces.
- ⇒ reaction to feedback from last year's cohort
  - A little more focus on theoretical underpinnings and compilation of abstractions to machine code.
- ⇒ help to understand where Haskell ideas are adopted elsewhere.
  - Not much focus on algorithmic complexity (not all non-CS students have seen it) ⇒ focus on elegant code instead.

## Feedback/questions

- Discussion forum: **<https://github.com/wenceorg/comp2221/discussions>**
- Happy to take them in live sessions
- Feedback form (anonymous submission allowed, but please do not abuse): see course webpage.

# Why programming languages?

---

# Abstracting from the machine

## Pseudo machine-code

$$b = a + 3$$

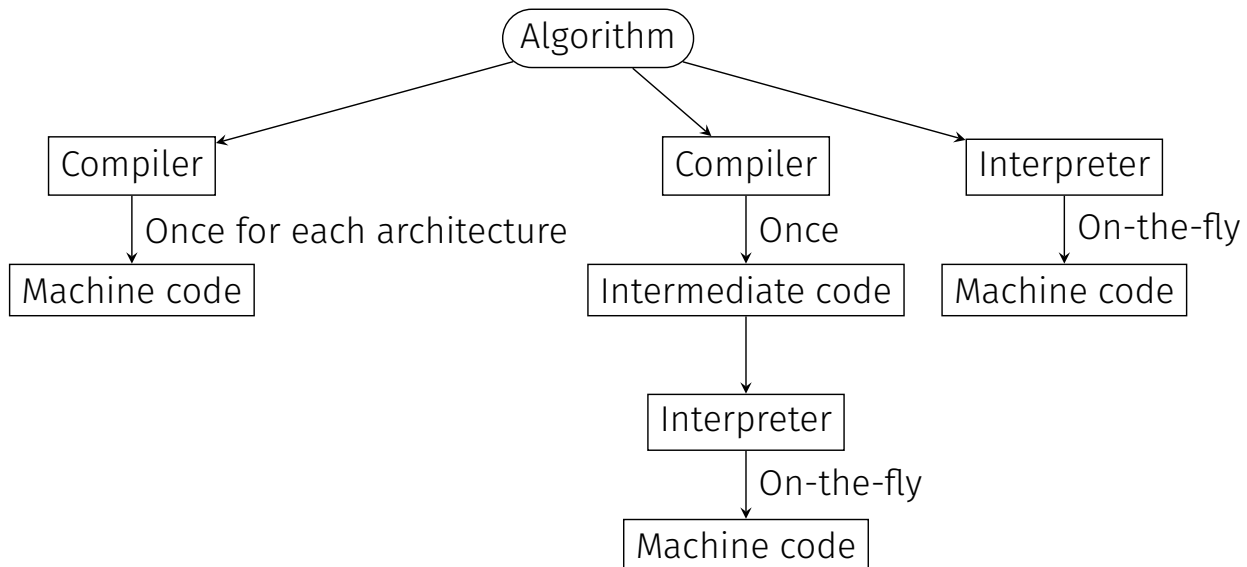
```
mov  addr_a, reg1  ## Load address of a into a reg1
add  3, reg1, reg2 ## add 3 to reg1 and write into reg2
mov  reg2, addr_b  ## write reg2 to address of b.
```

Good enough in the 1950s

- ✓ *Explicit* about what is going on
- ✗ Obscures algorithm from implementation
- ✗ Not portable
- ✗ Not easy to modify
- ✗ Not succinct

# Programming languages

- Allow writing code to an *abstract* machine model
  - A translator of some kind (perhaps a compiler) transforms this code into something that executes on some hardware
- ⇒ sometimes this “hardware” is a virtual machine (e.g. Python)
- Some virtual machines are “hybrid”: they do just-in-time compilation (e.g. V8 compiler)





# Programming languages

- Microarchitecture just reads an instruction stream
  - Not easy to program complex algorithms in such a “language”. C is arguably quite close
- ⇒ use abstractions leading to high level languages
- Features driven by programming paradigm considerations, domain knowledge, wanting to target particular hardware, ...
  - Compiler or interpreter maps this language onto machine instructions
  - We therefore need a formal specification of the input
- ⇒ languages *define* the syntax and semantics of their input

Functional programming languages don't map directly onto current hardware. A Haskell interpreter (or compiler) thus maps from one paradigm to the other.

## Development environment

- GHC (Glasgow Haskell Compiler) can be used as an interpreter `ghci` and compiler `ghc`
- Available freely from [www.haskell.org/platform](http://www.haskell.org/platform)
- De-facto standard implementation
- Interpreter sufficient for this course

## Standard library

- Ease of use of languages often determined by standard library
- Haskell has a large standard library, and is particularly strong manipulating lists
- We'll redo some of these things for practice purposes

# One slide example

```
filter :: (a -> Bool) -> [a] -> [a]
filter p [] = []
filter p (x:xs)
  | p x      = x : filter p xs
  | otherwise = filter p xs
```

- Higher order
- Polymorphic (works for all types  $a$ )
- Function defined with recursion and *pattern matching*

# Syntax and semantics

## Definition (Syntax)

What are valid sentences (expressions) in a language?

## Definition (Semantics)

What do these valid sentences (expressions) mean?

- Syntax *prescribed* by Haskell language standard
- Semantics of *primitive* code fragments also defined by standard
- Whole program semantics must be constructed by the reader

## Keywords and white space

Certain character sequences have special meaning: *keywords*.

e.g. (Python) **for**, **in**, **with**, **class**, ...

White space is used to separate tokens. Some languages make white space have *meaning*. Haskell and Python are two such.

# PROGRAMMING IN HASKELL



## Chapter 2 - First Steps

# Starting GHCi

The interpreter can be started from the terminal command prompt \$ by simply typing ghci:

```
$ ghci
```

```
GHCi, version X: http://www.haskell.org/ghc/ :? for help
```

```
Prelude>
```

The GHCi prompt > means that the interpreter is now ready to evaluate an expression.

For example, it can be used as a desktop calculator to evaluate simple numeric expressions:

```
> 2+3*4
```

```
14
```

```
> (2+3)*4
```

```
20
```

```
> sqrt (3^2 + 4^2)
```

```
5.0
```

# Haskell Scripts

- As well as the functions in the standard library, you can also define your own functions;
- New functions are defined within a script, a text file comprising a sequence of definitions;
- By convention, Haskell scripts usually have a .hs suffix on their filename. This is not mandatory, but is useful for identification purposes.



# My First Script

When developing a Haskell script, it is useful to keep two windows open, one running an editor for the script, and the other running GHCi.

Start an editor, type in the following two function definitions, and save the script as test.hs:

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

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

Leaving the editor open, in another window start up GHCi with the new script:

```
$ ghci test.hs
```

Now both the standard library and the file test.hs are loaded, and functions from both can be used:

```
> quadruple 10
40

> take (double 2) [1,2,3,4,5,6]
[1,2,3,4]
```

# Naming Requirements

- Function and argument names must begin with a lower-case letter. For example:

myFun

fun1

arg\_2

x'

- By convention, list arguments usually have an s suffix on their name. For example:

xs

ns

nss

# The Layout Rule

In a sequence of definitions, each definition must begin in precisely the same column:

```
a = 10
```

```
b = 20
```

```
c = 30
```



```
a = 10
```

```
  b = 20
```

```
c = 30
```



```
a = 10
```

```
b = 20
```

```
c = 30
```



The layout rule avoids the need for explicit syntax to indicate the grouping of definitions.

```
a = b + c
  where
    b = 1
    c = 2
d = a * 2
```

means

```
a = b + c
  where
    {b = 1;
     c = 2}
d = a * 2
```

implicit grouping

explicit grouping

# Comments

- Semantics of complex code fragments is given implicitly: *you* have to reconstruct it
- Code has to be written correctly for computers
- We can think about how to write it for humans to understand things
- Comments (or literate programming) can help

```
-- Compute the factorial of an integer
fac :: Int -> Int
{- Base case: 0! = 1
   Recursive case: n! = n (n-1)! -}
fac 0 = 1
fac n = n * fac (n - 1)
```

# Building block summary

- Prerequisites: none
- Content
  - Defined syntax and semantics
  - Classified translation of language to executable into interpreted and compiled
  - Familiarity with Haskell whitespace/layout rules
  - Seen function application
  - Seen how to write comments
  - Seen how to run scripts
- Expected learning outcomes
  - student *knows* definition of interpreting and compiling a programming language
  - student can *explain* difference between syntax and semantics
  - student can *explain* whitespace rules in Haskell
  - student can *use* the Haskell interpreter to run small toy problems.
- Self-study
  - Remainder of slides for Chapter 2 (I skipped some)
  - Optional: slides for Chapter 1 (historical background)