Durham University

Session 1: Introduction

COMP2221: Functional programming

Laura Morgenstern

laura.morgenstern@durham.ac.uk

Course organisation: methods

Lectures

- 10 lectures overall
- 2 per week
- Start with brief recap
- Finish with quick quiz

Practicals

- Programming requires practice
- Tackle exercises yourself (before the practical session) first
- Your chance to ask questions and get feedback
- First practicals start on Monday, the 16th of January

$\Rightarrow$ Lecture slides, recordings and exercises are hosted at
`https://comp2221.github.io/fp-website/`

Course organisation: literature

- Course mainly follows Graham Hutton's *Programming in Haskell* (2016)
- Available at the library
- Slides for the first 10 chapters are available at `http://www.cs.nott.ac.uk/~pszgmh/pih.html`

Course organisation: assessment

- By exam
- *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 underlying concepts? What properties does a solution have?
- Past papers available and sample paper will be provided

Course organisation: questions

- Discussion forum for questions:
  `https://github.com/comp2221/fp-website/discussions`
- Happy to take them in live sessions

What is functional programming?

- A programming *paradigm* where the building block of computation is the *application of functions* to arguments.
- Functional programs specify a data-flow to describe *what* computations should proceed
- Algebraic programming style dominated by function application and composition

In contrast, what is imperative programming?

- A programming *paradigm* where the building block of computation is the *modification of stored values*.
- Imperative programs specify a control-flow to describe *how* computations should proceed
- Algorithmic programming style dominated by variable assignments, loops, conditional statements

Effectful vs. pure functions

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

Will `y1 == y2`? How could it not?

Effectful vs. pure functions

```
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"
```

Definition (Side effect)

A side effect is a (hidden) state change that results from a function modifying or relying upon objects external to its parameter list.

Effectful vs. pure functions

Definition (Pure function)

A pure function is a function that takes all its inputs as arguments and produces all its outputs as results.

$\Rightarrow$ A pure function is a function without side effects

What is a functional programming language?

A functional programming language:

- *Supports* and *encourages* a programming style with function application and composition as basic building blocks
- Forbids variable assignment and side effects $\Rightarrow$ "Pure functional"
- Makes *reasoning* about code simpler (for humans and compilers)

Why are C/Java/Python not functional programming languages?

✓ It is indeed *possible* to write in a functional style in these languages...

✗ but they do not enforce it.

✗ Moreover, the language-level support is weak.

✓ In contrast, Haskell is a purely functional (side effect free) language, and built from scratch for functional programming.

Example: computing $n!$

Imperative style

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

Functional style

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

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

Which implementation maps more naturally onto a computer?

Which implementation is more convenient for the programmer?

Excursion: Why high-level 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
- ✗ Obfuscates algorithm from implementation
- ✗ Not portable
- ✗ Not easy to modify
- ✗ Not succint

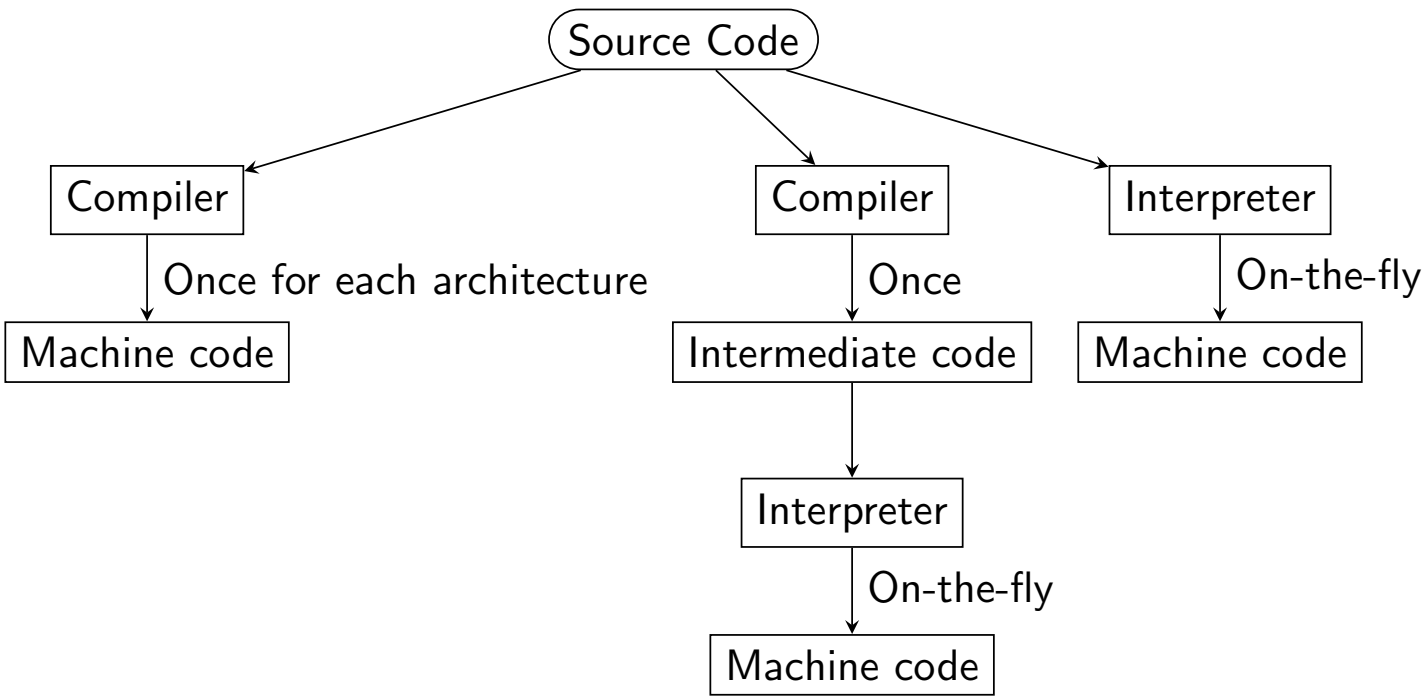
High-level Programming languages

- Allow writing code to an *abstract* machine model
- A translator of some kind (perhaps a compiler) transforms this code into something that executes on a machine

$\Rightarrow$ this machine can either be a physical processor

$\Rightarrow$ a virtual machine (e.g. Java JVM)

- or a "hybrid": they do just-in-time compilation (e.g. Java JIT)



Programming languages

- Microarchitecture just processes an instruction stream
- Not easy to program complex algorithms in such a "language" (C is arguably quite close)

$\Rightarrow$ 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

$\Rightarrow$ languages to *define* the syntax and semantics of their output

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

Haskell taster

Functional style

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

```
-- Compute the factorial of an integer
fac :: Int -> Int
fac 1 = 1
fac n = n * fac (n - 1)
```

- Semantics of complex code fragments is given implicitly: reader has to reconstruct it
- We can think about how to write it for humans to understand
- Comments (or literate programming) can help

Haskell environment

Development environment

- GHC (Glasgow Haskell Compiler) can be used as an interpreter `ghci` and compiler `ghc`
- Available freely from `www.haskell.org/download`
- De-facto standard implementation

Standard library

- Ease of use of languages often determined by standard library
- Haskell has a large standard library, and is particularly strong manipulating lists
- In practicals, you will redo some of its functionalities for practice purposes

Demo time

Summary

Content:

- Defined basic terms; functional style, side effects, pure functions, functional programming language
- Looked at toy problem from both a functional and imperative point of view
- Classified translation of language to executable into interpreted and compiled
- Haskell syntax for function application, naming and layout rules
- List operations of prelude

Self-study:

- Setup your Haskell programming environment, e.g. via `https://comp2221.github.io/fp-website/setup/`
- Tackle exercise 1
  `https://comp2221.github.io/fp-website/exercises/exercise1/`