Session 1: Introduction

COMP2221: Functional programming

Haskell

Lawrence Mitchell*

*lawrence.mitchell@durham.ac.uk

Outline

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

First practicals start this week. 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
```

Assignment

loop

update variable in place.

Functional style

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

n is integer, ≥ 1.

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

No modification of variables in place.

Which implementation maps more naturally onto a computer?

Which implementation is more convenient for the programmer?

Might have different answers.

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;

$\Rightarrow$ 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?

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?

modification of state.

f generates random #s.

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

f takes input from somewhere.

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

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

reasoning about answer is non local.

A functional approach

- Forbid variable assignment and side effects *in the language*. “Pure functional”

✓ Makes *reasoning* about code simpler (for humans and compilers).

✗ A new programming *paradigm*: takes some time to get used to.

C#

Why not C/Java/Python?

✓ 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 language for exploring lots of FP ideas.

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.
- Talk about how functional style is useful in software engineering.
- 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`
- Course will make links with other material/programming languages (C#/C/Python) $\Rightarrow$ seen in other submodules

Logistics: learning

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 ULTRA/course website for details).

this week. In person/hybrid from next week.

Logistics: assessment

open book. in summer.

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: last year's paper is a good guide, a sample paper will also be available.

Style of teaching

- Combination of slides and live coding
- Focus on theoretical underpinnings and concepts applied to design of software

$\Rightarrow$ help to understand where Haskell ideas are adopted elsewhere.

- Not much focus on algorithmic complexity (not all non-CS students have seen it) $\Rightarrow$ 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

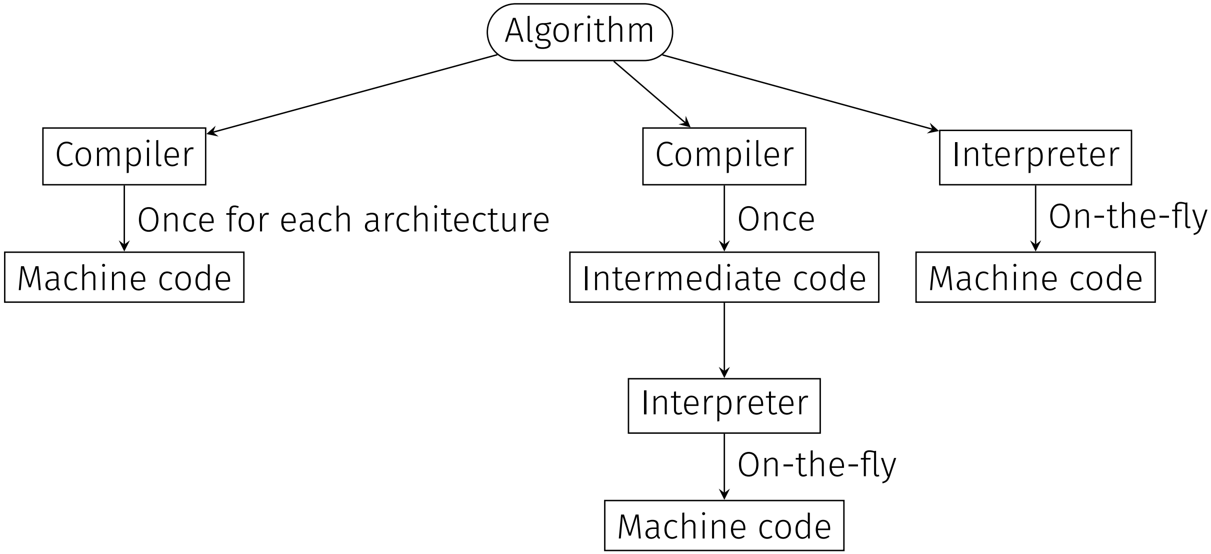
- ✓ *Explicit* about what is going on
- ✗ Obfuscates algorithm from implementation
- ✗ Not portable
- ✗ Not easy to modify
- ✗ Not succint

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 in Chrome)



Programming languages

- Microarchitecture just reads an instruction stream imperative
- Not easy to program complex algorithms in such a "language". C is arguably quite close PDP/11 assembler that thinks it's a programming language

⇒ 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.

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
- 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

Demo time

One slide example

Type signature

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

Definition

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

Polymorphic: works for any concrete type at all.

filter :: (a → Bool) → [a] → [a]

filter takes a predicate function
a list of values
and returns a new list of values
satisfying the predicate.

Definition via pattern matching
"what to do if you see a call
that looks like this".

Base case for empty list

filter p [] = []

Recursive case destru es list

matches a list with at least one entry

filter p (x : xs)

values shared

new list created

Guard expression selects definition based on result of p x

| p x = x : filter p xs
| otherwise = filter p xs

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.

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

```haskell
-- 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
  - Work through the `Lec01.hs` live code to check you understand things.