Important Lessons So Far

– Review –

COMP1100 — Introduction to Programming and Algorithms

Clem Baker-Finch

Australian National University

Semester 1, 2006

About the mid-semester exam:

- How hard was it?
- Will the final be that easy?
- Some advice . . .

About Assignment 2:

- Due Friday 12th May
- **Windows users be warned!**
  - DOS puts both *carriage return* and *line feed* at the end of the line. When you read a line, GHC only looks for line feed, **so the carriage return becomes part of the string!**
  - If you *save* the `Marks.txt` file in a Windows editor, your program will seem to *overwrite* lines of output!
  - **Solution:** Don't edit or save `Marks.txt`! If this problem occurs, either:
    1. Get a fresh copy of `Marks.txt`
    2. Stop using Windows ...

Does everything seem a bit hazy?

Yes? That’s okay — this is complicated and we’ve got all year. . .

(The nature of university education?)

No? Either you’re not trying or you’re kidding yourself.

Can’t See the Woods for the Trees?

This course is ***about programming***.

Fundamental concepts — what programming ***is***.

It is **not** about Haskell or any other programming ***language***.

It is not *utilitarian*. It is not *skills-based*.

But . . . to learn about programming, you need to experiment, so a programming language is a necessary vehicle.

Programming Languages

Programming languages are complicated things.

Like human languages, some are more complicated than others.
Complexity is caused by irregular special cases.

Programming languages seem to have masses of details (special cases) that can obfuscate the basic concepts. Some programming languages are more *orthogonal* (less irregular) than others.

Why Haskell?

- Not so complicated;
- Orthogonal design; not so many details; Few (no?) special cases.
- Express basic concepts more directly and therefore more clearly.

Important Lessons So Far

Mostly, I have tried to focus on some fundamental concepts of programming.

I have also talked about some details and some less important things. Why?

- Because programming *is complicated* and you need a certain amount of stuff to put the basic concepts into practice;

  For example, input-output is difficult in every useful language . . .
  but you can't write a whole program without it.

- Because some students are interested in more than the basics;

  For example, writing programs to draw fractals.

- Because we need some examples to reinforce the important lessons.

What have been the important lessons so far?

Types and Values

What's a **value**? What's a **type**? Why are types important?

Conditionals

Making choices is a necessary part of any non-trivial algorithm.
Different languages may have different ways of *expressing* choices, but the fundamental idea is the same.

Structured Data

Tuples, lists, user-defined (algebraic) data types.

Lists

An extremely important data structure.

Lists are built-in in some languages (e.g. Haskell).
Lists are provided in libraries in some languages (e.g. Java).
Lists are DIY (do it yourself) in some languages.

Recursion

The fundamental technique for expressing repetition in computation.

Repetition is the *essence* of computation.

All other means of expressing repetition (e.g. loops) are special cases — *patterns* of recursion.

Data Directed Design

Program design is largely independent of the implementation language.

Bottom Up and Top Down design focus on the **problem**. When we are writing the implementation we work with the same ideas, but express them differently depending on the language.

Algebraic Data Types

Not many programming languages have features corresponding to Haskell's algebraic data types, but they can be represented by classes, objects and inheritance in Object-Oriented languages (like Java).

Where Now? Java!

Why?

Java is an **object-oriented programming language**.

(Flavour of the month?)

Java is a widely used programming language. (Why? Libraries and hype.)

How?

Mostly, we will repeat the important lessons we have seen in Haskell.

The emphasis is on the **concepts** rather than the individual language.

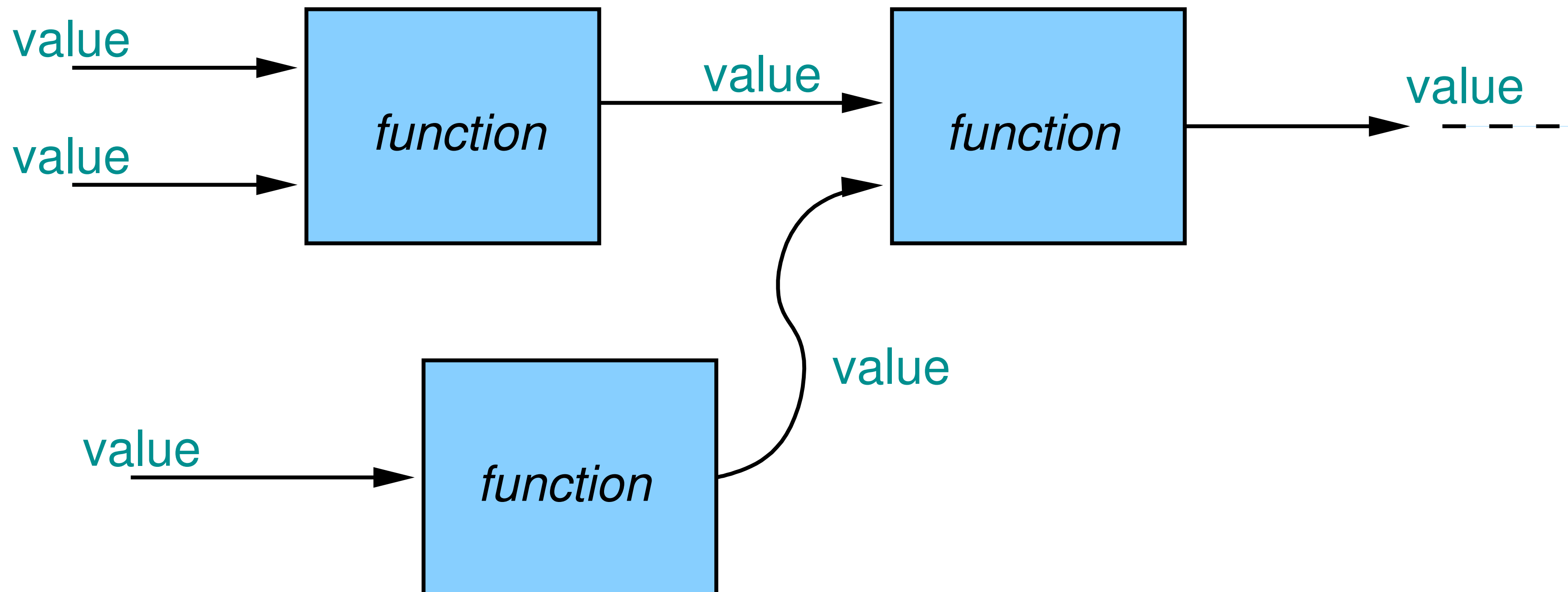
Down-side?

I'm afraid so — another bunch of details to overcome.

The Functional Model of Computation (Haskell)

- *Everything* is a value
- Every value has an explicit *type*
- We can give *names* to values
- *Functions* take values as arguments and return *new values* as results
- Computations are defined by *composing functions*

The Functional Model of Computation



The Imperative Model of Computation (Java)

- There is a **single store** consisting of a collection of **locations** which can *contain values*
- We can give *names* to locations and say what *types of values* they may contain
- Computations are defined by a **sequence of commands**
- Each command may **change the store** in some way

Fortunately for us, imperative programs can have *functions*, too . . .

The Imperative Model of Computation