

Modules and Assignment 1

COMP1100 — Introduction to Programming and Algorithms

Clem Baker-Finch

Australian National University

Semester 1, 2006

Announcements

- Assignment 1 is now available.
Due date: 12 noon, Monday 3rd April 2006.
See the COMP1100 web site.
- Reminder — Drop-in lab sessions.
For help with course material, practical exercises, assignments, etc.
 - Monday 10-11am in N116
 - Thursday 10-11am in N116

Modules in Haskell

Programs can grow to hundreds of thousands of lines of code.

How can we manage such complexity?

ABSTRACTION!

Decompose the system into smaller *strongly related modules*.

Aim: details of each module are only relevant *within* that module.

Separation of concerns: *HOW* versus *WHAT*.

All(?) programming languages provide a mechanism for splitting a program into separate **modules**.

Benefits of Good Modularisation

- Structuring tool to simplify program design
- Simultaneous development of modules by multiple teams
- *Parts of the program can be understood in isolation*
- Easier to change parts of the program without affecting others
- Code can be re-used — e.g. the Haskell library modules

Modular Design

The fundamental activity in the software design process. **But how?**

- **Each module should correspond to a single abstraction**
- Each module should have a clearly defined purpose
- It should be possible to understand each module in isolation
- Modules should be easy to test individually

In most cases, modules should correspond to an ***abstract data type***.

That is, **a type that models some aspect of the *problem domain***.

Modules in Haskell

A module in Haskell is a script where the first line of code is:

```
module Pixel where
```

The name of this module is `Pixel`. Module names begin with a capital letter.

In general, modules should be in files with the same name.

The `Pixel` module should be in a file called `Pixel.hs`.

To use the definitions of one module in another, it must be imported:

```
module PPM where
import Pixel
```

Example – Assignment 1

Assignment 1: build some computer image manipulation tools:
rotate, flip, desaturate . . .

Step 1: Analyse the Problem

How are images represented on computers?

Computers represent images by a grid of coloured dots, called pixels.

The pixels are so small that, when displayed on a computer screen, they appear to merge into a smooth image.

Abstract Data Types

Immediately from a simple problem analysis we can infer:

- The problem involves a **type** of things called **Pixel**
- The problem involves a **type** of things called **Image**
- Pixels have a type of things called **Colour**

How do we represent these abstract data types in Haskell?

Colours

There are many ways to represent colours. A simple one is **RGB**.

To specify a colour, you say how much **red**, how much **green**, and how much **blue** is in the colour, and the usual range is 0 . . . 255 for each value.

White is 0 red, 0 green, 0 blue.

Black is 255, 255, 255.

Red is 255, 0, 0.

Pixel Module

To represent the pixel abstract data type, what goes in module `Pixel`?

- A type for representing the values of pixel type
- All the core functions operating on pixels

Representing Images

A “grid” of pixels ... ?

In Haskell:

Each *row* can be a *list* of pixels.

Each *image* can be a *list* of *rows*.

PPM File Format

As well as our **internal** Haskell image representation, we need to deal with the **external** representation of images in files.

There are many file formats (jpeg, gif, eps, psd, pdf, ...)

We will use a very (*very*) simple one: **P**ortable **P**ixel **M**ap (.ppm)

PPM Module

The `PPM module` will include the (internal) representation of images.

Since images are made from pixels, `module PPM` will `import Pixel`

Core operations on images include:

- **decoding** PPM file format to values of type `Image`
- **encoding** values of type `Image` to PPM file format

Image Transformation Modules

Each image transformation tool will be developed in a separate module.

User Interaction Module

Another module (`module Main`) contains the user interaction:
get an input file; apply the transformation; produce an output file.

(A whole program at last!)

Test Harness

`module TestBed` allows us to test our image transformation tools more interactively and more conveniently than the `Main` module allows ...