Chapter 8

Simple Haskell

Language Features

> This chapter introduces the basic language features:

  – expressions
  – functions
  – types

> But first, there is some jargon

Declarative Languages

- Languages like C and Java are called ***imperative*** because they tell the computer how to do the job
- The idea of a ***declarative*** language is
  - a program describes ***what*** is to be done, rather than ***how*** to do it
  - it concentrates on the ***logic*** of the problem and not on the ***control flow*** which is left to the computer
  - programs are a lot less verbose, and a lot more reliable (there are fewer places for bugs to hide!)
  - you can even prove that a program is correct

Declarative Examples

- You will meet lots of languages, mostly special purpose, which are declarative or partly declarative:

Verilog	HTML
Maple	XML
Fril	SQL
Prolog	BNF
Python	JSP

- Haskell will give you an introduction to the ideas which will make these languages easier to pick up

Functional Languages

- The most important declarative types are ***logic*** languages (like Prolog) and ***functional*** languages
- Haskell is a functional language, which just means that programs are made out of functions
- Haskell is almost unique among the declarative languages in being a general purpose language rather than just a special purpose one
- You can think of it as getting rid of the bad features of other languages, and boosting the good features

GHC/GHCI

- GHC stands for the Glasgow Haskell Compiler
- In the lab, you can use **`ghci`** (interactive **`ghc`**)

```
ghci
Prelude> 2+2
4
Prelude> :quit       (or :q for short)
```

- **`Prelude`** is the standard library module, available in every program (like **`java.lang.*`**)
- Later, if you want, you can compile programs with **`ghc`** but that isn't needed for the assignments

Interactive output

> There is a difference between interactive output and printed output

```
ghci
Prelude> "one " ++ "two"
"one two"
Prelude> putStrLn("one " ++ "two")
one two
Prelude> :q
```

> In the first example, the *value* of the result is displayed, as a string, using program-style notation

> In the second, Haskell is asked to *print* the string, i.e. send the characters one by one to the screen

Simplified Haskell

- In this unit, all or most of the time, we are going to avoid Haskell's I/O which forms a different sublanguage of Haskell
- In assignments, your code will be tested directly, or a main program with I/O will be provided
- It is no harder to learn than I/O in other languages, but it distracts and confuses, and hasn't got much to do with the essence of functional programming
- In any case, Haskell is often used for prototyping using an interactive tool like `ghci` without I/O

Expressions

- You can use **ghci** as a calculator:

```
ghci
Prelude> 2^10
1024
Prelude> 2^100
1267650600228229401496703205376
Prelude> sum [1..100]
5050
Prelude> let square(n) = n*n
Prelude> square(16)
256
Prelude> :q
```

- It handles any precision, any types, any operations (the red brackets are redundant - you can omit them)

Side Effects

- One of the bad features of languages like C and Java is side effects
- A side effect is something that a program or "function" or procedure or method does beyond just returning a result
- It means that procedures are not self-contained, not predictable, and may have unwanted consequences

Large Scale Side Effects

> The classic Web example of side effects is:
  - while surfing, you find **`Armageddon.exe`**
  - it looks interesting, so you download and run it
  - it finds your most private information stored on your computer, and broadcasts it on the Web
  - then it wipes your disk clean

> Those are unwanted side effects!

> JavaScript programs and Java applets have to be run in sandboxes to make sure they are safe

Medium Scale Side Effects

- Software development is supposed to be about producing bullet-proof program fragments
- How can you trust a program fragment when it might:
  - change global variables
  - update other structures or objects
  - do some input or output
- Haskell is a way of investigating how much can be achieved with ultra-safe program fragments which are totally self-contained and cannot do any of these things

Small Scale Side Effects

> On the smallest scale, think about variables

> If you have a variable **`count`**, then its value changes over time by assignment (e.g. **`count = count + 1`**)

> One of the big problems with *debugging*, or trying to gain *confidence* in some code, or *proving* code correct is that different versions of **`count`** are confusing

> There isn't just one thing called **`count`**, there is a whole series at different times $\mathtt{count}_0$, $\mathtt{count}_1$, $\mathtt{count}_2$, $\mathtt{count}_3$, $\mathtt{count}_4$, ...

> Can a language be built *without* assignment?

Pure Functions

Pay attention! You have been warned!

- Haskell functions have no side effects, no input/output, no variables, no assignment
- A Haskell function returns a result
- It does nothing else whatsoever, just returns a result
- These are called ***pure functions***
- From now on in this course, a function means a pure function, and what C calls a "function" will be called a procedure or method

Scalable Functions

- A function like **`Character.toUpperCase`** in Java or **`toupper`** in C is obvious enough, taking a character as argument and returning a new character as result
- The Haskell function **`toUpper`** in the Char module does exactly the same thing
- But suppose you think of an entire program as a function from its input to its output
- Then you have a big function, which you can build up from medium sized functions, which you can build up from smaller functions etc., like Lego

Storing Functions

- So far, when using **ghci**, it remembers nothing the next time you use it
- To write programs, you store function definitions in files, and then try them out using **ghci**

Square.hs

```
-- Square an integer
square :: Integer -> Integer
square(n) = n*n
```

```
ghci Square.hs
*Main> square(42)
1764
*Main> :q
```

Comments

- A function should have an introductory comment
- A one-line comment starts with `--`

```
-- Square an integer
```

- Multi-line comments are for describing complex functions, or whole modules, or for commenting out
- They have the form `{- ... -}` and they nest

```
{- Sort records by surname and then by
first name if the surnames are equal -}
```

Declarations

- A function should be declared, which means giving its argument types and result type

```
square :: Integer -> Integer
```

- This is separated out from the function definition, unlike Java
- The sign `::` means "has type" and `X -> Y` means "function taking an `X` and returning a `Y`"

Definitions

- A simple function definition has an obvious form

```
square(n) = n*n
```

- The result is an expression, not a block, because a function doesn't do one thing after another, in fact it doesn't ***do*** anything at all, it just returns a result
- This requires a radical change of thinking about how to write programs (no variables, no assignment, no sequencing, no loops, so what ***can*** you use?)

Decisions

> Haskell does have testing with **`if..then..else..`**

```
ghci
Prelude> let n = 3
Prelude> if n>0 then n else -n
3
Prelude> let n = -3
Prelude> if n>0 then n else -n
3
Prelude> let size n = if n>0 then n else -n
Prelude> size 3 + size(-3)
6
Prelude> :q
```

If-Expressions

- The **then** and **else** clauses are expressions, not blocks

```
if n>0 then n else -n
```

- It means "if .. then the result is n else.."
- This is like the Java expression **n>0 ? n : -n**
- Again, you don't tell Haskell to *do* something, you just write an expression for the result

Brackets 1

Pay attention!   You have been warned!

- Haskell's bracket convention is unfamiliar, and therefore potentially confusing
- So, it needs to be studied carefully
- You are strongly recommended to type in all the examples yourself, including the wrong ones, and look at the bad answers or error messages that you get

Brackets 2

- The convention used in C, Java etc. is that brackets are used for two purposes:
  - grouping, e.g. **`x*(y+z)`**
  - function calls, e.g. **`f(x,y)`**
- The Haskell convention is simpler, more logical, agrees with Lisp-like languages since the 1950s, and with what has been done in mathematics for millennia
- Brackets are used for one purpose:
  - grouping, e.g. **`x*(y+z)`**

Brackets 3

- That means brackets round function calls are not always necessary:

`square(n)`	brackets unnecessary
`square n`	just as good
`square(n+1)`	brackets still needed
`square n+1`	means `square(n) + 1`
`square(-n)`	brackets still needed
`square -n`	means subtract `n` from `square`

- Function application binds tighter than any operator (same as in every language, but easier to get confused)

Brackets 4

- Suppose you see this, in any language

```
size * n+1
```

- You know that `size*n` happens first, then `+1`
- That's because you know compilers ignore spacing, and because you have made the mistake a couple of times
- In Haskell, suppose you see

```
square n+1
```

- You will know that `square n` happens first because the compiler ignores spacing, or at least you will after you have made the mistake a couple of times

Brackets 5

- What about multiple arguments?

  `div m n`

- The function **`div`** is a standard one which does integer division discarding the remainder (the **`/`** operator is reserved for exact division)
- Multiple arguments have no brackets ***and*** no commas

`div(13,2)`	doesn't work!
`div (n+1) 2`	brackets needed
`div n (m+1)`	brackets needed
`div (n+1) (m+1)`	brackets needed twice

Brackets 6

- When you aren't used to it, it sometimes looks as though the brackets are in the wrong place

  `square (square n)`

- There are often brackets round a call **`(f x)`** instead of around the argument **`f(x)`**
- But it is perfectly logical, because Haskell has only one rule:
  - use brackets for grouping, not for calling

Example

- Here is a well known problem (the Collatz problem)
- Change a number n (e.g. 25) according to these rules:
  - If n is odd, multiply by three and add one
  - If n is even, divide by two
  - Repeat, and stop if n becomes 1
- The question is do you always reach 1 and how long does it take? Mathematicians don't know yet
- All we have to do is write a function **`next`** which finds the next number

Library Functions

- To solve the problem, we need a couple of functions from the standard **`Prelude`** library
- Browsing the documentation shows **`div`** which we've seen, and **`odd`** which tests for an odd number

```
ghci
Prelude> div 27 2
13
Prelude> odd 3
True
Prelude> odd 4
False
Prelude> :q
```

Answer

> The answer to the problem is the function next

Next.hs

```
-- Next number in Collatz problem
next :: Integer -> Integer
next n =
  if n==1 then error "stop" else
  if odd n then 3*n+1 else div n 2
```

```
ghci Next.hs
*Main> next 25
76
*Main> iterate next 25
[25,76,38,19,58,29,88,44,22,11,34,17,52,26,
13,40,20,10,5,16,8,4,2,1,*** Exception: stop
*Main> :q
```

Errors

Pay attention!   You have been warned!

> You ***cannot*** use print statements in Haskell, because there is no sequencing and no I/O inside functions

> But you can throw errors

```
... error "stop" ...
```

> There is no (easy) way to catch them, so they are really errors (though Haskell calls them exceptions)

> This is ***really useful*** for debugging, because it tells you whether a point in the execution is reached or not, and you can print out the current value of something

Programming by Evolution

> Some people like Haskell or find it easy, and some don't, and it is interesting to try to work out why

> Apart from being 'brainwashed' by conventional languages, one of the reasons Haskell can seem difficult (and one of the reasons we teach it) is that it is difficult to ***program by evolution*** in Haskell

> Programming by evolution means writing code a line at a time and fiddling about with it endlessly until the program finally does what you want

> The problem with ***evolution*** as a programming technique is that it takes ages (***eons)***

Programming by Design

- Haskell's compactness means you can't write a line of code which "just messes about a bit"
- You are forced to design programs 'properly', which means e.g. splitting problems into smaller problems
- For example, look at the first Haskell coursework which is to write a program to play MasterMind
- The problem "play Mastermind" is far too difficult, but the assignment breaks it down for you (in the way you should learn to do for yourself) into ten subproblems, each of which is relatively easy to solve (or if not, break them down again)