15 Imperative Programming

[Reading: Thompson Appendix A]

Functional programming languages (like Haskell) are based on *evaluation of expressions*.

Imperative programming languages (like Java) are based on sequences of commands to a machine to *change its state*.

They are quite different styles of programming. Most commonly used languages are the imperative kind.

Why did we start with a functional language? Recall from section 1 of the lecture notes:

- It is **easier to get started** writing Haskell programs than Java programs.
  The simplest Haskell programs can be trivial. The simplest Java program requires you to know about I/O, libraries and many syntactic details. They also need to be compiled and run before testing. The Hugs system works more like a calculator.

- It is **easier to understand** how Haskell programs *execute* — they evaluate expressions.
  To understand Java programs you need to understand the underlying "machine model."

  Haskell is higher-level than Java. That is, it is oriented more toward humans than the computer.

- **Abstraction** and **generalisation** are clearer in Haskell than in imperative languages.

- Problem-oriented **data structures** are easier to define in Haskell than most imperative languages. (Abstraction again.)

- What you learn in this section *will* carry over to the second programming part.

Transfer of knowledge

The craft of writing programs depends on some common standard skills.

These skills apply irrespective of the particular language used.

For example:

- General problem solving and problem analysis.
- Expressing solutions as algorithms.
- The program development methodology applies for all languages.
- The concepts of problem-domain data types and abstract data types apply for all languages.

Functional programmers become better imperative programmers!

15.1 What is Imperative Programming?

To begin, we need some conceptual *machine model*. It will be based on a simple version of the usual "Von Neumann computer architecture."

Machine model

The simplest machine model is based on the notion of a ***store***.

A store consists of many ***locations*** which are containers capable of holding a *data value*.

In our programs we can give ***names*** to some of those locations, and then refer to them by those names.

This is done by means of *declarations*. For example, in Java:

```
int x;
```

sets aside a location which we can refer to by the name `x`. The location is able to contain a data value of type `int`.

Warning:
Names of locations are usually called "variables." They are *quite different* to variables in mathematics (and Haskell).

Imperative programs

An imperative program is a ***sequence of commands*** that *change the store*. Each command (also called *statements*) is an action affecting the store.

The fundamental action is to *put a value into a location*.

This is done with an ***assignment statement***. In Java:

```
x = 1;
```

has the effect of putting the value `1` into the location named `x`.

Warning: the "=" sign is inappropriate and misleading. The assignment statement does *not* mean that `x` is now equal to `1`.

You should read it as

"the value in the location named x becomes 1"

or more briefly (but less correctly) as

"x becomes 1".

Some other languages write `x := 1` instead.

Sequences

The idea of a sequence of commands merely putting values into locations seems very limited.

The algorithmic power comes from the fact that the sequencing can include ***repetitions*** and ***choices*** to alter the "flow of control."

Example:
It is possible to calculate the square of a number n by adding all the odd numbers less than $2n$. In Haskell we could define the function:

```
> square :: Int -> Int
> square n = sum [1,3..n+n]
```

In Java:

```
static int sqr (int n)
{
 int oddnum;
 int result;
 oddnum = 1;
 result = 0;
 while (oddnum < n+n) {
    result = result + oddnum;
    oddnum = oddnum + 2;
    }
 return result;
} // end sqr
```

Look at the Java function, line by line. (It is not a complete program, just a function definition.)

```
static int sqr (int n)
{
 . . . .
} // end sqr
```

This introduces the name of the function (`sqr`) and its type. The argument is `n` and its type is `int`. The type of the result is `int`, as indicated by `int sqr`. The body of the function is delimited by the braces "`{`" and "`}`" and "`// end sqr`" is a comment.

```
 int oddnum;
 int result;
```

These declarations each set aside a location that can hold an `int` value. The locations are given names `oddnum` and `result`. The storage allocation occurs each time the function is called. When the function call is completed, it is de-allocated.

```
 oddnum = 1;
 result = 0;
```

These two assignments *initialise* the two locations. The first odd number is 1 and the result is 0 (so far).

```
while (oddnum < n+n) {
    result = result + oddnum;
    oddnum = oddnum + 2;
    }
```

This is the core of the algorithm. Look at the sub-components first.

```
    result = result + oddnum;
```

This assignment statement shows the danger of interpreting “=” as equality: it is not possible for `result` to be equal to `result + oddnum` !

A key concept in the understanding of assignment statements is that we interpret variables on the *left* of the = differently from variables on the *right*.

In this case, `result` on the *left* is used to indicate the location where the value is to be stored.

`result` on the *right* is used to indicate the the value that is presently in the location with that name.

The effect is thus to add the value in `oddnum` to the value in `result` and store that sum in the location called `result`.

```
    oddnum = oddnum + 2;
```

Similarly, this adds 2 to the value in location called `oddnum`. In other words, advance to the next odd number.

```
while (oddnum < n+n) {
    result = result + oddnum;
    oddnum = oddnum + 2;
    }
```

This is a *repetitive* statement. It first tests whether `oddnum` is less than `n`. If not, that is the end of the whole while-statement.

If `(oddnum < n+n)` is true, then the command that is the *body* of the while-statement is executed, then we go back and repeat the test.

In other words, while the condition is true, repeatedly execute the body of the loop.

The body is delimited by braces. It consists of a sequence of two assignment statements which are executed in the order they are written.

```
return result;
```

Finally, the `return` command specifies the value that is returned by the function. Since `result` contains the sum of the odd integers, that is the value we want.

Execution vs. Evaluation

To understand a functional program we need to understand the *evaluation of expressions*.

For `square 5` using the Haskell function above, we can simply write down the list `[1,3..10]`, that is, `[1,3,5,7,9]` and then `sum` the elements.

To understand an imperative program is generally more difficult.

We need to trace through the *execution of the sequence of commands*, based on the *machine model*.

For `sqr (5)` using the Java function above, we need to keep track of the contents of three locations called `n`, `oddnum` and `result`. We also need to keep track of the point in the sequence of commands that the execution has reached.

Exercise:
Trace the execution of the function call `sqr (5)`.

Here is a slightly different version of the same example. Rather than n^2 being the sum of the odd numbers less than $2n$, say that n^2 is the sum of the first n odd integers. Obviously this is essentially the same, but leads to slightly different definitions.

In Haskell:

```
> square :: Int -> Int
> square n = sum (take n [1,3..])
```

In Java:

```
static int sqr (int n)
{
 int oddnum;
 int result;
 oddnum = 1;
 result = 0;
 for (int count = 1; count <= n; count++){
    result = result + oddnum;
    oddnum = oddnum + 2;
    }
 return result;
} // end sqr
```

15.2 Comparison of Functional and Imperative Programming

This reflects my personal bias.
Nevertheless it's true!

Closeness to the problem domain

A functional program is generally a *description* of a solution to a problem.

An imperative program is a sequence of instructions to a computer. While the *outcome* may still be a satisfactory solution, it is much further away from the problem domain. That means it will be harder to understand and possibly less reliable.

Emphasis on data or emphasis on control

The main concern in an imperative program is *storage management* (since it is based on a machine model and assignment).

Another major concern is the *flow of control* — repetitions, choices, getting the next piece of data, etc.

(Exercise: Give a convincing argument that the for-loop condition in the java program above should be `count <= n` and not `count < n`.)

In contrast, functional programs are “data-directed”. Algorithms are based on generating data and processing it.

For example, in the `square` function above the first step is to generate the odd numbers less than `n+n`. The second step is to sum them.

For example, in the supermarket docket exercise, the first step is to convert a list of bar codes to a list of names and prices. The next step is to convert that list into a list of docket lines, and so on. The major steps in data processing define the algorithm.

As a consequence, functional programs are easier to read and easier to write.

Proofs

In Haskell, as well as being a program, a function definition is a logical equation describing a property of the function. Using such definitions we can prove other properties of the program. (You did this in a section of Engineering Mathematics 1.)

The key point is that mathematical proofs can be carried out *in the language itself*.

This is not possible for imperative programs. While we can prove properties of imperative programs, it must be in another language that describes the *semantics* (i.e. meaning) of the program.

The machine-model basis for imperative languages makes their semantics extremely complex. The mathematics is far more complex and hence the proofs are either too difficult or too unreliable.

As a comparison, the (mathematical) semantics of all of Haskell could be fully described in much less than 100 pages. The complete (mathematical) semantics of Java would be more in the order of a city telephone directory.

Syntax

While the cleanness and simplicity of the syntactic structure of languages varies quite a lot, modern functional languages are generally less cluttered.

A simple demonstration is the “Hello World” program.

In Haskell we need only write:

```
> module Main where
> main ::IO()
> main = putStr "Hello world"
```

(Even then, the first two lines are optional.)

In Java, the following program is about as clean and brief as possible:

```
import java.io.*;

public class hello
{
  public static void main (String[] args)
  {
   System.out.println ("Hello world");
  } // end main
} // hello
```

15.3 More Example Programs

Here is a Java function that tests whether a given number is a perfect square:

```
static boolean isSquare (int x)
{
 int n = 1;
 int n_sq;

 n_sq = sqr (n);
 while (n_sq < x) {
   n = n+1;
   n_sq = sqr (n);
   }
 return (n_sq == x);
} // end isSquare
```

The function sets aside locations `n` and `n_sq`, running `n` up from 1 and calculating n^2 in `n_sq`. The loop keeps going as long as n^2 is less than `x`, the number being tested. The result is true if `x` = n^2.

Notice that *flow of control* dominates the program.

The values are generated one at a time.

Here is a Haskell function that tests whether a given number is a perfect square:

```
> perfSquare :: Int -> Bool
> perfSquare x = x `ordElem` squares
>   where
>   squares = map (^2) [0..]
```

It uses a function that you developed in the week 6 practical class:

```
> ordElem :: Ord a => a -> [a] -> Bool
> x `ordElem` [] = False
> x `ordElem` (y:ys)
>   | x == y     = True
>   | x < y      = False
>   | otherwise  = x `ordElem` ys
```

The `perfSquare` function design is *data-directed*. The first step is to generate a list of all the square numbers. The second step is to see if `x` is in that list (being careful to stop looking when it is a hopeless cause).

The three Java functions `sqr`, `anothersqr` and `isSquare` cannot stand alone. They are all included in a test class `square.java`.

Here is a (complete) Java program that reads in a sequence of characters representing grades and calculates a grade point average.

```
// Calculate a grade point average.
// Based on Deitel & Deitel 2.9

import java.io.*;

public class gpa {
  public static void main (String args[]) throws IOException
  {
    float average;
    int grade, count, total;

    total = 0;
    count = 0;
    System.out.print ("Enter letter grade, Z to end: ");
    System.out.flush();
    grade = System.in.read();

    while (grade != 'Z') {
      if (grade == 'A')
        total = total + 4;
      else if (grade == 'B')
        total = total + 3;
      else if (grade == 'C')
        total = total + 2;
      else if (grade == 'D')
        total = total + 1;
      else if (grade == 'F')
        total = total + 0;
```

```
      System.in.skip (1);
      count = count + 1;
      System.out.print ("Enter letter grade, Z to end: ");
      System.out.flush();
      grade = System.in.read();
      }
    if (count != 0) {
      average = (float) total / count;
      System.out.println ("Grade point average is " + average
      }
    else
      System.out.println ("No grades entered.");
  } // end main
} // end gpa
```

The following Haskell function is not really a fair comparison, since it is not an interactive IO function.

```
> gradePointAverage :: [Char] -> Float
> gradePointAverage grades
>   = sum (map gradeValue grades) /
>     fromIntegral (length grades)

> gradeValue :: Char -> Float
> gradeValue 'A' = 4
> gradeValue 'B' = 3
> gradeValue 'C' = 2
> gradeValue 'D' = 1
> gradeValue 'F' = 0
```

15.4 *Design* of Imperative Programs

The overall design process is the same, no matter what language is being used:

1. Understand the Problem

2. Identify the Objects

3. Identify the Basic Operations on Objects

4. Choose Representations of the Objects

5. Implement the Basic Operations on Objects

6. Factor the Process into Manageable Parts

Obviously, the *choice of representation of the objects* will depend on the facilities offered by the particular language. For example, lists are fundamental in functional languages, but are not in most imperative languages.

Still, they are the *same ADTs*.

Their representation may be more complex than in Haskell, since it may be less direct. In Haskell, we can directly define recursive types like trees. In most language we need to *construct* recursive types from more primitive types, links and so on.

The *implementation of the basic operations on the objects* will depend on the language of implementation.

The point in the design process where the difference between the functional and imperative approaches is greatest is *factoring the process into manageable parts*.

The difference comes from the point already made: design of functional programs is *data directed*, design of imperative programs is *control-flow directed*.

Example: Supermarket Bill

Designing for a *functional* language, the process was broken down into the composition (a sequence) of functions including the following:

1. Take the list of bar codes, look each one up in the database and produce a list of name, price pairs. (`makeBill`)
2. Take the list of name, price pairs and produce a line of the docket for each pair. (`formatLines`)
3. Take the list of name, price pairs and calculate the total for the bill. (`makeTotal`)
4. Combine those lines, the total and a heading to produce the printed docket. (`printBill`)

Notice how the steps were defined by the main data processing steps.

Designing for an *imperative* language, the emphasis switches to flow of control and *processing the data elements one at a time*.

For each bar code, we look it up in the database to get the corresponding name and price. Then we format and print the coresponding line of the docket. At the same time we keep a running total of the prices. This is repeated until all the bar codes have been processed.

In "*pseudo-code*" we have:

```
total ← 0
while more bar codes
    get bar code
    look up bar code for name and price
    format and print line
    total ← total + price
endwhile
```

Notice how each bar code is fully processed to give the docket line and running total *before* the next bar code is fetched.

That leaves the heading and the total to be added to the docket. Since the docket is being produced line by line as we go through the bar codes, we must take into account the flow of control when adding those components.

The heading must be printed *before starting to process the bar codes*.

The total line must be printed *after* processing the bar codes.

```
print docket heading
total ← 0
while more bar codes
    get bar code
    look up bar code for name and price
    format and print line
    total ← total + price
endwhile
print total line
```