8.5 Define a strategy which looks at the frequencies that the opponent has played the three choices. If you assume they are making random plays, then why not predict they are about to play the least frequent, and use the trick from the previous exercise to make a choice.

8.6 [Harder] Go to the World RPS website – see the box at the end of this section – to find out more strategies, and try to implement them.

8.7 [Harder] Define a function

```
alternate :: Strategy -> Strategy -> Strategy
```

so that the strategy given by

```
alternate str1 str2
```

is to combine the two strategies `str1` and `str2`, using them alternately.

8.8 [Hard] Can you write a function which will *analyse* an arbitrary strategy to work out what it does? You will need to think about how to represent this result: is it a string describing what the strategy does, or something a bit less informal?

8.9 [Hard] Can you use the facilities of QuickCheck – described in more detail in Section 19.6 – to help to analyse what an arbitrary strategy does?

More information about Rock – Paper – Scissors

In fact the choice of the first move in defining `echo` is not completely arbitrary: men are most likely to choose `Rock` first. This and lots more information about Rock – Paper – Scissors can be found at the World RPS Society website:

`http://www.worldrps.com/`

8.2 Why is I/O an issue?

A functional program consists of a number of definitions, such as

```
val :: Integer
val = 42

function :: Integer -> Integer
function n = val + n
```

The effect of these definitions is to associate a fixed value with each name; in the case of `val` the value is an integer and in the case of `function` it is a function from integers to integers. How is an input or an output action to fit into this model?

One approach, taken in Standard ML (Milner et al. 1997) and F# (Smith 2009), for instance, is to include operations like

```
inputInt :: Integer
```

whose effect is to read an integer from the input; the value read in becomes the value given to `inputInt`. Each time `inputInt` is evaluated it will be given a new value, and so it is not a fixed integer value as it ought to be according to our original model.

Allowing this operation into our language may not seem to cause too big a problem, but examining the example of

```
inputDiff = inputInt - inputInt                    (inputDiff)
```

shows how it has two important consequences for our model of functional programming.

- Suppose that the first item input is `4`, and that the next is `3`. Depending upon the order in which the arguments to '`-`' are evaluated, the value of `inputDiff` will be either `1` or `-1`.
- More seriously, `(inputDiff.1)` breaks the model of reasoning which we have used. Up to now we would have thought that subtracting a value from itself would have given a result of `0`, but that is *not* the case here.

  The reason for this is precisely that the meaning of an expression is no longer determined by looking only at the meanings of its parts, since we cannot give a meaning to `inputInt` without knowing where it occurs in a program; as we saw in the previous point, the first and second occurrences of `inputInt` in `inputDiff` will generally have different values.

As the second point shows, if we take this approach then it will be substantially more difficult to understand the meaning of any program. This is because *any* definition in a program may be affected by the presence of the I/O operations. An example is the function

```
funny :: Integer -> Integer
funny n = inputInt + n
```

from whose definition we can see the dependence on I/O, but potentially any function may be affected in a similar way.

Because of this, I/O proved to be a thorny issue for functional programmers for some considerable time, and there have been a number of attempts to find the right model for I/O – indeed, earlier versions of Haskell included two of these. An illuminating history and overview of functional I/O is given in Gordon (1994).

In this chapter we introduce the basics of I/O using the `IO` types; Chapter 18 describes the *monadic* approach to programming which underlies I/O and other forms of interaction with the 'world outside'.

8.3 The basics of input/output

In thinking about input/output or **I/O** it makes more sense to think of *actions happening in sequence*. For instance, first some input might be read, and then on the basis of that some further input might be read, or output might be produced.

Haskell provides the types `IO a` of **I/O actions** of type `a` or **I/O programs** of type `a`. An object belonging to `IO a` is a *program* which will do some I/O and then

return a value of type `a`. Built into Haskell are some primitive I/O programs, as well as a mechanism to sequence these I/O programs.

One way of looking at the `IO a` types is that they provide a small *imperative programming language* for writing I/O programs on top of Haskell, without compromising the functional model of Haskell itself.[1]

We start by looking at the basic `IO` capabilities built into the standard prelude, and then we look at a whole lot of examples to see how to put these components together using the `do` notation. In the next section we look at how to write 'functional loops' to form more complex I/O programs.

Reading input

The operation which reads a line of text from the standard input does some I/O and returns a `String` which is the line just read. According to the explanation above, this should be an object of type `IO String`, and indeed, the built-in function

```
getLine :: IO String
```

reads a line from the standard input. In a similar way,

```
getChar :: IO Char
```

will read a single character from the input.

The one-element type

Haskell contains the type `()`, which contains one element only. This element is also written `()`. A value of this type can convey no useful information and so the type is not often used. However, it *is* useful in performing `IO`, as there are cases of IO programs whose only significance is their I/O actions and not the results they return. Programs of that sort will have type

```
IO ()
```

and they will return the value `()` as their result.

The `Main` module and the `main` program

If we compile a Haskell project using GHC, the Glasgow Haskell Compiler, then this produces executable program which runs the function

```
main :: IO t
```

for some type `t`: often this is `()`, so that

```
main :: IO ()
```

[1] The language is unusual in that it is *single assignment*, like Erlang (Armstrong 2007; Cesarini and Thompson 2009); we'll explain this when we discuss the examples.

By default, this `main` program is expected to be in the `Main` module, but it can be given in any module from the project.

Writing `Strings`

The operation of writing the string `"Hello, World!"` will be an object which performs some I/O, but which has nothing of significance to pass back to the program. It is therefore of type `IO ()`.

The general operation to print a text string will be a **function** which takes the string to be written, and gives back the I/O object which writes that string:

```
putStr :: String -> IO ()
```

and using this we can write our 'hello, world' program.

```
helloWorld :: IO ()
helloWorld = putStr "Hello, World!"
```

Using `putStr` we can define a function to write a line of output.

```
putStrLn :: String -> IO ()
putStrLn = putStr . (++ "\n")
```

The effect of this is to add a newline to the end of its input before passing it to `putStr`.

Writing values in general

The Haskell prelude provides the class `Show` with the function

```
show :: Show a => a -> String
```

which can be used to write values of many types. For example, we can define a general print function from the standard prelude thus

```
print :: Show a => a -> IO ()
print = putStrLn . show
```

Returning a value: `return`

Suppose we want to write an I/O action which does no I/O but does return a value – we will see examples of this in due course. This is achieved by the built-in function

```
return :: a -> IO a
```

The effect of `return x` is to do no I/O, but simply to return the result `x`.

If we're writing a program of type `IO ()` then `return ()` has the effect of a '`skip`' in a traditional language: it's a command that does nothing. We use this in cases where we only want to perform an action when a condition holds, like this:

```
if condition
  then action
  else return ()
```

Running an I/O program

We have written a simple I/O program, namely `helloWorld`; how is it run? In GHCi we can evaluate it at the prompt:

```
Main> helloWorld
Hello, World!
Main> ...
```

Strictly speaking, the `main` definition of a Haskell program should be of type `IO a` for some `a`. In GHCi, if we ask to evaluate an expression `e` of type `b` then it is wrapped up as an object of type `IO ()` by applying the `print` function.

This completes our introduction to the basic I/O functions in the standard prelude as well as the method by which `IO` programs are run.

Next we look at how programs are sequenced, and also how to use the values read in by means of input programs like `getLine`; this is the topic of the next section.

8.4 The `do` notation

The `do` notation gives us a way of building IO programs from the components that we discussed in the previous section. The `do` notation supports two things:

- it is used to *sequence* I/O programs, and
- it is used to **name**[2] the values returned by `IO` actions; this means that the later actions can depend on values captured earlier in the program.

Together these ideas make a `do` expression appear like a simple imperative program, containing a sequence of commands and assignments; although this analogy is not complete – we examine how it breaks down in the next section – it shows that the model of I/O given by the `IO` types is a familiar one, albeit in a different guise.

Sequencing I/O actions

One purpose of the `do` construct is to sequence I/O actions and we show how it is used through a series of examples.

Examples

1. We begin by looking at the definition of `putStrLn` from the standard prelude. The effect of `putStrLn str` is to do two things: first the string `str` is output, then a newline. This is accomplished by

```
putStrLn :: String -> IO ()

putStrLn str = do putStr str
                  putStr "\n"
```

[2] Or 'capture' or 'bind'.

Here we see the effect of do is to sequence a number of IO actions into a single action. The syntax of do is governed by the offside rule, and do can take any number of arguments. We see an example of more arguments next.

2. We can write an I/O program to print something four times. The first version of this is

```
put4times :: String -> IO ()

put4times str
  = do putStrLn str
       putStrLn str
       putStrLn str
       putStrLn str
```

3. So far, we have only seen examples of output, but we can also make inputs a part of a sequence of actions. For instance, we can read two lines of input and then output the message `"Two lines read."` thus:

```
read2lines :: IO ()

read2lines
  = do getLine
       getLine
       putStrLn "Two lines read."
```

and by analogy with Example 3 it is not difficult to see that we could write an I/O program which reads an arbitrary number of lines.

Capturing the values read

As was apparent in Section 8.2, it is necessary to be careful in the way that the results of input actions are handled. The operation `inputInt :: Integer` was shown to be too powerful to fit into the functional model, but some mechanism to handle input values is required. This is the second purpose of the do notation; it is only possible to use the result of an input within a do expression, and this limitation prevents the I/O actions from 'contaminating' the whole program.

The sequence of examples continues by examining this aspect of the do notation.

Examples

4. The last example read two lines, but did nothing with the results of the getLine actions. How can we use these lines in the remainder of the I/O program? As part of a do program we can **name** the results of `IO a` actions. A program to read a line and then write that line is given by

```
getNput :: IO ()

getNput = do line <- getLine
             putStrLn line
```

where the '`line <-`' names the result of the getLine.

If you are familiar with imperative programming you can think of this as like an assignment to a variable, as in

```
line := getLine
```

but you should be aware that there are important differences between the names in a Haskell I/O program and the variables in an imperative program. The essential difference is that each 'var <-' creates a *new* variable var, and so the language permits '**single** assignment' rather than the 'updatable assignment' familiar from the vast majority of modern imperative languages; we look at an example of the different in the exercises for Section 8.5.

5. We are not forced simply to output the lines we have read, unchanged, so that we might define

```
reverse2lines :: IO ()

reverse2lines
  = do line1 <- getLine
       line2 <- getLine
       putStrLn (reverse line2)
       putStrLn (reverse line1)
```

In this example, we read two lines, and then write them in the opposite order, reversed.

Local definitions in a do expression

The notation var <- getLine names the output of the getLine, and so acts like a definition. It is also possible to make local definitions within a do expression so that we can revisit the last example, as follows.

Example

6. Example 5 can be redefined to contain local definitions of the reversed lines

```
reverse2lines :: IO ()

reverse2lines
  = do line1 <- getLine
       line2 <- getLine
       let rev1 = reverse line1
       let rev2 = reverse line2
       putStrLn rev2
       putStrLn rev1
```

Reading values in general

Haskell contains the class Read with the function

```
read :: Read a => String -> a
```

which can be used to parse a string representing a value of a particular type into that value.

Example

7. As an example, suppose that we want to write an I/O program to read in an integer value. To read an integer from a line of input we start by saying

```
do line <- getLine
```

but then we need to sequence this with an I/O action to return the `line` interpreted as an `Integer`. We can convert the `line` to an integer by the expression

```
read line :: Integer
```

What we need is the `IO Integer` action which returns this value – this is the purpose of `return` introduced in the previous section. Our program to read an `Integer` is therefore

```
getInt :: IO Integer

getInt = do line <- getLine
            return (read line :: Integer)
```

Summary

This section has shown that a `do` expression provides a context in which to do sequential programming. It is possible to program complicated I/O interactions, by sequencing simpler I/O programs. Moreover, the '`<-`' allows us to name the value returned by an action and then to use this named value in the remainder of the I/O program. It is also possible to make these programs more readable by judicious use of `let` definitions to name intermediate calculations.

In the next section we look at how to write repetitive I/O programs, reading all the lines in the input, for example. We shall see that this can be done by defining a looping construct recursively. We also discuss the way in which '`<-`' behaves differently from the usual assignment operator.

Exercises

8.10 Write an I/O program which will read a line of input and test whether the input is a palindrome. The program should 'prompt' for its input and also output an appropriate message after testing.

8.11 Write an I/O program which will read two integers, each on a separate line, and return their sum. The program should prompt for input and explain its output.

8.12 Define a function

```
putNtimes :: Integer -> String -> IO ()
```

so that the effect of `putNtimes n str` is to output `str`, a string, `n` times, one per line.

8.13 Write an I/O program which will first read a positive integer, `n` say, and then read `n` integers and write their sum. The program should prompt appropriately for its inputs and explain its output.

8.5 Loops and recursion

In this section we examine how to build I/O programs with a repetitive nature; again we do this by working through a series of examples, concluding with a general pattern for recursive IO programs. In particular we program a number of different variations of a function to copy lines from input to output.

Examples

8. The simplest copying program loops forever:

```
copy :: IO ()

copy =
    do line <- getLine
       putStrLn line
       copy
```

The effect of `copy` is to read a line, and name the result `line`; in the next step this is output, and the final 'command' is to call `copy` again, so looping forever. This can be run within GHCi simply by typing `copy`; it can be interrupted by typing `Ctrl-C`.

Programs like this where the only recursive call to the function is the last statement in the `do` block are called **tail recursive**. Typically tail recursive functions are efficient to implement, because they resemble a loop in an imperative language.

9. We can control the number of lines that are copied by passing the number as a parameter:

```
copyN :: Integer -> IO ()

copyN n =
    if n <= 0
    then return ()
    else do line <- getLine
            putStrLn line
            copyN (n-1)
```

The effect of `copyN 3` should be to copy three lines of input to output. In general, if the number of lines to be copied is less then or equal to zero – the `then` branch – the program just returns `()`, which does no IO. On the other hand, in the `else` branch we read a line, print it out, and then call `copyN (n-1)`.

The `Integer` variable here is sometimes called the **loop data** and the value of the loop data is usually modified in the tail recursive call. You can think of the value of the data as like the value of a variable in an imperative language: in this case the value decreases by one at each call, and so we 'count down' to the base case where the program terminates.

10. We can also control the termination of the loop by a condition on the data; we will copy lines until an empty line is encountered.

```
copyEmpty :: IO ()

copyEmpty =
    do line <- getLine
       if line == ""
          then return ()
          else do putStrLn line
                  copyEmpty
```

Here we first get a line from the input, and call it `line`. If it is empty we terminate – just as we did in the last example – by calling `return ()`. If it is not empty, we output the line, and tail-recursively call `copyEmpty`.

11. Putting together what we saw in examples 9 and 10, we can *count* the number of lines that we have copied, outputting this when we reach an empty line:

```
copyCount :: Integer -> IO ()

copyCount n =
    do line <- getLine
       if line == ""
          then putStrLn (show n ++ " lines copied.")
          else do putStrLn line
                  copyCount (n+1)
```

This behaves like example 10, except that we use the loop data to keep track of the number of lines copied. We can see this in action here:

```
*RPS> copyCount 0
foo
foo
bar
bar

2 lines copied.
```

We will now look at how these ideas can be used to program a Rock – Paper – Scissors tournament.

Exercises

8.14 Define a `wc` function which copies input to output until an empty line is read. The program should then output the number of lines, words and characters that have been copied. [`wc` is a standard unix command line program.]

8.15 Define an interactive palindrome checker. You should neglect capitalization, white space and punctuation, so that

```
Madam I'm Adam.
```

is recognized as a palindrome.

8.16 Write a program which repeatedly reads lines and tests whether they are palindromes until an empty line is read. The program should explain clearly to the user what input is expected and output is produced.

8.17 Write a program which repeatedly reads integers (one per line) until finding a zero value and outputs the sum of the inputs read.

8.18 Write a program which repeatedly reads integers (one per line) until finding a zero value and outputs a sorted version of the inputs read. Which sorting algorithm is most appropriate in such a case?

8.19 Explain the behaviour of this `copy` program

```
copy :: IO ()

copy =
    do
      line <- getLine
      let whileCopy =
              do
                if (line == "")
                  then (return ())
                  else
                    do putStrLn line
                       line <- getLine
                       whileCopy
      whileCopy
```

where the definition of `whileCopy` is modelled on a `while` loop in a traditional programming language.

8.6 Rock – Paper – Scissors: playing the game

In this section we look at how to play the Rock – Paper – Scissors game, both interactively and by playing one strategy off against another.

Playing interactively

You can play interactively against a particular strategy using the function

```
play :: Strategy -> IO ()
```

so that `play rock` allows you to play against the strategy which always plays `Rock`. More interesting to play against is

```
randomPlay :: IO ()
```

which makes a random choice of strategy for you to play against: this is much more of a challenge to play! A typical example in action is shown here, where you jusy have to