12.12 Define a function

\[ \text{train} :: \text{Moves} \to [\text{Strategy}] \to \text{Strategy} \]

which is supplied with a list of opponent's moves to train it, and a list of possible strategies to use. The function should run all the strategies in the list on the list of moves, and choose the strategy which is most successful in winning against the given moves.

12.3 Functions as data: recognizing regular expressions

Regular expressions are patterns which can be used to describe sets of strings of characters of various kinds, such as these.

- The identifiers of a programming language – strings of alphanumerical characters which begin with an alphabetic character.
- The numbers – integer or real – given in a programming language.
- Regular expressions can also be used to extend the pattern language in a programming language, allowing functions to match in more powerful ways than those built in.

There are five sorts of pattern, or regular expression:

\[ \varepsilon \]

This is the Greek character epsilon, which matches the empty string.

\[ x \]

x is any character. This matches the character itself.

\[ (r_1 | r_2) \]

\( r_1 \) and \( r_2 \) are regular expressions.

\[ (r_1 r_2) \]

\( r_1 \) and \( r_2 \) are regular expressions.

\[ (r)^* \]

\( r \) is a regular expression.

Examples of regular expressions include \((a | (ba))\), \(((ba) | (e | (a)^*))\) and \(\text{hello}\). In order to give a more readable version of these, it is assumed that * binds more tightly than juxtaposition \((i.e. (r_1 r_2))\), and that juxtaposition binds more tightly than \(|\). This means that \(r_1 r_2^*\) will mean \((r_1 (r_2)^*)\), not \(((r_1 r_2)^*)\), and that \(r_1 | r_2 r_3\) will mean \(r_1 | (r_2 r_3)\), not \((r_1 | r_2) r_3\).

Regular expressions are patterns, so we need to describe which strings match each regular expression.

\[ \varepsilon \]

The empty string matches epsilon.

\[ x \]

The character \( x \) matches the pattern \( x \), for any character \( x \).

\[ (r_1 | r_2) \]

The string \( st \) will match \((r_1 | r_2)\) if \( st \) matches either \( r_1 \) or \( r_2 \) (or both).

\[ (r_1 r_2) \]

The string \( st \) will match \((r_1 r_2)\) if \( st \) can be split into two substrings \( st_1 \) and \( st_2 \), \( st = st_1++st_2 \), so that \( st_1 \) matches \( r_1 \) and \( st_2 \) matches \( r_2 \).

\[ (r)^* \]

The string \( st \) will match \((r)^*\) if \( st \) can be split into zero or more substrings, \( st = st_1++st_2++\ldots++st_n \), each of which matches \( r \). The zero case implies that the empty string will match \((r)^*\) for any regular expression \( r \).
Let's build a model of regular expressions in Haskell; we choose to embed them as functions from String to Boolean, which is the function which recognizes exactly the strings matching the pattern.

```haskell
type RegExp = String -> Bool
```

Now we define the five different kinds of regular expression, starting off with epsilon, \( \varepsilon \), which is matched by the empty string only. We use an operator section to define the function:

```haskell
epsilon :: RegExp
epsilon = (=="")
```

We use a similar definition for the function that recognizes a single character, passed in as its argument

```haskell
char :: Char -> RegExp
char ch = (==[ch])
```

We next define the Haskell operator \( ||| \), which implements the 'or' operation, \( . \). Applying this to \( e_1 \) and \( e_2 \) gives a function which takes the string \( x \) to the 'or' of the two values \( e_1 \ x \) and \( e_2 \ x \):

```haskell
(|||) :: RegExp -> RegExp -> RegExp

\( e_1 \ ||| \ e_2 = \n\ x \rightarrow e_1 \ x \ || \ e_2 \ x \)
```

Sequencing the match of two regular expressions is given by the Haskell \( <*> \) operator. In defining this we'll use the function \( \text{splits} \) that returns a list of all the ways that a string can be split in two

```haskell
splits "Spy" \rightarrow [(""","Spy""),("S","py""),("Sp","y""),("Spy","""")]
```

Now we can give the definition of \( <*> \)

```haskell
(\<\>) :: RegExp -> RegExp -> RegExp

\( e_1 \ <*> \ e_2 = \n\ x \rightarrow \text{or} [ e_1 \ y \&& \ e_2 \ z \ | \ (y,z) \leftarrow \text{splits} \ x ] \)
```

How does this definition work? The list comprehension runs through all the splits of the input string, \( x \). For each of these we test whether the front half (\( y \)) matches the first pattern (\( e_1 \)) by applying \( e_1 \) to \( x \), and similarly we apply \( e_2 \) to the second half of the string (\( z \)). Since we need both matches to succeed, we combine the results with 'and', \( \&\& \). The result of this is to give a list of the answers for each split: we only need one of these to succeed, and so we combine the results with the built-in function \( \text{or} \) that takes the 'or' of a list of Boolean values.
We can define the star operation using the operators that we've already defined, like this:

\[
\text{star} :: \text{RegExp} \rightarrow \text{RegExp}
\]

\[
\text{star } p = \text{epsilon} || (p \leftrightarrow \text{star } p)
\]

The definition says 'to match \((p)^*\), either match it zero times (epsilon) or match \(p\) followed by \((p)^*\)'. What is elegant about this is that we just used the operators \(|||\) and \(\leftrightarrow\), together with recursion, to make the definition at the level of the RegExp type; we didn't need to think about \text{star } p being a function.

\section*{Getting star right}

There is a flaw in the definition of \text{star} that we have just given: if it is possible for \(p\) to match the empty string, i.e. if \(p \text{ ""} \) is True, then the definition may go into an infinite loop.

We need to modify the definition of \text{star} to say instead that

\[
\text{star } p = \text{epsilon} || (p \leftrightarrow\leftrightarrow \text{star } p)
\]

where \(\leftrightarrow\leftrightarrow\) is defined like \(\leftrightarrow\) except that it omits the split \("", st\) from \text{splits}\(st\). This change is enough to make sure that the infinite loop is avoided, as it means that the first match of \(p\) can't be with an empty string, and so the next match of \((p)^*\) must be on a shorter string.

\section*{Exercises}

\subsection*{12.13} Define the function

\[
\text{splits} :: [a] \rightarrow ([a],[a])
\]

which defines the list of all the ways that a list can be split in two (see the example of \text{splits} "Spy" above).

\subsection*{12.14} By trying it with a number of examples, which strings does this regular expression match?

\[
\text{star } ((a || b) \leftrightarrow (a || b))
\]

where \(a\) and \(b\) are defined by

\[
a, b :: \text{RegExp}
\]

\[
a = \text{char } 'a''
\]

\[
b = \text{char } 'b''
\]

\subsection*{12.15} Which strings does this regular expression match?

\[
\text{star } (\text{star } ((a || b) \leftrightarrow (a || b)))
\]
12.16 Define functions

\[
\text{option, plus} :: \text{RegExp} \to \text{RegExp}
\]

where \(\text{option}~e\) matches zero or one occurrences of the pattern \(e\), and \(\text{plus}~e\) matches one or more occurrences of the pattern \(e\).

12.17 Define regular expressions which match

- Strings of digits which begin with a non-zero digit.
- Fractional numbers: two strings of digits separated by "."; make sure that these numbers have no superfluous zeroes at the beginning or the end, so exclude strings like "01.34" and "1.20".

In doing this you might find it useful to define a function

\[
\text{range} :: \text{Char} \to \text{Char} \to \text{RegExp}
\]

so that, for example, \(\text{range } 'A' 'Z'\) will match any capital letter.

12.18 Give regular expressions describing the following sets of strings

- All strings of as and bs containing at most two as.
- All strings of as and bs containing exactly two as.
- All strings of as and bs of length at most three.
- All strings of as and bs which contain no repeated adjacent characters, that is no substring of the form aa or bb.

12.19 [Hard] Add to the regular expressions the facility to name substrings that match particular sub-expressions, so that instead of returning a \text{Bool} a \text{RegExp} will return a \text{list} of bindings of names to substrings.

Why a list? First, it allows for no matching to happen (empty list, []) or for multiple matches, which can also happen as matching the regular expressions \((r_1 r_2)\) and \((r)\)\star can succeed in multiple different ways.

### 12.4 Case studies: functions as data

This section introduces a number of shorter case studies which use functions to represent data. First we show then we can model natural numbers as higher-order functions, next we look at graphics can be represented by functions, in a ‘bit-mapped’ style.

#### Natural numbers as functions

We can represent the natural numbers 0, 1, 2, … by functions of type

\[
\text{type Natural}~a = (a \to a) \to (a \to a)
\]

where the number \(n\) is represented by ‘apply the argument \(n\) times’, so

\[
\begin{align*}
\text{zero } f &= \text{id} \\
\text{one } f &= f \\
\text{two } f &= f \circ f
\end{align*}
\]