1 Introduction

In these notes Haskell is used as a vehicle to introduce regular expressions, pattern
matching, and their implementations by means of non-deterministic and determin-
istic automata.

As part of the material, we give an implementation of the ideas, contained in a
set of files. References to this material are scattered through the text. The files can
be obtained by following the instructions in

http://www.cs.ukc.ac.uk/people/staff/sjt/Further/regExp.html

This material is based on the treatment of the subject in [Aho et. al.], but provides
full implementations rather than their pseudo-code versions of the algorithms.

The material gives an illustration of many of the features of Haskell, includ-
ing polymorphism (the states of an NFA can be represented by objects of any
type); type classes (in practice the states need to have equality and an ordering
defined on them); modularisation (the system is split across a number of modules);
higher-order functions (used in finding limits of processes, for example) and other
features. A tutorial introduction to Haskell can be found in [Thompson].

The paper begins with definitions of regular expressions, and how strings are
matched to them; this also gives our first Haskell treatment also. After describing
the abstract data type of sets we define non-deterministic finite automata, and their
implementation in Haskell. We then show how to build an NFA corresponding
to each regular expression, and how such a machine can be optimised, first by
transforming it into a deterministic machine, and then by minimising the state space
of the DFA. We conclude with a discussion of regular definitions, and show how
recognisers for strings matching regular definitions can be built.

2 Regular Expressions

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

- the identifiers of a programming language – strings of alphanumeric charac-
ters which begin with an alphabetic character;

- the numbers – integer or real – given in a programming language; and so on.

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 \((a|(ba)), ((ba)|()|(a)^*))\) and 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 \( (r_1|r_2) \). 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 \( (r_1;r_2);r_3 \).

A Haskell algebraic type representing regular expressions is given by

```haskell
data Reg = Epsilon |
  Literal Char |
  Or Reg Reg |
  Then Reg Reg |
  Star Reg
 deriving Eq
```

The statement \texttt{deriving Eq} at the end of the definition ensures that the type \texttt{Reg} is made to belong to the type class \texttt{Eq}; in other words the equality function \texttt{==} is defined over \texttt{Reg}.

This definition and those which follow can be found in the file \texttt{RegExp.hs}; this file contains the module \texttt{RegExp}, which will be included in other modules in the system. The Haskell representations of \((a|(ab))\) and \((((ba)|(\varepsilon|((a)^*)))\) are

\begin{align*}
\text{Or} \ (\text{Literal } 'a') & \ (\text{Then} \ (\text{Literal } 'a') \ (\text{Literal } 'b')) \\
\text{Or} \ (\text{Then} \ (\text{Literal } 'b') \ (\text{Literal } 'a')) \\
& \ (\text{Or} \ Epsilon \ (\text{Star} \ (\text{Literal } 'a'))) \\
\end{align*}

respectively. In order to shorten these definitions we will usually define constant literals such as

\begin{align*}
a & = \text{Literal } 'a' \\
b & = \text{Literal } 'b' \\
\end{align*}

so that the expressions above become

\begin{align*}
\text{Or} \ a \ (\text{Then} \ a \ b) & \quad \text{Or} \ (\text{Then} \ b \ a) \ (\text{Or} \ Epsilon \ (\text{Star} \ a)) \\
\text{If we use the infix forms of \texttt{Or} and \texttt{Then}, 'Or' and 'Then', they read} \\
a 'Or' \ (a 'Then' \ b) & \quad (a 'Then' \ b) 'Or' \ (Epsilon 'Or' (Star \ a)) \\
\end{align*}
Functions over the type of regular expressions are defined by recursion over the structure of the expression. Examples include

\[
\text{literals :: Reg -> [Char]}
\]

\[
\text{literals Epsilon} = []
\]
\[
\text{literals (Literal ch)} = \text{[ch]}
\]
\[
\text{literals (Or r1 r2)} = \text{literals r1 ++ literals r2}
\]
\[
\text{literals (Then r1 r2)} = \text{literals r1 ++ literals r2}
\]
\[
\text{literals (Star r)} = \text{literals r}
\]

which prints a list of the literals appearing in a regular expression, and

\[
\text{printRE :: Reg -> [Char]}
\]

\[
\text{printRE Epsilon} = "@"
\]
\[
\text{printRE (Literal ch)} = \text{[ch]}
\]
\[
\text{printRE (Or r1 r2)}
\] = "(" ++ \text{printRE r1} ++ "|" ++ \text{printRE r2} ++ ")"
\[
\text{printRE (Then r1 r2)}
\] = "(" ++ \text{printRE r1} ++ \text{printRE r2} ++ ")"
\[
\text{printRE (Star r)} = "(" ++ \text{printRE r} ++ ")*"
\]

which gives a printable form of a regular expression. Note that `@` is used to represent epsilon in ASCII. The type \text{Reg} can be made to belong to the \text{Show} class thus:

\[
\text{instance Show Reg where}
\]
\[
\text{show} = \text{printRE}
\]

or indeed an instance could be derived automatically (like \text{Eq} earlier).

\text{Exercises}

1. Write a more readable form of the expression \(((a|b)|c)((a)++(b)++)(c|d))

2. What is the unabbreviated form of \(((x)++(y)++)++)

\section{Matching regular expressions}

Regular expressions are patterns. We should ask which strings match each regular expression.
The empty string matches epsilon.

The character x matches the pattern x, for any character x.

(r₁ | r₂) The string st will match (r₁ | r₂) if st matches either r₁ or r₂ (or both).

(r₁ r₂) The string st will match (r₁ r₂) if st can be split into two substrings st₁ and st₂, st = st₁ ++ st₂, so that st₁ matches r₁ and st₂ matches r₂.

(r)* The string st will match (r)* if st can be split into zero or more substrings, st = st₁ ++ st₂ ++ ... ++ stₙ, each of which matches r. The zero case implies that the empty string will match (r)* for any regular expression r.

This can be implemented in Haskell, in the module Matches. The first three cases are a simple transliteration of the definitions above.

matches :: Reg -> String -> Bool

matches Epsilon st = (st == "")
matches (Literal ch) st = (st == [ch])
matches (Or r₁ r₂) st
  = matches r₁ st || matches r₂ st

In the case of juxtaposition, we need an auxiliary function which gives the list containing all the possible ways of splitting up a list.

splits :: [a] -> [[a]]

splits st = [ splitAt n st | n <- [0 .. length st] ]

For example, splits [2,3] is [[[],[2,3]],[[2],[3]],[[2],[3]],[[],[]]]. A string will match (Then r₁ r₂) if at least one of the splits gives strings which match r₁ and r₂.

matches (Then r₁ r₂) st
  = or [matches r₁ s₁ && matches r₂ s₂ | (s₁,s₂)<-splits st]

The final case is that of Star. We can explain a* as either ε or as a followed by a*. We can use this to implement the check for the match, but it is problematic
when a can be matched by ε. When this happens, the match is tested recursively on
the same string, giving an infinite loop. This is avoided by disallowing an epsilon
match on a – the first match on a has to be non-trivial.

\[
\text{matches} \ (\text{Star} \ r) \ st \\
= \text{matches} \ \text{Epsilon} \ st \ |
\]
\[
\text{or} \ [ \text{matches} \ r \ s1 \ \&\& \ \text{matches} \ (\text{Star} \ r) \ s2 \ |
\]
\[
(s1, s2) \leftarrow \text{frontSplits} \ st
\]

\text{frontSplits} is defined like \text{splits} but so as to exclude the split (\text{[]}, st).

\textbf{Exercises}

3. Argue that the string ε matches (a| (bc)*)* and that the string abba matches
   a((b|a)*(ba)*).

4. Why does the string bab not match a((b|a)*(ba)*)?

5. Give informal descriptions of the sets of strings matching the following regular
   expressions.

   \[(a|b)a(a|b)a(a|b) \quad (a|b)a(a|b)(a|b) \quad \varepsilon|a|b|ba|b?|a|b|a?\]

6. Give regular expressions describing the following sets of strings
   \begin{itemize}
     \item All strings of as and bs containing at most two as.
     \item All strings of as and bs containing exactly two as.
     \item All strings of as and bs of length at most three.
     \item All strings of as and bs which contain no repeated adjacent characters, that
          is no substring of the form aa or bb.
   \end{itemize}

4 \ Set

A set is a collection of elements of a particular type, which is both like and unlike
a list. Lists are familiar from Haskell, and examples include

\[
[\text{Joe}, \text{Sue}, \text{Ben}] \quad [\text{Ben}, \text{Sue}, \text{Joe}] \\
[\text{Joe}, \text{Sue}, \text{Sue}, \text{Ben}] \quad [\text{Joe}, \text{Sue}, \text{Ben}, \text{Sue}]
\]
module Sets ( Set ,
   empty      ,  -- Set a
   sing       ,  -- a -> Set a
   memSet     ,  -- Ord a => Set a -> a -> Bool
   union,inter,diff ,  -- Ord a => Set a -> Set a -> Set a
   eqSet      ,  -- Eq a => Set a -> Set a -> Bool
   subSet     ,  -- Ord a => Set a -> Set a -> Bool
   makeSet    ,  -- Ord a => [a] -> Set a
   mapSet     ,  -- Ord b => (a -> b) -> Set a -> Set b
   filterSet  ,  -- (a -> Bool) -> Set a -> Set a
   foldSet    ,  -- (a -> a -> a) -> a -> Set a -> a
   showSet    ,  -- Show a => Set a -> String
   card       ,  -- Set a -> Int
   flatten    ,  -- Set a -> [a]
   setlimit   -- Eq a => (Set a -> Set a) -> Set a -> Set a
   ) where

import List hiding ( union )

Figure 1: The functions in the set abstract data type

Each of these lists is different – not only do the elements of a list matter, but also the order in which they occur, and their multiplicity (the number of times each element occurs).

In many situations, order and multiplicity are irrelevant. If we want to talk about the collection of people coming to our birthday party, we just want the names – we cannot invite someone more than once, so multiplicity is not important; the order we might list them in is also of no interest. In other words, all we want to know is the set of people coming. In the example above, this is the set containing Joe, Sue and Ben.

Sets can be implemented in a number of ways in Haskell, and the precise form is not important for the user. It is sensible to declare the type as an abstract data type, so that its implementation is hidden from the user. This is done by failing to export the constructor of the type which implements sets. Details of this mechanism are given in Chapter 16 of [Thompson], which also discusses the particular implementation given here in rather more detail. The definition is given in the module Sets which is defined in the file Sets.hs. The heading of the module is illustrated in Figure 1.
The implementation we have given represents a set as an *ordered list of elements without repetitions*, wrapped up by the constructor `SetI`. For instance, the set of birthday party attendees will be given by

\[
\text{SetI } [\text{Ben, Joe, Sus}]
\]

The implementation of the type `Set` is hidden because the `SetI` constructor for this type is not exported from the module.

Since the lists are ordered we expect to have an ordering over the type of set elements; it is this requirement that gives rise to the constraint `Ord a` in many of the set-manipulating functions. The individual functions are described and implemented as follows.

The **empty** set is the empty list

\[
\text{empty} = \text{SetI } []
\]

and **sing** a is the singleton set, consisting of the single element a

\[
\text{sing } x = \text{SetI } [x]
\]

Figure 2 defines the functions `union`, `inter`, `diff` which give the union, intersection and difference of two sets. The union consists of the elements occurring in either set (or both), the intersection of those elements in both sets and the difference of those elements in the first but not the second set. (Note also that `union` here is a redefinition of the function with the same name from the `Prelude.hs`.)

These definitions each follow the same pattern: a function like `union` implements the operation over lists, and the top-level `union` function lifts this to operate over the lists ‘wrapped’ by the constructor `SetI`.

The operation `memSet xs x` tests whether x is a member of the set `xs`. Note that this is an optimisation of the function `elem` over lists; since the list is ordered, we need look no further once we have found an element greater than the one we seek.

\[
\text{memSet :: Ord a } \Rightarrow \text{ Set a } \rightarrow \text{ a } \rightarrow \text{ Bool }
\]

\[
\text{memSet (SetI [] ) } y = \text{False}
\]

\[
\text{memSet (SetI (x:xs)) } y
\]

\[
| x < y = \text{memSet (SetI xs) } y \\
| x = y = \text{True} \\
| \text{otherwise} = \text{False}
\]

The function `subSet xs ys` tests whether `xs` is a subset of `ys`; that is whether every element of `xs` is an element of `ys`. 

union :: Ord a => Set a -> Set a -> Set a
union (SetI xs) (SetI ys) = SetI (uni xs ys)

uni :: Ord a => [a] -> [a] -> [a]
uni [] ys     = ys
uni xs []    = xs
uni (x:xs) (y:ys)
  | x<y     = x : uni xs (y:ys)
  | x==y    = x : uni xs ys
  | otherwise = y : uni (x:xs) ys

inter :: Ord a => Set a -> Set a -> Set a
inter (SetI xs) (SetI ys) = SetI (int xs ys)

int :: Ord a => [a] -> [a] -> [a]int [] ys     = []int xs []    = []int (x:xs) (y:ys)
  | x<y     = int xs (y:ys)
  | x==y    = x : int xs ys
  | otherwise = int (x:xs) ys

diff :: Ord a => Set a -> Set a -> Set a
diff (SetI xs) (SetI ys) = SetI (dif xs ys)

dif :: Ord a => [a] -> [a] -> [a]
dif [] ys     = []
dif xs []    = xs
dif (x:xs) (y:ys)
  | x<y     = x : dif xs (y:ys)
  | x==y    = dif xs ys
  | otherwise = dif (x:xs) ys

Figure 2: Set operations
subSet :: Ord a => Set a -> Set a -> Bool
subSet (SetI xs) (SetI ys) = subS xs ys

subS :: Ord a => [a] -> [a] -> Bool
subS [] ys = True
subS xs [] = False
subS (x:xs) (y:ys)
    | x<y     = False
    | x==y    = subS xs ys
    | x>y     = subS (x:xs) ys

eqSet x y tests whether two sets are equal.
eqSet (SetI xs) (SetI ys) = (xs == ys)

and an instance declaration for Eq over Set makes eqSet into == over Set.

The functions mapSet, filterSet and foldSet behave like map, filter and foldr except that they operate over sets. separate is a synonym for filterSet.

mapSet :: Ord b => (a -> b) -> Set a -> Set b
mapSet f (SetI xs) = makeSet (map f xs)

filterSet :: (a -> Bool) -> Set a -> Set a
filterSet p (SetI xs) = SetI (filter p xs)

foldSet :: (a -> a -> a) -> a -> Set a -> a
foldSet f x (SetI xs) = (foldr f x xs)

The operation makeSet turns a list into a set

makeSet :: Ord a => [a] -> Set a
makeSet = SetI . remDups . sort
    where
        remDups [] = []
        remDups [x] = [x]
        remDups (x:y:xs)
            | x<y     = x : remDups (y:xs)
            | otherwise = remDups (y:xs)

showSet f gives a printable version of a set, one item per line, using the function f to give a printable version of each element.

showSet :: Show a => Set a -> String
showSet (SetI xs) = concat (map ((""++"\n") . show) xs)
card gives the number of elements in a set,

\[
\text{card} :: \text{Set } a \rightarrow \text{Int} \\
\text{card} (\text{SetI } xs) = \text{length } xs
\]

flatten turns a set into an ordered list of the elements of the set

\[
\text{flatten} :: \text{Set } a \rightarrow [a] \\
\text{flatten} (\text{SetI } xs) = xs
\]

Obviously this breaks the abstraction barrier, but it is necessary in some situations to do this.

The function \(\text{setlimit } f x\) gives the ‘limit’ of the sequence

\[x, f(x), f(f(x)), f(f(f(x))), \ldots\]

that is the first element in the sequence whose successor is equal, as a set, to the element itself. In other words, keep applying \(f\) until a fixed point or limit is reached.

\[
\text{setlimit} :: \text{Eq a} \Rightarrow (\text{Set } a \rightarrow \text{Set } a) \rightarrow \text{Set } a \\
\text{setlimit } f s = \begin{cases} \\
s = \text{next} & \text{=} s \\
\text{otherwise} & = \text{setlimit } f \text{ next} \\
\end{cases} \\
\text{where} \\
\text{next} = f s
\]

**Exercises**

7. Define the function \(\text{powerSet} :: \ldots \Rightarrow \text{Set } a \rightarrow \text{Set } (\text{Set } a)\) which returns the set of all subsets of a set. What context information is required on the type \(a\)?

8. How would you define the functions

\[
\text{setUnion} :: \ldots \Rightarrow \text{Set } (\text{Set } a) \rightarrow \text{Set } a \\
\text{setInter} :: \ldots \Rightarrow \text{Set } (\text{Set } a) \rightarrow \text{Set } a
\]

which return the union and intersection of a set of sets? What contexts are required on the types?

9. Can infinite sets (of numbers, for instance) be adequately represented by ordered lists? Can you tell if two infinite lists are equal, for instance?

10. The abstract data type \(\text{Set } a\) can be represented in a number of different ways. Alternatives include: arbitrary lists (rather than ordered lists without repetitions), and boolean valued functions, that is elements of the type \(a \rightarrow \text{Bool}\). Give implementations of the type using these two representations.
5 Non-deterministic Finite Automata

A Non-deterministic Finite Automaton or NFA is a simple machine which can be used to recognise regular expressions. It consists of four components

- A finite set of states, \( S \).
- A finite set of moves.
- A start state (in \( S \)).
- A set of terminal or final states (a subset of \( S \)).

In the Haskell module \texttt{NfaTypes} this is written

\begin{verbatim}
data Nfa a = NFA (Set a)
    (Set (Move a))
   a
    (Set a)
deriving (Eq,Show)
\end{verbatim}

This has been represented by an algebraic type rather than a 4-tuple simply for readability. The type of states can be different in different applications, and indeed in the following we use both numbers and sets of numbers as states.

A move is between two states, and is either given by a character, or an \( \varepsilon \).

\begin{verbatim}
data Move a = Move a Char a |
    Emove a a
deriving (Eq,Ord,Show)
\end{verbatim}

The first example of an NFA, called \( M \), follows.

![Diagram of NFA M]

The states are 0, 1, 2, 3, with the start state 0 indicated by an incoming arrow, and the final states indicated by shaded circles. In this case there is a single final state, 3. The moves are indicated by the arrows, marked with characters \( a \) and \( b \) in this case. From state 0 there are two possible moves on symbol \( a \), to 1 and to remain at 0. This is one source of the non-determinism in the machine.

The Haskell representation of the machine is

\begin{verbatim}
data Nfa a = NFA (Set a)
    (Set (Move a))
   a
    (Set a)
deriving (Eq,Show)
\end{verbatim}

This has been represented by an algebraic type rather than a 4-tuple simply for readability. The type of states can be different in different applications, and indeed in the following we use both numbers and sets of numbers as states.

A move is between two states, and is either given by a character, or an \( \varepsilon \).

\begin{verbatim}
data Move a = Move a Char a |
    Emove a a
deriving (Eq,Ord,Show)
\end{verbatim}

The first example of an NFA, called \( M \), follows.

![Diagram of NFA M]

The states are 0, 1, 2, 3, with the start state 0 indicated by an incoming arrow, and the final states indicated by shaded circles. In this case there is a single final state, 3. The moves are indicated by the arrows, marked with characters \( a \) and \( b \) in this case. From state 0 there are two possible moves on symbol \( a \), to 1 and to remain at 0. This is one source of the non-determinism in the machine.

The Haskell representation of the machine is

\begin{verbatim}
data Nfa a = NFA (Set a)
    (Set (Move a))
   a
    (Set a)
deriving (Eq,Show)
\end{verbatim}
A second example, called $N$, is illustrated below.

The Haskell representation of this machine is

```haskell
NFA
(makeSet [0 .. 3])
(makeSet [ Move 0 'a' 0 ,
          Move 0 'a' 1 ,
          Move 0 'b' 0 ,
          Move 1 'b' 2 ,
          Move 2 'b' 3 ])
0
(sing 3)
```

This machine contains two kinds of non-determinism. The first is at state 0, from which it is possible to move to either 1 or 3 on reading a. The second occurs at state 3: it is possible to move 'invisibly' from state 3 to state 4 on the epsilon move, $\epsilon$.

The Haskell code for these machines together with a function `print_nfa` to print an nfa whose states are numbered can be found in the module `NfaMisc`.

How do these machines recognise strings? A move can be made from one state $s$ to another $t$ either if the machine contains `Emove s t` or if the next symbol to
be read is, say, a and the machine contains a move $\text{Move } s \rightarrow t$. A string will be accepted by a machine if there is a sequence of moves through states of the machine starting at the start state and terminating at one of the terminal states – this is called an accepting path. For instance, the path

$$0 \xrightarrow{a} 1 \xrightarrow{b} 2 \xrightarrow{b} 3$$

is an accepting path through $\mathcal{M}$ for the string $aabb$. This means that the machine $\mathcal{M}$ accepts this string. Note that other paths through the machine are possible for this string, an example being

$$0 \xrightarrow{a} 0 \xrightarrow{b} 0 \xrightarrow{b} 0$$

All that is needed for the machine to accept is one accepting path; it does not affect acceptance if there are other non-accepting (or indeed accepting) paths. More than one accepting path can exist. Machine $\mathcal{M}$ accepts the string $ab$ by both

$$0 \xrightarrow{a} 1 \xrightarrow{b} 2$$

and

$$0 \xrightarrow{a} 3 \xrightarrow{\varepsilon} 4 \xrightarrow{b} 5$$

A machine will reject a string only when there is no accepting path. Machine $\mathcal{N}$ rejects the string $a$, since the two paths through the machine labelled by $a$ fail to terminate in a final state:

$$0 \xrightarrow{a} 1 \xrightarrow{a} 3$$

Machine $\mathcal{N}$ rejects the string $aa$ since there is no path through the machine labelled by $aa$: after reading $a$ the machine can be in state 1, 3 or 4, from none of these can an $a$ move be made.

6 Simulating an NFA

As was explained in the last section, a string $xt$ is accepted by a machine $\mathcal{M}$ when there is at least one accepting path labelled by $xt$ through $\mathcal{M}$, and is rejected by $\mathcal{M}$ when no such path exists.

The key to implementation is to explore simultaneously all possible paths through the machine labelled by a particular string. Take as an informal example the string $ab$ and the machine $\mathcal{N}$. After reading no input, the machine can only be in state 0. On reading an $a$ there are moves to states 1 and 3; however this is not the whole story. From state 3 it is possible to make an $\varepsilon$-move to state 4, so after reading a the machine can be in any of the states $\{1,3,4\}$. 

14
On reading a ∶ CQ, we have to look for all the possible ∶ CQ moves from each of the states {1, 3, 4}. From 1 we can move to 2, from 3 to 4 and from 4 to 5 – no ε-moves are possible from the states {2, 4, 5}, and so the states accessible after reading the string ab are {2, 4, 5}. Is this string to be accepted by ∶ H? We accept it exactly if the set contains a final state – it contains both 2 and 5, so it is accepted. Note that the states accessible after reading a are {1, 3, 4}; this set contains no final state, and so the machine ∶ H rejects the string a.

There is a general pattern to this process, which consists of a repetition of

- Take a set of states, such as {1, 3, 4}, and find the set of states accessible by a move on a particular symbol, e.g. b. In this case it is the set {2, 4, 5}. This is called onemove in the module NfaLib.

- Take a set of states, like {1, 3}, and find the set of states accessible from the states by zero or more ε-moves. In this example, it is the set {1, 3, 4}. This is the ε-closure of the original set, and is called closure in NfaLib.

The functions onemove and closure are composed in the function onetrans, and this function is iterated along the string by the trans function of the module ImplementNfa.

**Implementation in Haskell**

We discuss the development of the function

trans :: Ord a => Nfa a -> String -> Set a

top-down. Iteration along a string is given by foldl

foldl :: (Set a -> Char -> Set a)
        -> Set a -> String -> Set a

foldl f r []       = r
foldl f r (c:cs)   = foldl f (f r c) cs

The first argument, f, is the step function, taking a set and a character to the states accessible from the set on the character. The second argument, r, is the starting state, and the final argument is the string along which to iterate.

How does the function operate? If given an empty string, the start state is the result. If given a string (c:cs), the function is called again, with the tail of the string, cs, and with a new starting state, (f r c), which is the result of applying the step function to the starting set of states and the first character of the string. Now to develop trans.
trans mach str
    = foldl step startset str
      where
        step set ch = onetrans mach ch set
        startset = closure mach (sing (startstate mach))

step is derived from onetrans simply by suppling its machine argument mach, similarly startset is derived from the machine mach, using the functions closure and startstate. All these functions are defined in the NfaLib module. We discuss their definitions now.

onetrans :: Ord a => Nfa a -> Char -> Set a -> Set a

onetrans mach c x = closure mach (onemove mach c x)

Next, we examine onemove,

onemove :: Ord a => Nfa a -> Char -> Set a -> Set a

onemove (NFA states moves start term) c x
    = makeSet [ s | t <- flatten x ,
                  Move z d s <- flatten moves ,
                  z==t , c==d ]

The essential idea here is to run through the elements t of the set x and the set of moves, moves looking for all c-moves originating at t. For each of these, the result of the move, s, goes into the resulting set.

The definition uses list comprehensions, so it is necessary first to flatten the sets x and moves into lists, and then to convert the list comprehension into a set by means of makeSet.

closure :: Ord a => Nfa a -> Set a -> Set a

closure (NFA states moves start term)
    = setlimit add
      where
        add stateset = union stateset (makeSet accessible)
          where
            accessible
              = [ s | x <- flatten stateset ,
                    Emove y s <- flatten moves ,
                    y==x ]
The essence of closure is to take the limit of the function which adds to a set of states all those states which are accessible by a single $\varepsilon$-move; in the limit we get a set to which no further states can be added by $\varepsilon$-transitions. Adding the states got by single $\varepsilon$-moves is accomplished by the function add and the auxiliary definition accessible which resembles the construction of onemove.

7 Implementing an example

The machine $P$ is illustrated by

Exercise

11. Give the Haskell definition of the machine $P$.

The $\varepsilon$-closure of the set $\{0\}$ is the set $\{0, 1, 2, 4\}$. Looking at the definition of closure above, the first application of the function add to $\{0\}$ gives the set $\{0, 1\}$; applying add to this gives $\{0, 1, 2, 4\}$. Applying add to this set gives the same set, hence this is the value of setlimit here. The set of states with which we start the simulation is therefore $\{0, 1, 2, 4\}$. Suppose the first input is $a$; applying onemove reveals only one $a$ move, from 2 to 3. Taking the closure of the set $\{3\}$ gives the set $\{1, 2, 3, 4, 6, 7\}$. A $b$ move from here is only from 4 to 5; closing under $\varepsilon$-moves gives $\{1, 2, 4, 5, 6, 7\}$. An $a$ move from here is possible in two ways: from 2 to 3 and from 7 to 8; closing up $\{3, 8\}$ gives $\{1, 2, 3, 4, 6, 7, 8\}$. Is the string aba therefore accepted by $P$? Yes, because 8 is a member of $\{1, 2, 3, 4, 6, 7, 8\}$. This sequence can be illustrated thus.
Exercise

12. Show that the string abb is not accepted by the machine P.

8 Building NFAs from regular expressions

For each regular expression it is possible to build an NFA which accepts exactly those strings matching the expression. The machines are illustrated in Figure 3.

The construction is by induction over the structure of the regular expression: the machines for an character and for $\varepsilon$ are given outright, and for complex expressions, the machines are built from the machines representing the parts. It is straightforward to justify the construction.

$(e|f)$ Any path through $M(e|f)$ must be either a path through $M(e)$ or a path through $M(f)$ (with $\varepsilon$ at the start and end.

ef Any path through $M(ef)$ will be a path through $M(e)$ followed by a path through $M(f)$.

$e^*$ Paths through $M(e^*)$ are of two sorts; the first is simply an $\varepsilon$, others begin with a path through $M(e)$, and continue with a path through $M(e^*)$. In other words, paths through $M(e^*)$ go through $M(e)$ zero or more times.

The machine for the pattern $(ab|ba)^*$ is given by
Figure 3: Building NFAs for regular expressions
The Haskell description of the construction is given in BuildNfa. At the top level the function

```
build :: Reg -> Nfa Int
```
does the recursion. For the base case,

```
build (Literal c)
  = NFA
    (makeSet [0 .. 1])
    (sing (Move 0 c 1))
    0
    (sing 1)
```

The definition of build Epsilon is similar. In the other cases we define

```
build (Or r1 r2)  = m_or  (build r1) (build r2)
build (Then r1 r2) = m_then (build r1) (build r2)
build (Star r)    = m_star (build r)
```
in which the functions m_or and so on build the machines from their components as illustrated in Figure 3.

We make certain assumptions about the NFAs we build. We take it that the states are numbered from 0, with the final state having the highest number. Putting the machines together will involve adding various new states and transitions, and renumbering the states and moves in the constituent machines. The definition of m_or is given in Figure 4, and the other functions are defined in a similar way. The function renumber renumbers states and renumber_move renumbers moves.

9 Deterministic machines

A deterministic finite automaton is an NFA which

- contains no ε-moves, and
- has at most one arrow labelled with a particular symbol leaving any given state.

The effect of this is to make operation of the machine deterministic – at any stage there is at most one possible move to make, and so after reading a sequence of characters, the machine can be in one state at most.
m_or :: Nfa Int -> Nfa Int -> Nfa Int

  m_or (NFA states1 moves1 start1 finish1)
    (NFA states2 moves2 start2 finish2)
      = NFA
        (states1' 'union' states2' 'union' newstates)
        (moves1' 'union' moves2' 'union' newmoves)
        0
        (sing (m1+m2+1))

  where
    m1 = card states1
    m2 = card states2
    states1' = mapSet (renumber 1) states1
    states2' = mapSet (renumber (m1+1)) states2
    newstates = makeSet [0,(m1+m2+1)]
    moves1' = mapSet (renumber_move 1) moves1
    moves2' = mapSet (renumber_move (m1+1)) moves2
    newmoves = makeSet [ Emove 0 1 , Emove 0 (m1+1),
                         Emove m1 (m1+m2+1), Emove (m1+m2) (m1+m2+1) ]

Figure 4: The definition of the function m_or
Implementing a machine of this sort is much simpler than for an general NFA: we only have to keep track of a single position. Is there a general mechanism for finding a DFA corresponding to a regular expression? In fact, there is a general technique for transforming an arbitrary NFA into a DFA, and this we examine now.

The conversion of an NFA into a DFA is based on the implementation given in Section 6. The main idea there is to keep track of a set of states, representing all the possible positions after reading a certain amount of input. This set itself can be thought of as a state of another machine, which will be deterministic: the moves from one set to another are completely deterministic.

We show how the conversion works with the machine $P$. The start state of the machine will be the closure of the set $\{0\}$, that is:

$$A = \{0,1,2,4\}$$

Now, the construction proceeds by finding the sets accessible from $A$ by moves on $a$ and on $b$ – all the characters in the alphabet of the machine $P$. These sets are states of the new machine; we then repeat the construction with these new states, until no more states are produced by the construction.

From $A$ on the symbol $a$ we can move to $3$ from $2$. Closing under $\epsilon$-moves we have the set $\{1,2,3,4,6,7\}$, which we call $B$

$$B = \{1,2,3,4,6,7\}$$

$$A \xrightarrow{a} B$$

In a similar way, from $A$ on $b$ we have

$$C = \{1,2,4,5,6,7\}$$

$$A \xrightarrow{b} C$$

Our new machine so far looks like

![Diagram](attachment:diagram.png)

We now have to see what is accessible from $B$ and $C$. First $B$.

$$D = \{1,2,3,4,6,7,8\}$$

$$B \xrightarrow{a} D$$

which is another new state. The process of generating new states must stop, as there is only a finite number of sets of states to choose from $\{0,1,2,3,4,5,6,7,8\}$. What happens with a $b$ move from $B$?
This gives the partial machine

Similarly,

\begin{align*}
  C & \xrightarrow{a} D \\
  C & \xrightarrow{b} C \\
  D & \xrightarrow{a} D \\
  D & \xrightarrow{b} C
\end{align*}

which completes the construction of the DFA

Which of the new states is final? One of these sets represents an accepting state exactly when it contains a final state of the original machine. For $P$ this is $B$, which is contained in the set $D$ only. In general there can be more than one accepting state for a machine. (This need not be true for NFAs, since we can always add a new final state to which each of the originals is linked by an $\varepsilon$-move.)

\section{10 Transforming NFAs to DFAs}

The Haskell code to covert an NFA to a DFA is found in the module \texttt{NfaToDfa}, and the main function is
make\_deterministic :: Nfa Int -> Nfa Int

make\_deterministic = number . make\_deter

A deterministic version of an NFA with numeric states is defined in two stages, using

make\_deter :: Nfa Int -> Nfa (Set Int)

number :: Nfa (Set Int) -> Nfa Int

make\_deter does the conversion to the deterministic automaton with sets of numbers as states, number replaces sets of numbers by numbers (rather than capital letters, as was done above). States are replaced by their position in a list of states – see the file for more details.

The function make\_deter is a special case of the function

deterministic :: Nfa Int -> [Char] -> Nfa (Set Int)

make\_deter mach = deterministic mach (alphabet mach)

The process of adding state sets is repeated until no more sets are added. This is a version of taking a limit, given by the nfa\_limit function, which acts as the usual limit function, except that it checks for equality of NFAs as collections of sets.

deterministic mach alpha
  = nfa\_limit (addstep mach alpha) startmach
    where
      startmach = NFA
        (sing starter)
        empty
        starter
        finish
      starter = closure mach (sing start)
      finish
        | (term 'inter' starter) == empty = empty
        | otherwise = sing starter
      (NFA sts mvs start term) = mach

The start machine, startmach, consists of a single state, the $\varepsilon$-closure of the start state of the original machine. addstep mach alpha takes a partially built DFA and adds the state sets of mach accessible by a single move on any of the characters in alpha, the alphabet of mach.
This involves iterating over the state sets in the partially built DFA, which is done using \texttt{addmoves}. \texttt{addmoves} \texttt{mach x alpha dfa} will add to \texttt{dfa} all the moves from state set \texttt{x} over the alphabet \texttt{alpha}.

\texttt{addmoves} :: \texttt{Nfa Int -> Set Int -> [Char] -> Nfa (Set Int) -> Nfa (Set Int)}

\texttt{addmoves mach x [] dfa} = dfa

\texttt{addmoves mach x (c:r) dfa}
  = addmoves mach x r (addmove mach x c dfa)

In turn, \texttt{addmoves} iterates along the alphabet, using \texttt{addmove}. \texttt{addmove mach x c dfa} will add to \texttt{dfa} the moves from state set \texttt{x} on character \texttt{c}.

\texttt{addmove} :: \texttt{Nfa Int -> Set Int -> Char -> Nfa (Set Int) -> Nfa (Set Int)}

\texttt{addmove mach x c (NFA states moves start finish)}
  = \texttt{NFA states’ moves’ start finish’}
    where
    \texttt{states’} = states ‘union’ (sing new)
    \texttt{moves’} = moves ‘union’ (sing (Move x c new))
    \texttt{finish’}
      = \texttt{empty} /= (term ‘inter’ new) \texttt{finish} ‘union’ (sing new)
      = \texttt{finish otherwise}
    \texttt{new} = onetrans mach c x
    \texttt{(NFA s m q term) = mach}

The new state set added by \texttt{addmove} is defined using the \texttt{onetrans} function first defined in the simulation of the NFA.
11 Minimising a DFA

In building a DFA, we have produced a machine which can be implemented more efficiently. We might, however, have more states in the DFA than necessary. This section shows how we can optimise a DFA so that it contains the minimum number of states to perform its function of recognising the strings matching a particular regular expression.

Two states \( q \) and \( r \) in a DFA are *distinguishable* if we can find a string \( s \) which reaches an accepting state from \( q \) but not from \( r \) (or vice versa). Otherwise, they can be treated as the same, because no string makes them behave differently — putting it a different way, no *experiment* makes the two different.

How can we tell when two states are different? We start by dividing the states into two partitions: one contains the accepting states, and the other the remainder, or non-accepting states. For our example, we get the partition

\[
\begin{align*}
I : & \quad D \\
II : & \quad A, B, C
\end{align*}
\]

Now, for each set in the partition, we check whether the elements in the set can be further divided. We look at how each of the states in the set behaves relative to the previous partition. In pictures,

\[
\begin{align*}
& A \xrightarrow{a} B (\text{II}) \quad B \xrightarrow{a} D (\text{I}) \quad C \xrightarrow{a} D (\text{I}) \\
& b \quad b \quad b
\end{align*}
\]

This means that we can re-partition thus:

\[
\begin{align*}
I : & \quad D \\
II : & \quad A \\
III : & \quad B, C
\end{align*}
\]

We now repeat the process, and examine the only set which might be further subdivided, giving
This shows that we don’t have to re-partition any further, and so that we can stop now, and collapse the two states $B$ and $C$ into one, thus:

The Haskell implementation of this process is in the module `MinimiseDfa`.

**Exercises**

13. For the regular expression $b(ab|ba)*a$, find the corresponding NFA.
14. For the NFA of question 1, find the corresponding (non-optimised) DFA.
15. For the DFA of question 2, find the optimised DFA.

### 12. Regular definitions

A *regular definition* consists of a number of *named* regular expressions. We are allowed to use the defined names on the right-hand sides of definitions *after* the definition of the name. For example,

- $\text{alpha} \rightarrow [a-zA-Z]$  
- $\text{digit} \rightarrow [0-9]$  
- $\text{alphanum} \rightarrow \text{alpha} \mid \text{digit}$  
- $\text{ident} \rightarrow \text{alpha} \mid \text{alphanum}^*$  
- $\text{digits} \rightarrow \text{digit}^*$  
- $\text{fract} \rightarrow (.\text{digits})?$  
- $\text{num} \rightarrow \text{digits} \text{fract}$
Because of the stipulation that a definition precedes the use of a name, we can expand each right-hand side to a regular expression involving no names.

We can build machines to recognise strings from a number of regular expressions. Suppose we have the patterns

\begin{align*}
p1: &\quad a \\
p2: &\quad abb \\
p3: &\quad a^*b^*
\end{align*}

We can build the three NFAs thus:

\begin{itemize}
  \item \begin{tikzpicture}
    \node (1) at (0,0) [shape=circle,draw] {1};
    \node (2) at (1,0) [shape=circle,draw] {2};
    \node (3) at (0,-1) [shape=circle,draw] {3};
    \node (4) at (1,-1) [shape=circle,draw] {4};
    \node (5) at (2,-1) [shape=circle,draw] {5};
    \node (6) at (3,-1) [shape=circle,draw] {6};
    \node (7) at (0,-2) [shape=circle,draw] {7};
    \node (8) at (1,-2) [shape=circle,draw] {8};
    \draw [->] (1) edge [bend left] node {a} (2);
    \draw [->] (2) edge [bend left] node {} (3);
    \draw [->] (3) edge [bend left] node {a} (4);
    \draw [->] (4) edge [bend left] node {b} (5);
    \draw [->] (5) edge [bend left] node {} (6);
    \draw [->] (3) edge [loop below] node {$\varepsilon$} (3);
    \draw [->] (4) edge [loop below] node {$\varepsilon$} (4);
    \draw [->] (5) edge [loop below] node {$\varepsilon$} (5);
  \end{tikzpicture}

  \item \begin{tikzpicture}
    \node (1) at (0,0) [shape=circle,draw] {1};
    \node (2) at (1,0) [shape=circle,draw] {2};
    \node (3) at (0,-1) [shape=circle,draw] {3};
    \node (4) at (1,-1) [shape=circle,draw] {4};
    \node (5) at (2,-1) [shape=circle,draw] {5};
    \node (6) at (3,-1) [shape=circle,draw] {6};
    \node (7) at (0,-2) [shape=circle,draw] {7};
    \node (8) at (1,-2) [shape=circle,draw] {8};
    \draw [->] (1) edge [bend left] node {a} (2);
    \draw [->] (2) edge [bend left] node {} (3);
    \draw [->] (3) edge [bend left] node {a} (4);
    \draw [->] (4) edge [bend left] node {b} (5);
    \draw [->] (5) edge [bend left] node {} (6);
    \draw [->] (3) edge [loop below] node {$\varepsilon$} (3);
    \draw [->] (4) edge [loop below] node {$\varepsilon$} (4);
    \draw [->] (5) edge [loop below] node {$\varepsilon$} (5);
  \end{tikzpicture}
\end{itemize}

and then they can be joined into a single machine, thus

\begin{itemize}
  \item \begin{tikzpicture}
    \node (0) at (0,0) [shape=circle,draw] {0};
    \node (1) at (1,0) [shape=circle,draw] {1};
    \node (2) at (2,0) [shape=circle,draw] {2};
    \node (3) at (0,-1) [shape=circle,draw] {3};
    \node (4) at (1,-1) [shape=circle,draw] {4};
    \node (5) at (2,-1) [shape=circle,draw] {5};
    \node (6) at (3,-1) [shape=circle,draw] {6};
    \node (7) at (0,-2) [shape=circle,draw] {7};
    \node (8) at (1,-2) [shape=circle,draw] {8};
    \draw [->] (0) edge [bend left] node {$\varepsilon$} (1);
    \draw [->] (1) edge [bend left] node {a} (2);
    \draw [->] (2) edge [bend left] node {} (3);
    \draw [->] (3) edge [bend left] node {a} (4);
    \draw [->] (4) edge [bend left] node {b} (5);
    \draw [->] (5) edge [bend left] node {} (6);
    \draw [->] (3) edge [loop below] node {$\varepsilon$} (3);
    \draw [->] (4) edge [loop below] node {$\varepsilon$} (4);
    \draw [->] (5) edge [loop below] node {$\varepsilon$} (5);
  \end{tikzpicture}

  \item \begin{tikzpicture}
    \node (0) at (0,0) [shape=circle,draw] {0};
    \node (1) at (1,0) [shape=circle,draw] {1};
    \node (2) at (2,0) [shape=circle,draw] {2};
    \node (3) at (0,-1) [shape=circle,draw] {3};
    \node (4) at (1,-1) [shape=circle,draw] {4};
    \node (5) at (2,-1) [shape=circle,draw] {5};
    \node (6) at (3,-1) [shape=circle,draw] {6};
    \node (7) at (0,-2) [shape=circle,draw] {7};
    \node (8) at (1,-2) [shape=circle,draw] {8};
    \draw [->] (0) edge [bend left] node {$\varepsilon$} (1);
    \draw [->] (1) edge [bend left] node {a} (2);
    \draw [->] (2) edge [bend left] node {} (3);
    \draw [->] (3) edge [bend left] node {a} (4);
    \draw [->] (4) edge [bend left] node {b} (5);
    \draw [->] (5) edge [bend left] node {} (6);
    \draw [->] (3) edge [loop below] node {$\varepsilon$} (3);
    \draw [->] (4) edge [loop below] node {$\varepsilon$} (4);
    \draw [->] (5) edge [loop below] node {$\varepsilon$} (5);
  \end{tikzpicture}
\end{itemize}

In using the machine we look for the longest match against any of the patterns:

\begin{align*}
0, &\quad 1, 3, 7, 8(p3) \\
a &\quad 2(p1), 4, 7, 8(p3) \\
a &\quad 7, 8(p3) \\
b &\quad 8(p3) \\
a &\quad -
\end{align*}
In the example, the segment of `a\ab` matches the pattern `p3`.

**Exercises**

16. Fully expand the names `digits` and `num` given above.

17. Build a Haskell program to recognise strings according to a set of regular definitions, as outlined in this section.

**Bibliography**
