Programming Languages and Computation

Problem Sheet 3: Regular Expressions

We originally motivated finite automata as a very simple theoretical model for computation by physical machines. However, like most theory in computer science, it also has a very wide range of practical applications. Probably the best known application of finite automata is in regular expressions and pattern matching, which is what we implement in this problem sheet.

You may think to yourself, "why should we be interested in such a restricted kind of computational system? If we want to solve real problems, we have much more sophisticated means to describe algorithms, like programming languages." One answer is that, by imposing restrictions on the way we write out algorithms, we can get back desirable guarantees about them.

We do this all the time when we write programs in statically typed programming languages (those programming languages where type errors will prevent you from compiling). There is no need to use a static type system – you could write your programs in Python, Ruby or Scheme – and the type system indeed restricts what you can write. By way of a silly example, in almost all mainstream typed programming languages, the type system will prevent you from writing `(if true then 1 else "hello") + 1`. Why should this be disallowed? It is a perfectly good program expression and patently equivalent to `2`, which you certainly can write! However, by swallowing the type system pill you get back a desirable guarantee about your program: it is guaranteed not to crash at run time due to a mismatch between functions and their arguments.

So too you can obtain a desirable guarantee if you swallow the (admittedly far more bitter pill) of expressing your algorithm as deterministic finite automaton. Such an algorithm will be guaranteed to run online[1], in worst-case linear time and with *constant* space.

One area where these algorithmic complexity guarantees are especially important, and where automata are anyway a natural choice, is in pattern matching on strings. However, actually writing down a finite automaton is a bit clumsy and rather difficult to do in a programmers text editor. What is needed is a convenient syntax for finite automata, a way that we can write them down and combine them as we do with numbers in arithmetic. These requirements are satisfied by the language of *regular expressions*, and they are the standard interface to string matching in programming languages.

They are also the topic of this sheet, you will be working with reg/src/Automata.hs and reg/src/RegExp.hs.

Regular expressions

Regular expressions are a syntax for specifying languages of strings by describing patterns that the strings must match.

Like arithmetic expressions, they directly denote (stand for) values, but rather than denoting numerical values, they denote languages over some given alphabet Σ. Also like arithmetic expressions, they are built by taking some primitive values and combining them with certain operators. These primitives and operators are given in Figure 1.

[1] This means, roughly, that it can process its input in a stream, bit-by-bit, rather than having to wait until all input is available.

- The symbol $\emptyset$ is a regular expression. It denotes the empty language that contains no strings.
- The symbol ϵ is a regular expression. It denotes the language $\{\epsilon\}$ that consists solely of the empty string.
- Every letter a of the alphabet Σ is itself a regular expression. It denotes the language $\{a\}$ that contains only the word of one letter: a.
- Given two regular expressions e_1 and e_2, we can form their concatenation $e_1 \cdot e_2$. This expression denotes the language of all words of shape $w \cdot v$ where w is a word in e_1 and v is a word in e_2.
- Given two regular expressions e_1 and e_2, we can form their alternation $e_1 \mid e_2$. This expression denotes the language of all words that are either in e_1 or are in e_2.
- Given a regular expression e, we can form the Kleene (pronounced clay-knee) star e^*. This expression denotes the language of all words formed by concatenating finitely many (including 0) copies of words from e. For example, {}

Figure 1: Syntax and semantics of regular expressions.

Like arithmetic expressions, there is ambiguity when we nest operators without using parentheses, and there is a need to reduce syntactic clutter. So, let's agree some conventions:

- Kleene star binds tightest, so $e_1^* \mid e_2$ should be read as $(e_1^*) \mid e_2$.
- Concatenation binds tighter than alternation, so $e_1 \mid e_2 \cdot e_3$ should be read as $e_1 \mid (e_2 \cdot e_3)$.
- Concatenation can be abbreviated by juxtaposition, so we can write $(abc)^* \mid e$ instead of $(a \cdot b \cdot c)^* \mid e$.
- We can write Σ as an abbreviation for the nested alternation of all letters of the alphabet. For example, if $\Sigma = \{a, b, c\}$ then we can write Σ^* instead of $(a \mid b \mid c)^*$.
- We can write e^+ as an abbreviation for ee^*, in other words, to denote the language of all words formed from at least one copy of words in e. Let's agree that $\_^+$ binds as tightly as $\_^*$.

Here are some examples over the alphabet $\Sigma = \{0, 1\}$ (taken from Sipser's "Introduction to the

Theory of Computation", 3rd edition, pp 65 – note that the syntax is slightly different):

0^*10^*	$\{w \mid$ w contains a single 1 $\}$
$\Sigma^*1\Sigma^*$	$\{w \mid$ w contains at least one 1 $\}$
$\Sigma^*001\Sigma^*$	$\{w \mid$ w contains 001 as a substring $\}$
$1^*(01^+)^*$	$\{w \mid$ every 0 in w is followed by at least one 1 $\}$
$(\Sigma\Sigma)^*$	$\{w \mid$ w is a string of even length $\}$
$(\Sigma\Sigma\Sigma)^*$	$\{w \mid$ the length of w is a multiple of 3 $\}$
$01 \mid 10$	$\{01, 10\}$
$0\Sigma^*0 \mid 1\Sigma^*1 \mid 0 \mid 1$	$\{w \mid$ w starts and ends with the same symbol $\}$
$(0 \mid \epsilon)1^*$	$\{w \mid$ w is either of shape $0v$ or v for some $v \in \{1\}^*$ $\}$
$(0 \mid \epsilon)(1 \mid \epsilon)$	$\{\epsilon, 0, 1, 01\}$
$1^*\emptyset$	$\emptyset$
$\emptyset^*$	$\{\epsilon\}$

The file reg/src/RegExp.hs contains a datatype `RegExp` for regular expressions over the alphabet **`Char`**, as well as a `Show` instance for printing. For convenience, there is an `IsString` instance too. This instance allows for our abbreviation of concatenation by juxtaposition: writing `"abc"` in a context of type `RegExp` will evaluate to:

```
((Sym 'a' `Cat` Sym 'b') `Cat` Sym 'c') `Cat` Eps
```

However, you will need to set the corresponding extension to use this feature in GHCI:

```
λ :set -XOverloadedStrings
```

* 1. Implement the each of the following regular expressions (a)–(e) as the corresponding Haskell value `exprA` – `exprE`. Although the expressions are defined generally over the whole of the type **`Char`**, assume that $\Sigma = \{0, 1\}$.

(a) 0^*10^*

(b) $\Sigma^*001\Sigma^*$

(c) $(\Sigma\Sigma)^*$

(d) $(0 \mid \epsilon)1^*$

(e) $(0 \mid \epsilon)(1 \mid \epsilon)$

** 2. Implement the function `alts` which, given a non-empty string (list of characters), constructs a regular expression that matches any one of the characters exactly once (i.e. a large alternation). Then design regular expressions for each of the following languages over the alphabet of all **`Char`**:

(a) `exprUN`: the language of valid Bristol University usernames

(b) `exprTime`: the language of valid 24 hour clock times in format HH:MM

(c) `exprIPv4`: the language of valid IPv4 addresses written in decimal

You will have an opportunity to debug your answers shortly.

Regular expressions and finite automata are *equi-expressive*: they define *exactly the same class of languages*. For every regular expression e we can find a finite automaton M such that the language denoted by e is exactly $L(M)$. Conversely, for every finite automaton M, we can find a regular expression e to denote $L(M)$. The goal of this problem sheet is to implement the translation from regular expressions to automata, so first we need to consider how to represent automata in Haskell and compute with them.

The file reg/src/Automata.hs contains a definition for a type `Auto a q` of nondeterministic finite automata. The definition of an NFA requires that we give 5 components: the finite set of *states*, the finite *alphabet*, the *transition function*, the *initial state*, and the set of *final states*.

You can see all of these components in our type of NFAs, `Auto a q`, but we have chosen for some components to be represented only as a type, while others are values.

```
data Auto a q =
  MkAuto {
    start  :: q,
    trans  :: [(q, Label a, q)],
    final  :: [q]
  } deriving (Show)
```

A value of type `Auto a q` is a record with three fields, `start`, `trans` and `final` which specify the initial state, the transition function and the set of final states respectively. The record itself is created using the constructor `MkAuto`. The type is parametrised by the type `a` of the alphabet used and the type `q` of the states. An automaton transition function $\delta : Q \times \Sigma_\epsilon \to \mathscr{P}(Q)$ is represented a listing of triples (q, a, q') that record which inputs (q, a) to δ are related to which possible outputs $q' \in \delta(q, a)$. When some type a represents the alphabet Σ of the automaton, the type `Label a` represents the type Σ_ϵ, in other words the "labels" on the transitions of the automaton diagram. A value of type `Label` is either a letter x from the alphabet a, represented as `L x` or is epsilon, represented as `E`. A more precise specification is given in Figure 2, see if you can understand it with reference to the three examples `ev`, `zzo` and `evzzo`.

The file reg/src/Vis.hs gives some utility functions for drawing diagrams of automata and transition systems.

You will need graphviz installed on your system in order to use these functions. I suggest one of the following:

- On Linux, or Windows WSL2, use your package manager: e.g. `sudo apt-get install graphviz`
- On Mac, use homebrew: `brew install graphviz`
- On Windows (not WSL2), use chocolatey (from a privileged console): `choco install graphviz`

A value `m :: Auto a q` represents an automaton $M = (Q, \Sigma, \delta, q_0, F)$ just if:

- each state of Q is represented by some value of type `q`,
- each letter of Σ is represented by some value of type `a`

and, according to this representation scheme:

- the state $q_0 \in Q$ is represented by the value `start m :: q`
- the final states $F \subseteq Q$ are in 1-1 correspondence with the entries in the list `final m :: List q`
- the transition function δ of M and `trans m` are related by:

  `(q, E, q') `elem` trans m == True` *iff* $q' \in \delta(q, \epsilon)$

  `(q, S a, q') `elem` trans m == True` *iff* $a \in \Sigma$ and $q' \in \delta(q, a)$

Figure 2: Representation of finite automata.

You can verify your installation by running `dot -V` from the terminal. If the executable cannot be found, please adjust your path as usual.

If you are not able to install graphviz, it is not essential, you can skip this part and move on to the next question. Otherwise, import `Vis` into your current GHCI session and use `vizAuto` to create a png image of the diagram of the automaton `evzzo` in your working directory. You will need to import Automaton too, if evzzo is not already in scope.

```
λ import Vis
λ import Automaton
λ vizAuto evzzo "evzzo.png"
"evzzo.png"
```

The function `vizTrSys :: (Ord a, Show a) => TrSys a → Set a → FilePath → IO FilePath` can be used to draw a similar diagram for a transition system but only a finite part at a time. The first argument is the (possibly infinite) transition system and the second is a finite subset of the configurations to draw. The function performs an IO action to draw that part of the transition system that is wholly contained inside the finite subset. You may find these functions useful for debugging during the next exercises.

```
λ let reach234 = reachable' chameleons (2,3,4)
λ vizTrSys chameleons reach234 "chameleons-2-3-4.png"
"chameleons-2-3-4.png"
```

** 3. Implement the function `autoTrSys` which, given an NFA `m` of type `Auto` generates the infinite transition system that describes computation in `m` according to the definition. You will find the function **`next`** helpful since, for a given automaton `m` and state `p` it returns the set of all the states that can be reached from `p` in one step, along with the letter (or epsilon) that is required to get there e.g.

```
λ next zzo 0
fromList [('0',0),('0',1),('1',0)]
λ next ev Odd
fromList [('0',Odd),('1',Even)]
λ next evzzo Init
fromList [( ,Even),( ,0)]
```

You can test your implementation by running the *A1: Automaton Transition System Tests*:

```
$ cabal test --test-show-detail=direct --test-option="--pattern=A1"
Running 1 test suites...
Test suite test: RUNNING...
```

** 4. Implement the function `member` which, given an automaton `m` and a word `w` determines if `w` is accepted by `m`. You can test your implementation using *A2: Membership Tests*.

Now we come to the method for translating each regular expression into an equivalent finite automaton (i.e. that recognises exactly the language denoted by the expression). Since our type of regular expressions has fixed the alphabet **Char**, the corresponding automata will also have this alphabet. It is not important how we label the states of the new automata, but our algorithm for performing the translation will be constructing states as it goes along, so we need to make sure we don't accidentally label two states with the same name. A simple way to guarantee this is to use integers for the names of the states (rather than creating states with names q_0, q_1, q_2 etc, we create states with names 0, 1, 2 etc.) and to keep track of which numbers have already been used when creating new automata.

A start is made in the given function `compile :: RegExp → Int → (Int, Auto Char Int)`, which is defined in reg/src/RegExp.hs. This function pattern matches on its regular expression argument and then delegates control to one of several functions that cover each of the cases[2], passing the arguments to the constructor that was matched on. The first argument `rex` is the regular expression to translate to an automaton and the second is the first natural number `n` that has not yet been used as the name of an automaton state. The idea is that the function constructs a new automaton that is guaranteed to recognise the language denoted by `rex` and, in the process of doing so, generates names for the states of that automaton starting from n. The output is a pair consisting of the constructed automaton and the first number out of `[n,n+1,n+2,...]` that it did *not* use as the name of a state. For example, if `compile rex 4` constructs an automaton `m` with 6 states, then the states will be named `4`, `5`, `6`, `7`, `8`, `9` and the output will be the pair `(10,m)`.

This can be seen in the given function `compileAlt`, which begins construction of an automaton equivalent to `Alt e1 e2` by recursively computing automata equivalent to `e1` and `e2`.

[2]Ordinarily you would probably deal with each case directly, inside the `compile` function, but this setup is more convenient as a problem sheet question.

```
compileAlt e1 e2 n0 = (n2+3, m)
 where
  -- Recursively compute automata
  -- corresponding to /e1/ and /e2/.
  (n1,m1) = compile e1 n0
  (n2,m2) = compile e2 n1
```

To make sure that these two recursive sub-computations do not accidentally share names, we pass the output number, `n1`, from the compilation of `e1` as the input number for the compilation of `e2` and when we need to generate a name for a state in the computation after this point, we start from the output `n2` of compiling `e2`.

*** 5.

(a) Design and implement the other cases in the compilation of regular expressions to finite automata.

(b) The function `match`, which you are given, brings together your work from Q3, Q4 and Q5 to implement pattern matching.

```
match :: RegExp → String → Bool
match rex = member (snd (compile rex 0))
```

It is used to define two groups of tests: *R1: Sipser Tests*, which use your definitions from Q1, and then the tests *R2: Star Free Tests*, which use your definitions from Q2. Check that all the tests are passed.

** 6. Design regular expressions `cIdentLex`, `intLitLex` and `floatLex` for the following programming language lexemes:

(a) A *C program identifier* is any string of length at least 1 containing only letters ('a'–'z', lower and uppercase), digits ('0'–'9') and the underscore, and which begins with a letter or the underscore.

(b) In most programming languages, *integer literals* can be written in:

- decimal – as a non-empty sequence of digits
- hexadecimal – as a non-empty sequence of characters from '0'–'9','a'–'e' (upper or lowercase) that are preceded by "0x"
- binary – as a non-empty sequence of bits '0' and '1' that are preceded by "0b"

(c) Haskell *floating point literals* were described in Sheet 2, Q3.

Optional extension on programming with nondeterminism

Finally, I want to give you an example of programming with nondeterminism, using the guess-and-verify paradigm, in a real programming language.

In the file reg/src/NonDet.hs you can find the definition of a list `prop` of satisfying assignments to a simple propositional formula. Neither the formula nor its list of satisfying assignments (those assignments of truth values to the propositional formula that make it true) is remarkable, but the way that we express how they are computed is!

```
prop :: [(Bool,Bool,Bool,Bool,Bool)]
prop =
 do p <- bools
    q <- bools
    r <- bools
    s <- bools
    t <- bools
    guard ((p && (q `implies` r) || not s) && not t)
    return (p,q,r,s,t)

 where
  True `implies` False = False
  _    `implies` _     = True
```

Here we are using the list monad, which is an extremely simply way to express nondeterminism in Haskell. The way to read the program is as follows. Binding a value to a variable corresponds to making a nondeterministic choice. For example, given that `bools = [True, False]`:

```
q <- bools
```

should be thought of as nondeterministically choosing a boolean (i.e. one of `True` or `False`). In the first part of the program, we nondeterministically *guess* assignments of booleans to `p`, `q`, `r`, `s` and `t` that will make the formula true. Then, in the second part, the function `guard` is used in order to *verify* that the assignment does indeed make the formula true – that the guess was correct. Then, we simply return the correct guess.

**** 7. (Optional)

(a) By looking at the definitions of `guard` and the monad instance for `List`, try to understand the workings of the `prop` program.

The SEND-MORE-MONEY puzzle is to find an assignment of *distinct* digits 0–9 to the letters S, E, N, D, M, O, R and Y such that the following equation is satisfied:

$$\begin{array}{cccccc} & S & E & N & D \\ + & M & O & R & E \\ \hline M & O & N & E & Y \end{array}$$

Note the distinctness requirement: no two distinct letters may have the same digit assigned.

(b) Implement a solution to SEND-MORE-MONEY using nondeterminism.