Introduction to Language Theory and Compilation Exercises

Session 2: Regular expressions

Reminders

Regular expressions (RE)

Finite automata (FA) are an equivalent formalism to regular languages (RL) (for each regular language, there exists at least one FA that recognizes it, and each FA recognizes a RL). RE are another formalism defined inductively just as RL. It can be proven that RE and RL are equivalent. Moreover, a RE is equivalent to *one and one only* RL, but a RL can have more than one corresponding RE.

Table 1 shows the basics case of the RE formalism and Figure 2 shows operators applied on the two RE p and q.

RE	language
ϕ	$\emptyset$
ε	$\{\varepsilon\}$
$a \quad (\forall a \in \Sigma)$	$\{a\}$

Table 1: Base cases

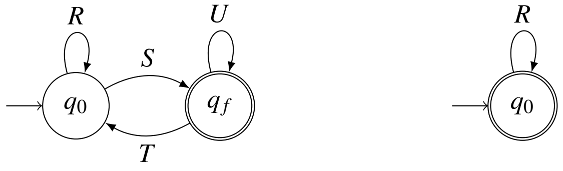
RE	language
$p+q$	$P \cup Q$
pq (or $p \cdot q$)	$P \cdot Q$
p^*	P^*

Table 2: RE operators

For instance, the RE $a(b+c)$ denotes the language $\{a\} \cdot \{b,c\} = \{ab, ac\}$ which could also be denoted by $ab + ac$.

State elimination method

Given a DFA M, we can craft a corresponding regular expression using the *state elimination* method. The general idea is to label transitions in the automaton using RE, pick a final state, then remove all other states step by step to finally reach a simple automaton which can then be used to easily determine a RE. The process terminates in a two or one state automaton of the forms shown in Figure 1 with the corresponding REs, depending on whether the initial state is also a final state.



Corresponding RE: $(R+SU^*T)^*SU^*$ Corresponding RE: R^*

Figure 1: The two possible forms for an automaton A_{q_f} obtained by eliminating all states but q_0 and q_f, and their corresponding regular expressions. We obtain the right automaton whenever $q_0 = q_f$.

For each final state $q^F \in F$, one has to build such a simple automaton to derive a regular expression $RE(q^F)$ that expresses all possible inputs that are accepted when M stops in q^F. The actual regular expression that describes the language $L(M)$ of the automaton M then simply becomes:

$$RE(q_1^F) + RE(q_2^F) + \ldots + RE(q_k^F) \qquad \text{where } \{q_1^F, \ldots, q_k^F\} = F$$

Algorithm

1. Preprocess by labelling all transitions by a RE.
2. for each state q_x to be eliminated, consider each transition (q_a, q_x), (q_x, q_b) or (q_x, q_x) with respective labels A, B and X.
3. The transition (q_a, q_b) labelled by E becomes the absorbing transition $E + (AX^*B)$ and remove A, B, X and E.

Note: some transitions can be null. In that case, do not consider the transition. For instance, if $E = (q_a, q_b)$ cannot be generated by δ (the transition function, see definition), then the absorption transition will be AX^*B.

Extended regular expressions (ERE)

The ERE syntax shown in Table 3, is very popular and grants more flexibility than traditional RE.

Expression	Accepted language
`r*`	0 or more `r`s
`r+`	1 or more `r`s
`r?`	0 or 1 `r`
`[abc]`	`a` or `b` or `c`
`[a-z]`	Any character in the interval `a`...`z`
`.`	Any character except `\n`
`[^s]`	Any character but those in `s`
`r{m,n}`	Between `m` and `n` occurrences of `r`

Expression	Accepted language
`r1 r2`	The concatenation of `r1` and `r2`
`r1 | r2`	`r1` or `r2`
`(r)`	`r`
`^r`	`r` if it starts a line
`r$`	`r` if it ends a line
`"s"`	The string `s`
`\c`	The character `c`
`r1(?=r2)`	`r1` when it's followed by `r2`

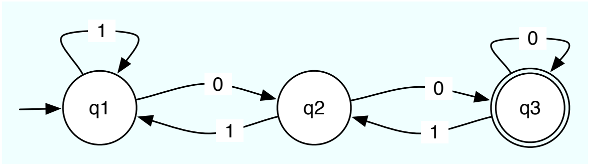
Table 3: ERE syntax

Exercises

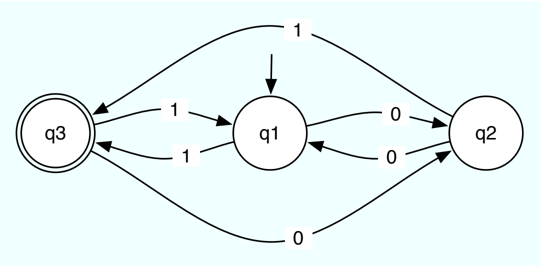
Ex. 1. For each of the following languages (defined on the alphabet $\Sigma = \{0, 1\}$), design a regular expression that recognizes it:

1. The set of strings ending with 00.
2. The set of strings whose 10th symbol, counted from the end of the string, is a 1.
3. The set of strings where each pair of zeroes is followed by a pair of ones.
4. The set of strings not containing 101.
5. The set of binary numbers divisible by 4.

Ex. 2. For each of the following DFAs, give a regular expression accepting the same language:



(a)



(b)

Ex. 3. Convert the following REs into ε-NFAs:

1. 01^*
2. $(0+1)01$
3. $00(0+1)^*$

Ex. 4.

1. Give an extended regular expression (ERE) that targets any sequence of 5 characters, including the newline character `\n`.
2. Give an ERE that targets any string starting with an arbitrary number of \ followed by any number of `*`.
3. UNIX-like shells (such as `bash`) allow the user to write *batch* files in which comments can be added. A line is defined to be a comment if it starts with a `#` sign. What ERE accepts such comments?
4. Design an ERE that accepts numbers in scientific notation. Such a number must contain at least one digit and has two optional parts:
   - A "decimal" part : a dot followed by a sequence of digits
   - An "exponential" part: an `E` followed by an integer that may be prefixed by `+` or `-`
   - Examples : `42`, `66.4E-5`, `8E17`, ...
5. Design an ERE that accepts "correct" phrases that fulfill the following criteria:
   - No prepending/appending spaces
   - The first word must start with a capital letter
   - The phrase must end with a dot `.`
   - The phrase must be made of one or more words (made of the characters `a...z` and `A...Z`) separated by a single space
   - There must be one sentence per line

   Punctuation signs other than a dot are not allowed.
6. Craft an ERE that accepts old school DOS-style filenames (8 characters in `a...z`, `A...Z` and `_`) whose extension is `.ext` and that begin with the string `abcde`. We ask that the ERE only accept the filename *without the extension*! Example: on `abcdeLOL.ext`, the ERE must accept `abcdeLOL`.