Assignment 12, Authomata Theory

Oleg Sivokon

<2015-09-07 Mon>

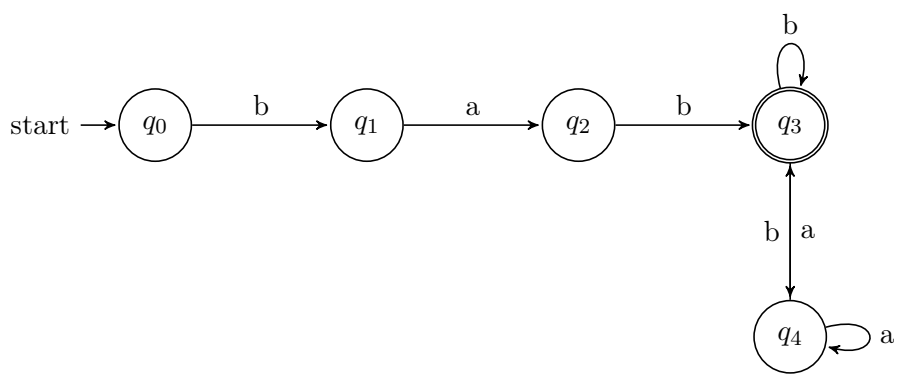
Contents

1 Problems

1.1 Problem 1

1. Build an NFA for the language over alphabet $\{a, b\}$ where words must start with bab and end in b.
2. Build an NFA for the language over alphabet $\{a, b, c\}$, defined as follows: $L = \{w \mid \exists n, m, k \in \mathbb{N}.(w = a^n b^m c^k \wedge |w| \mod 2 = 0)\}$.
3. Build an NFA for the language over alphabet $\{a, b\}$, s.t. it contains all words with substring aba repeated at least twice.

1.1.1 Answer 1



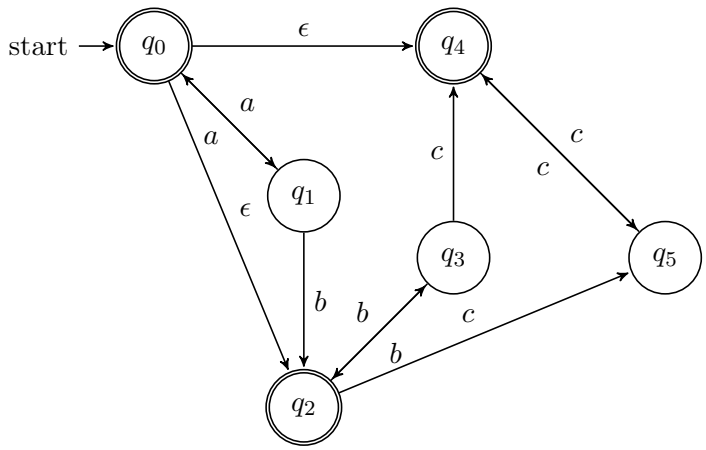
The nodes where automaton dies are not shown.

The language accepted by this automaton must start with the prefix bab as can be seen in diagram above. The only accepting state has transitions pointing at it only on inputs b, thus all words accepted by this language must end in b.

Conversely, if the words start with the prefix bab, then we must reach the statate $\hat{\delta}(bab, q_0) = q_4$. From q_4 the input can be either accepted, since it already ends in any number of b, or it may follow through to q_4 and whenever b is encountered in the input—return to q_3. Since all execution path will thus lead to the accepting state on b or to rejecting state on a, I conclude that all words with prefix bab and ending in b are accepted by the described automaton.

1.1.2 Answer 2

A regular expression to summarize the effort: $(((aa)^*((bb)^*(cc)^*) + (b(bb)^*c(cc)^*))) + (a(aa)^*((b(bb)^*(cc)^*) + (b(bb)^*(cc)^*)))$.

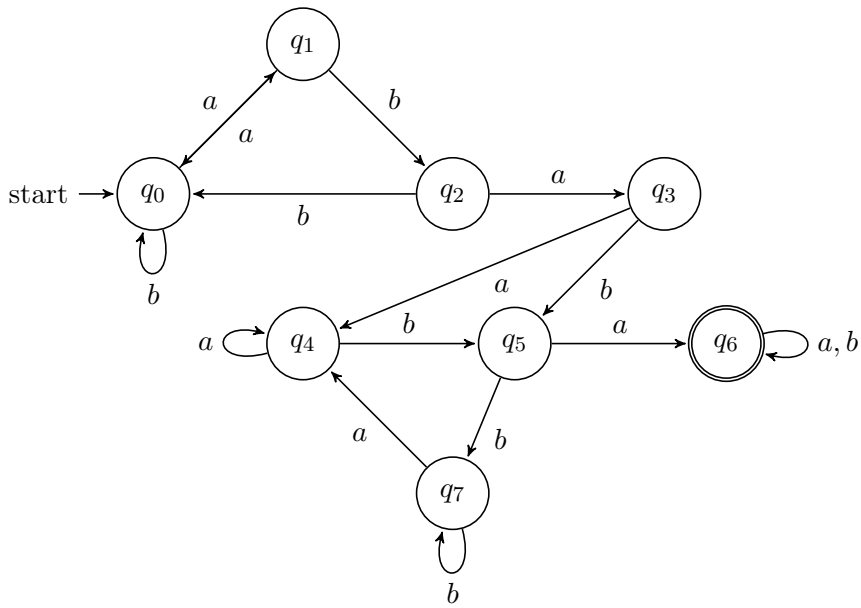


Essentially, what happens in this diagram is as follows:

1. On first input we decide whether the prefix starts with a, b or c.
2. Once decided, we parse more of the prefix.
   - If the prefix was c, we make sure there is even number of c.
   - If the prefix started with b, we create two branches, one will pick the non-accepting state of the bit of automaton processing c prefix once we counted even number of b inputs, and an accepting state otherwise.
   - If our prefix was a, then we will proceed similarly to the previous case, however, we will switch roles: on odd number of $a's$, we will switch to an accepting state and to a non-accepting state otherwise.

1.1.3 Answer 3

The diagram below can be also given by the regular expression: $((ab)^*b^*)^*aba(ba + (((ab)^*b^*)aba))(a + b)^*$. An easier, but a sloppier way to write the same regexp would be: $(a + b)^*(ababa + (aba(a + b)^*aba))(a + b)^*$.



Put in words: skip over repetitions of bb possibly preceded by a, until encountering aba substring. Once that happens, consider the prefix of the second aba substring found. If the next input is b, continue matching, else—bail out and essentially repeat the previously described procedure.

1.2 Problem 2

Prove or disprove that pairs of regular expressions to follow accept the same language.

1. $(0(10^*)^*)^* + 1^*$ and $(1+0)^*$.
2. $(1+0)^+$ and $(0^*1)^*(1^*0)^+ + (0^*1)^+$.
3. $1^*(0^*10^*)^*$ and $(101^*)^*1^*$.

1.2.1 Answer 3

Two expressions are not equivalent. $(1+0)^*$ matches any binary string, while $(0(10^*)^*)^* + 1^*$ doesn't match any binary string containing with a prefix 00 or more consequtive zeros.

1.2.2 Answer 4

Two expressions are equivalent. $(0^*1)^+$ will match any binary at least one character long string edning in 1, while $(1^*0)^+$ will match any binary string at least one character long ending in

0. The union of these two expressions will match all binary strings of length at least 1, which is equivalent to $(1+0)^+$. The $(0^*1)^*$ of the second expression plays no role (is redundant).

1.2.3 Answer 5

Teo expressions are not equivalent. $(101^*)^*1^*$ will not match string containing 00 or more consequent zeros as a substring, while this is not a problem for $1^*(0^*10^*)^*$.

1.3 Problem 3

Write a regular expression for the language over alphabet $\{a, b\}$ s.t. all words in this language start with either aa or bbb and none of them contains substring bab.

1.3.1 Answer 6

The desired regex is $(bbb^+)^*(aa^+b^*)^*$.

1.4 Problem 4

Write an algorithm which accepts a regular expression r and produces a language $\overline{L[r]}$.

1.4.1 Answer 7

The basic idea is to convert given regular expression to NFA, from NFA to DFA, switch roles of accepting and rejecting states, then convert the DFA into a regular expression again. I wrote concerte implementation for this algorithm. The documentation to my code is given in 2, the code itself can be found together with this document (it is not included for brevity).

String to regexp This step requires writing a recursive parser (since we need to balance parenthesis). Given a string containing regular expression this step produces an AST of regular expression code (further AST).

Regexp to NFA This step starts off with creating a starting state and a distinct accepting state. It recursively processes every node of AST and for each one of four node types it appends nodes to NFA:

terminal No nodes are added, only an arc between two active nodes.

concatenation One node is added between two currently active nodes, and each node is processed further with the added node as either its source or its destination.

union No nodes are added. This is similar to the `terminal` step, except both regexp are expanded further.

star Add ϵ-transition from both the source and the destination nodes to the node currently processed. Add ϵ-transition from the destination to the source.

NFA to DFA At first, arrange all transitions into a matrix $M_{i,q}$ indexed by inputs i, including ϵ and states q. Create a new matrix $M'_{j,p}$ indexed by inputs concatenated to ϵ^* and new states p. New states are obtained as follows: Select a cell $m = M_{i,q}$, m will be the set of all states reacheable on input i from state q, suppose $m = \{q_n, q_{n+1}, \dots q_{n+m}\}$. Now, for each $p \in \{q_n, \dots q_{n+m}\}$ find the cell $e = M_{\epsilon,p}$. The union of these cells is the label of target DFA.

Flip rejecting and accepting states This step is trivial: make switch the roles of all states. Note that this requires the often omitted "dead" state.

DFA to regexp For each state S of the DFA record the union of all states immediatel reacheable from the given state in a form of a grammar rule $S_a \to iS_b$. Substitute rules into each other to eliminate non-terminals as follows:

- $S_n \to i\{\epsilon\} \implies S_n \to i$.
- $S_n \to iS_n \implies S_n \to i^*$.
- $S_n \to iS_m, S_m \to jS_k \implies S_n \to ijS_k$.
  It is useful to apply set-theoretic identities, such as distributivity of union over concatenation, eg. $xz \cup yz = (x \cup y)z$ to produce a better regular expression.

Serialize regexp to string Recursively visit every node of AST and substitue the node contents with its string representation.

Example code inverts a regular expression $x(y+x)^* + z$. However, the current version of this code lacks the ability to optimize the produced regular expressions.

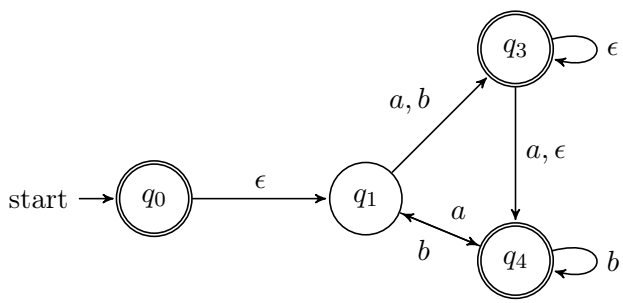
```
:- use_module(automata).

replace_tex(Out) -->
    [], { Out = "" } ;
    "*", replace_tex(Y),
    { string_concat("^*", Y, Out) } ;
    [603], replace_tex(Y),
    { string_concat("\\epsilon", Y, Out) } ;
    [X], replace_tex(Y),
    { text_to_string([X], Xs), string_concat(Xs, Y, Out) }.

assignment_12a :-
    invert_regex(`x(y+x)*+z`, Regex),
    string_codes(Regex, Codes),
    phrase(replace_tex(X), Codes),
    format('$$~w$$', [X]).
```

$$(((zz + (zy + (zx + (yz + (yy + (yx + xz)))))) + ((\epsilon + y) + ((zz + (zy + (zx + (yz + (yy + (yx + xz)))))((z + (y + x)))^* + ((xy + xx)z + (xy + xx)((y + x))^*)))) + ((xy + xx)zz + ((xy +$$

$xx)zy + (xy + xx)zx))(\epsilon + ((z + (y + x)))^*))$ $(((zz + (zy + (zx + (yz + (yy + (yx + xz)))))) + ((\epsilon + y) + ((zz + (zy + (zx + (yz + (yy + (yx + xz)))))))((z + (y + x)))^* + ((xy + xx)z + (xy + xx)((y + x))^*)))) + ((xy + xx)zz + ((xy + xx)zy + (xy + xx)zx))(\epsilon + ((z + (y + x)))^*))$

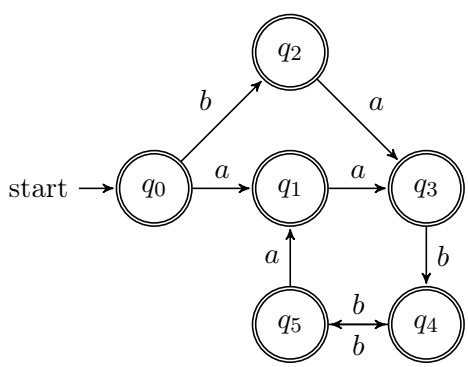
1.5 Problem 5

Build a DFA from given NFA:



1.5.1 Answer 8

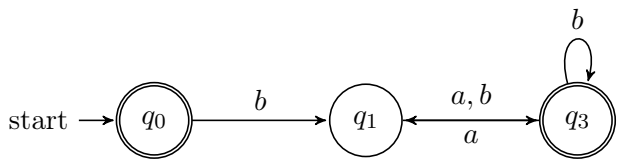
The corresponding DFA can be written as:



Nodes where automata dies are not shown.

1.6 Problem 6

Write a regular expression for the diagram below:



Nodes where the automata dies are not shown.

1.6.1 Answer 9

The regular expression for the diagram above: $\epsilon + b((a+b)b^*)^+$.

1.7 Problem 7

Given regular expression r and L, a language over Σ which designates regular expression $r\Sigma^*$. Prove that unless $L = \Sigma^*$ and $L = \emptyset$, there doesn't exist a regular expression s s.t. $s\Sigma^*$ designates $\overline{L}$.

2 Appendix A

2.1 automata.pl: High-level predicates for dealing with regular expressions

`https://github.com/wvxvw/intro-to-automata-theory`

This module defines predicates for searching and replacing in strings using regular expressions.

match_regex(*+Regex, +String***)** *[det]*

Evaluates to true if *String* is accepted by *Regex*.

`match_suffix_regex/3`, `match_all_regex/3`

match_suffix_regex(*+Regex, +String, -Suffix***)** *[det]*

Evaluates to true if *Suffix* is the remaining part of the *String* not matched by *Regex*.

`match_regex/2`, `match_all_regex/3`

match_all_regex(*+Regex, +String, -Match***)** *[nondet]*

Instantiates *Match* to all possible matches of *Regex* in *String*.

`match_regex/2`, `match_suffix_regex/3`

2.2 automata(ast): Grammar constituents used when parsing regular expressions

`https://github.com/wvxvw/intro-to-automata-theory`

This module defines predicates for generating abstract syntax trees representing regular expressions.

rterminal(*?Regex* **)** *[nondet]*

Evaluates to true if *Regex* is either an atom or an empty list. Empty list denotes empty string, atoms stand for characters of the strings.

`runion/2`, `rstar/1`, `rconcat/2`, `regex/1`

runion(*+Regex1, +Regex2* **)** *[det]*

Evaluates to true if *Regex1* and *Regex2* are valid regular expressions as defined in `regex/1`.

`runion/2`, `rstar/1`, `rconcat/2`, `regex/1`

rstar(*+Regex* **)** *[det]*

Evaluates to true if *Regex* is a valid regular expressions as defined in `regex/1`.

`rterminal/1`, `rstar/1`, `rconcat/2`, `regex/1`

rconcat(*+Regex1, +Regex2* **)** *[det]*

Evaluates to true if *Regex1* and *Regex2* are valid regular expressions as defined in `regex/1`.

`runion/2`, `rterminal/1`, `rconcat/2`, `regex/1`

regex(*+Regex* **)** *[det]*

Evaluates to true if *Regex* is either a `runion/2`, or a `rstar/1`, or a `rconcat/2` or a `rterminal/1`.

`runion/2`, `rstar/1`, `rconcat/2`, `rterminal/1`

2.3 automata(convert): Convert between different automata represenations

`https://github.com/wvxvw/intro-to-automata-theory`

This module defines defines conversions between regular expression AST, DFA represented as a list of transitions or as a table, and NFA represented similarly to DFA.

This module also defines data types:

- Transition record

  ```
  trn(from:integer, to:integer, input:input, acc:boolean)
  ```

  To describe a single transition between states on some input. `acc` is `true` whenever the target state is an accepting one.

- State record

  ```
  state(label, acc:boolean)
  ```

  To describe states.

- Transition table row record

  ```
  row(state:state, trns:trns)
  ```

  To describe all inputs for a given state.

- Transition table record

  ```
  table(inputs:list(input), tab:list(row))
  ```

  To describe a complete table of transitions between all states of some automata.

has(*+Accessor, ?Value, ?Record***)** *[det]*
Flips arguments for Accessr (the predicate generated to access fields of the record).

regex_to_nfa(*+Regex, -Nfa***)** *[det]*
Evaluates to true when given regular expression *Regex* can be decomposed into a list of transitions *Nfa*.

`gexps/3`

nfa_inputs(*+Nfa, -Inputs***)** *[det]*
Evaluates to true when *Inputs* is the alphabet of the *Nfa* automata.

nfa_states(*+Nfa, -States***)** *[det]*
Evaluates to true when *States* is the states of the *Nfa* automata.

reacheable_states(*+Input, +Transitions, +Table, -States***)** *[det]*

Evaluates to true when *States* can be reached from all *Transitions* on given *Input*. This also accounts for epsilon transitions.

nfa_table(*+Transitions, -Table:table***)** *[det]*

Evaluates to true when *Table* is the transitions table containing all transitions given by *Transitions*.

nfa_to_dfa(*+Nfa, -Dfa***)** *[det]*

Evaluates to true when *Dfa* accepts the same language as *Nfa*.

table_to_diagram(*+Table, -Diagram***)** *[det]*

Evaluates to true when *Diagram* contains all the transitions described in *Table*.

2.4 automata(parser): DCG rules for parsing regular expressions

```
https://github.com/wvxvw/intro-to-automata-theory
```

This module defines DCG rules for parsing regular expressions from string.

gstar(*+Exp, +Prefix, -Suffix***)** *[det]*

Parses a regular expression followed by an asterisk (the Kleene operator). Instantiates its first term to the rstar AST nonterminal.

```
rstar/1
```

gunion(*+Exp, +Prefix, -Suffix***)** *[det]*

Parses a union of two regular expressions joined by the `+` sign. Instantiates its first term to the runion AST nonterminal.

```
runion/2
```

gchar(*+Exp, +Prefix, -Suffix***)** *[det]*

Parses a single terminal character and instantiates it to AST rterminal term.

```
rterminal/1
```

gexps(*+Tree, +Prefix, -Suffix***)** *[det]*

Parses regular expression from the string *Prefix* and instantiates the *Tree* to the parse `regex/1` term.

```
regex/1
```

2.5 automata(printing): Predicates for pretty printing

`https://github.com/wvxvw/intro-to-automata-theory`

This module defines predicates useful to print structures generated by other automata modules.

regex_to_string(+*Exp*, -*Result***)** *[det]*

Evaluates to true when given regular expression *Exp* can be written as *Result* string.

`regex/1`

format_table(+*Table:table***)** *[det]*

Pretty-prints the contents of the transitions table.

`nfa_to_dfa/2`, `nfa_table/2`