Exercise Sheet Solutions 2

Compiler Design Lab
Prof. Dr. Sebastian Hack
Fabian Ritter, B.Sc.

This is a solution proposal. It does not necessarily describe complete solutions but gives the initial ideas to solve the theoretical exercises and presents the results.

Exercise 2.1 Push-Down Automata

Let $(\{S, A, B, C, D, H, K\}, \{a, b, c, d, e\}, P, S)$ be a context-free grammar with the following productions P:

$$\begin{aligned} S &\to KA \mid BK \\ A &\to abA \mid BcH \mid \varepsilon \\ B &\to eBd \mid c \\ C &\to dAb \mid aa \\ D &\to S \mid \varepsilon \\ H &\to CD \\ K &\to cd \end{aligned}$$

Write down a successful run of the push-down automaton constructed for this grammar (using the algorithms presented in the lecture) on the input word $cdeecddcaaccd$.

Solution: For the generic approach presented in the lecture, we need to have an extended context-free grammar. So we add the synthetic start symbol S'.

Stack	Remaining Input
$[S' \to .S]$	$cdeecddcaaccd$
$[S' \to .S][S \to .KA]$	$cdeecddcaaccd$
$[S' \to .S][S \to .KA][K \to .cd]$	$cdeecddcaaccd$
$[S' \to .S][S \to .KA][K \to c.d]$	$deecddcaaccd$
$[S' \to .S][S \to .KA][K \to cd.]$	$eecddcaaccd$
$[S' \to .S][S \to K.A]$	$eecddcaaccd$
$[S' \to .S][S \to K.A][A \to .BcH]$	$eecddcaaccd$
$[S' \to .S][S \to K.A][A \to .BcH][B \to .eBd]$	$eecddcaaccd$
$[S' \to .S][S \to K.A][A \to .BcH][B \to e.Bd]$	$ecddcaaccd$
$[S' \to .S][S \to K.A][A \to .BcH][B \to e.Bd][B \to .eBd]$	$ecddcaaccd$
$[S' \to .S][S \to K.A][A \to .BcH][B \to e.Bd][B \to e.Bd]$	$cddcaaccd$
$[S' \to .S][S \to K.A][A \to .BcH][B \to e.Bd][B \to e.Bd][B \to .c]$	$cddcaaccd$
$[S' \to .S][S \to K.A][A \to .BcH][B \to e.Bd][B \to e.Bd][B \to c.]$	$ddcaaccd$
$[S' \to .S][S \to K.A][A \to .BcH][B \to e.Bd][B \to eB.d]$	$ddcaaccd$
$[S' \to .S][S \to K.A][A \to .BcH][B \to e.Bd][B \to eBd.]$	$dcaaccd$
$[S' \to .S][S \to K.A][A \to .BcH][B \to eB.d]$	$dcaaccd$
$[S' \to .S][S \to K.A][A \to .BcH][B \to eBd.]$	$caaccd$
$[S' \to .S][S \to K.A][A \to B.cH]$	$caaccd$
$[S' \to .S][S \to K.A][A \to Bc.H]$	$aaccd$
$[S' \to .S][S \to K.A][A \to Bc.H][H \to .CD]$	$aaccd$
$[S' \to .S][S \to K.A][A \to Bc.H][H \to .CD][C \to .aa]$	$aaccd$
$[S' \to .S][S \to K.A][A \to Bc.H][H \to .CD][C \to a.a]$	$accd$
$[S' \to .S][S \to K.A][A \to Bc.H][H \to .CD][C \to aa.]$	ccd
$[S' \to .S][S \to K.A][A \to Bc.H][H \to C.D]$	ccd
$[S' \to .S][S \to K.A][A \to Bc.H][H \to C.D][D \to .S]$	ccd
$[S' \to .S][S \to K.A][A \to Bc.H][H \to C.D][D \to .S][S \to .BK]$	ccd
$[S' \to .S][S \to K.A][A \to Bc.H][H \to C.D][D \to .S][S \to .BK][B \to .c]$	ccd
$[S' \to .S][S \to K.A][A \to Bc.H][H \to C.D][D \to .S][S \to .BK][B \to c.]$	cd
$[S' \to .S][S \to K.A][A \to Bc.H][H \to C.D][D \to .S][S \to B.K]$	cd
$[S' \to .S][S \to K.A][A \to Bc.H][H \to C.D][D \to .S][S \to B.K][K \to .cd]$	cd
$[S' \to .S][S \to K.A][A \to Bc.H][H \to C.D][D \to .S][S \to B.K][K \to c.d]$	d
$[S' \to .S][S \to K.A][A \to Bc.H][H \to C.D][D \to .S][S \to B.K][K \to cd.]$	
$[S' \to .S][S \to K.A][A \to Bc.H][H \to C.D][D \to .S][S \to BK.]$	
$[S' \to .S][S \to K.A][A \to Bc.H][H \to C.D][D \to S.]$	
$[S' \to .S][S \to K.A][A \to Bc.H][H \to CD.]$	
$[S' \to .S][S \to K.A][A \to BcH.]$	
$[S' \to .S][S \to KA.]$	
$[S' \to S.]$	

Exercise 2.2 Grammar Ambiguity, Theory and Practice

Consider a grammar that captures a subset of the statements of C with starting non-terminal *stmt* and the following productions (expression rules are omitted for brevity):

$$
\begin{array}{rcl}
\textit{stmt} & \to & \textit{expr}\textbf{;} \\
& | & \textbf{return } \textit{expr}\textbf{;} \\
& | & \textbf{if (}\textit{expr}\textbf{) } \textit{stmt} \\
& | & \textbf{if (}\textit{expr}\textbf{) } \textit{stmt} \textbf{ else } \textit{stmt} \\
& | & \textbf{while (}\textit{expr}\textbf{) } \textit{stmt} \\
& | & \textbf{\{ } \textit{stmt-seq} \textbf{ \}} \\
\textit{stmt-seq} & \to & \textit{stmt-seq stmt} \mid \textit{stmt} \\
\textit{expr} & \to & \ldots
\end{array}
$$

1. Give an input token string which demonstrates that this grammar is ambiguous.

2. Adjust the grammar such that it becomes unambiguous (but still describes the same language as before). *Hint: Introduce new non-terminal symbols to group the different statements into categories and add slightly modified duplicates of some of the productions.*

3. Check how this ambiguity issue is treated in the C standard.

Solution:

1. The following input cannot be parsed unambiguously with the given grammar as it does not specify which `if` the `else` belongs to:

   ```
   if (c1) if (c2) return 1; else return 0;
   ```

2. We introduce two new non-terminals *matched_stmt* (for statements that, if they were followed by an `else`, could no longer be accepted by the original grammar) and *open_stmt* (for the remaining statements) and adjust the productions as follows:

$$
\begin{array}{rcl}
\textit{stmt} & \rightarrow & \textit{matched\_stmt} \mid \textit{open\_stmt} \\
\textit{matched\_stmt} & \rightarrow & \textbf{if (}\textit{expr}\textbf{)}\ \textit{matched\_stmt}\ \textbf{else}\ \textit{matched\_stmt} \\
& \mid & \textbf{while (}\textit{expr}\textbf{)}\ \textit{matched\_stmt} \\
& \mid & \textit{expr}\textbf{;} \\
& \mid & \textbf{return}\ \textit{expr}\textbf{;} \\
& \mid & \textbf{\{}\ \textit{stmt-seq}\ \textbf{\}} \\
\textit{open\_stmt} & \rightarrow & \textbf{if (}\textit{expr}\textbf{)}\ \textit{stmt} \\
& \mid & \textbf{if (}\textit{expr}\textbf{)}\ \textit{matched\_stmt}\ \textbf{else}\ \textit{open\_stmt} \\
& \mid & \textbf{while (}\textit{expr}\textbf{)}\ \textit{open\_stmt} \\
\textit{stmt-seq} & \rightarrow & \textit{stmt-seq}\ \textit{stmt} \mid \textit{stmt} \\
\textit{expr} & \rightarrow & \ldots
\end{array}
$$

   In general, we have to duplicate productions for all *stmt* productions whose right-hand side ends in a *stmt*.

3. The C standard gives a grammar like the one in the exercise text and remarks: "An `else` is associated with the lexically nearest preceding `if` that is allowed by the syntax." (§6.8.4.1-3)

Exercise 2.3 LL(k)

A grammar is an LL(k)-grammar for some $k \in \mathbb{N}$ if whenever there exist $u, x, y \in {V_T}^*$ with $k : x = k : y, Y \in V_N$ and $\alpha, \beta, \gamma \in (V_T \cup V_N)^*$ such that

$$
\begin{array}{lllllll}
S & \overset{*}{\underset{lm}{\Longrightarrow}} & uY\alpha & \underset{lm}{\Longrightarrow} & u\beta\alpha & \overset{*}{\underset{lm}{\Longrightarrow}} & ux \\
S & \overset{*}{\underset{lm}{\Longrightarrow}} & uY\alpha & \underset{lm}{\Longrightarrow} & u\gamma\alpha & \overset{*}{\underset{lm}{\Longrightarrow}} & uy
\end{array}
$$

then $\beta = \gamma$
A language L is an LL(k)-language if there exists an LL(k)-grammar that generates L.

1. Prove that for each $k \in \mathbb{N}$ there exists a grammar which is $LL(k+1)$ but not LL(k).

2. Prove that for each $k \in \mathbb{N}$ an LL(k)-grammar is an LL($k+1$)-grammar.

3. Investigate the relationship between LL(0)-languages and regular languages. In particular provide the following information.

   - $\{|x| \mid x \in \text{LL(0)}\}$, where LL(0) is the set of all LL(0)-languages.
   - $\{|x| \mid x \in L_{reg}\}$, where L_{reg} is the set of all regular language.

- Which set relation holds between LL(0) and L_{reg}?

4. A grammar is left-recursive if it has a production of the form $A \to A\mu$. Show that a left-recursive grammar is not LL(k) for any k.

Solution:

1. Consider the grammar $G : S \to a^k b | a^k c$. Let $u, x, y \in {V_T}^*, Y \in V_N$ and thus $Y = S$ and let $\alpha, \beta, \gamma \in (V_T \cup V_N)^*$. Then $\beta, \gamma, x, y \in \{a^k b, a^k c\}$ and $u = \alpha = \varepsilon$ for all possible derivations

$$\begin{aligned} S &\Rightarrow^*_{lm} uY\alpha \Rightarrow_{lm} u\beta\alpha \Rightarrow^*_{lm} ux \; and \\ S &\Rightarrow^*_{lm} uY\alpha \Rightarrow_{lm} u\gamma\alpha \Rightarrow^*_{lm} uy. \end{aligned}$$

   For LL($k+1$) we see that $k+1 : x = k+1 : y$ is always false and the implication thus always true. Thus G is a LL($k+1$) grammar. For LL(k) the condition $k : x = k : y$ holds, but if $x = \beta = a^k b, y = \gamma = a^k c$ we find that $\beta \neq \gamma$ and thus $G \notin$ LL(k).

2. Given $k \in \mathbb{N}, x \in {V_T}^*$ we see that $k+1 : x = k+1 : y$ implies $k : x = k : y$. By the transitivity of implications, we can thus replace the latter by the first in the implication in the definition. This yields the definition of LL(k+1) grammars.

3. For $k = 0$ the condition $0 : x = 0 : y$ simplifies to true. Thus for LL(0) grammars $(S \Rightarrow^*_{lm} uY\alpha \Rightarrow_{lm} u\beta\alpha \Rightarrow^*_{lm} ux \wedge S \Rightarrow^*_{lm} uY\alpha \Rightarrow_{lm} u\gamma\alpha \Rightarrow^*_{lm} uy) \Rightarrow \beta = \gamma$ holds. This means that each step in a derivation is uniquely determined and thus completely independent of the input. An LL(0) grammar thus produces only languages that contain one or zero (if there is no production at all) words. As all finite languages are regular and there are regular languages with size >1 we conclude that LL(0) $\subsetneq$ REG.

   - $\{|x| \mid x \in LL(0)\} = \{0, 1\}$
   - $\{|x| \mid x \in L_{reg}\} = \mathbb{N}$
   - $LL(0) \subseteq L_{reg}$

4. Let $k \in \mathbb{N}$ and G be a reduced, left-recursive grammar and $A \to A\mu \in P$ a left recursive production of G. As G is reduced the production is not useless and there exists a derivation that contains this production. Thus a production $A \to \nu \in P$ exists that eventually allows A to terminate, that is $\nu \Rightarrow^*_{lm} v \in {V_T}^*$. We also assume that $\mu \Rightarrow^*_{lm} w \in {V_T}^+$.

   We now construct the following derivations using $A \to A\mu$ repeatedly until terminating with $A \to \nu$:

$$\begin{aligned} S &\Rightarrow^*_{lm} uA\mu^k\alpha \Rightarrow_{lm} u \underbrace{\nu}_{\beta}\mu^k\alpha \Rightarrow^*_{lm} u\underbrace{vw^k z}_{x} \\ S &\Rightarrow^*_{lm} uA\mu^k\alpha \Rightarrow_{lm} u \underbrace{A\mu}_{\gamma}\mu^k\alpha \Rightarrow^*_{lm} u\underbrace{vw^{k+1}z}_{y} \; with \; \alpha \Rightarrow^*_{lm} z \in {V_T}^*. \end{aligned}$$

   We find that $k : x = k : y$ holds but $\beta \neq \gamma$ and thus $G \notin$ LL(k).

Exercise 2.4 Checkable LL(k) conditions

The formal definition of an LL(k)-grammar as given in the lecture is not very handy for checking if a given grammar is an LL(k)-grammar. Therefore the lecture about LL-parsing introduced some checkable LL(k) conditions (slides 31 and 32).

- Show that an LL(k)-grammar does in general not have to be a strong LL(k)-grammar for $k > 1$.
- Show that an LL(1)-grammar is always also a strong LL(1)-grammar. (Prove one direction of the theorem on slide 31 of the lecture about LL-parsing.)

- Provide a sufficient condition to find out if a given context-free grammar is an LL(k)-grammar. This condition should be weaker than the check if a grammar is a strong LL(k)-grammar. Give an example where your condition classifies a grammar as LL(k)-grammar even if it is no strong LL(k)-grammar. Remember that the definition of an LL(k)-grammar itself is of course also a sufficient condition, but for grammars that define infinite languages it cannot be checked.

Solution:

- We show this by giving an example of a grammar G which is LL(3) but not strong LL(3) with the following production rules.

$$\begin{aligned} S &\rightarrow aAhu \mid bAuhu \\ A &\rightarrow hu \mid h \end{aligned}$$

The grammar is LL(3). From the sententional form $aAhu$ we can only derive $ahuhu$ and $ahhu$. $3 : huhu = huh \neq 3 : hhu = hhu$. From the sententional form $bAuhu$ we can only derive $bhuuhu$ and $bhuhu$. $3 : huuhu = huu \neq 3 : huhu = huh$. So the left side of the implication in the definition of LL(3)-grammar is always false which makes the overall implication always true.

It is however not strong LL(3) since:

$$(First_3(hu) \oplus_3 Follow_3(A)) \cap (First_3(h) \oplus_3 Follow_3(A)) = \{huh\} \neq \emptyset$$

- We have to show that in the special case of $k = 1$ it holds that an LL(1)-grammar is always also a strong LL(1)-grammar.

The expanded definition of an LL(k)-grammar looks as follows for the special case $k = 1$:

If

$$\begin{aligned} S &\overset{*}{\underset{lm}{\Longrightarrow}} uY\alpha \underset{lm}{\Longrightarrow} u\beta\alpha \overset{*}{\underset{lm}{\Longrightarrow}} ux \\ S &\overset{*}{\underset{lm}{\Longrightarrow}} uY\alpha \underset{lm}{\Longrightarrow} u\gamma\alpha \overset{*}{\underset{lm}{\Longrightarrow}} uy \end{aligned}$$

and $1 : x = 1 : y$ then $\beta = \gamma$

We prove this by proving its contrapositive. We assume to have a reduced grammar G that is not strong LL(1) and then show that it is also not LL(1).

If a grammar is not strong LL(1), then it has two different productions $A \rightarrow \beta$ and $A \rightarrow \gamma$ s.t.:

$$(First_1(\beta) \oplus_1 Follow_1(A)) \cap (First_1(\gamma) \oplus_1 Follow_1(A)) = intersect \neq \emptyset$$

Now we take a look at an arbitrary $a \in intersect$. Since G is reduced, a may only consist of a single token or of $\#$. Therefore we can distinguish a few cases:

1. $a \in First_1(\beta) \wedge a \in First_1(\gamma)$:
   In this case we can get a derivation:

$$\begin{aligned} S &\overset{*}{\underset{lm}{\Longrightarrow}} uY\alpha \underset{lm}{\Longrightarrow} u\beta\alpha \overset{*}{\underset{lm}{\Longrightarrow}} ux \\ S &\overset{*}{\underset{lm}{\Longrightarrow}} uY\alpha \underset{lm}{\Longrightarrow} u\gamma\alpha \overset{*}{\underset{lm}{\Longrightarrow}} uy \end{aligned}$$

   and $1 : x = 1 : y = a$ and $\beta \neq \gamma$ by assumption.
   So in this case we violated the definition of LL(1).

2. $a \in First_1(\beta) \wedge a \notin First_1(\gamma)$:
   Then we know that $\varepsilon \in First_1(\gamma)$ and $a \in Follow_1(A)$.
   So we can get a derivation:

$$\begin{array}{lllllll} S & \overset{*}{\underset{lm}{\Longrightarrow}} & uYa\alpha & \underset{lm}{\Longrightarrow} & u\beta a\alpha & \overset{*}{\underset{lm}{\Longrightarrow}} & ux \\ S & \overset{*}{\underset{lm}{\Longrightarrow}} & uYa\alpha & \underset{lm}{\Longrightarrow} & u\gamma a\alpha & \overset{*}{\underset{lm}{\Longrightarrow}} & uy \end{array}$$

   and $1 : x = 1 : y = a$ and $\beta \neq \gamma$ by assumption.
   We get this derivation by deriving ε from γ and something that starts with a from β.
   So again we have violated the definition of LL(1).
3. $a \notin First_1(\beta) \wedge a \in First_1(\gamma)$:
   Works in the same way as the case before just with the roles of β and γ switched.
4. $a \notin First_1(\beta) \wedge a \notin First_1(\gamma)$:
   Then we know that $\varepsilon \in First_1(\beta)$, $\varepsilon \in First_1(\gamma)$ and $a \in Follow_1(A)$.
   Again we can get a derivation:

$$\begin{array}{lllllll} S & \overset{*}{\underset{lm}{\Longrightarrow}} & uYa\alpha & \underset{lm}{\Longrightarrow} & u\beta a\alpha & \overset{*}{\underset{lm}{\Longrightarrow}} & ux \\ S & \overset{*}{\underset{lm}{\Longrightarrow}} & uYa\alpha & \underset{lm}{\Longrightarrow} & u\gamma a\alpha & \overset{*}{\underset{lm}{\Longrightarrow}} & uy \end{array}$$

   and $1 : x = 1 : y = a$ and $\beta \neq \gamma$ by assumption.
   We get this derivation by deriving ε from γ and ε from β.

We have seen that the set $intersect$ contains at least one single symbol element and that in any possible combination that this element may have arised from the first and follow it leads to the grammar not being LL(1). We follow that a grammar being LL(1) is also strong LL(1).

- The following algorithm provides an appropriate condition:

```
bool isLL(CFG G, unsigned int k) {
  for ( (A → β), (A → γ) ∈ G.P with β ≠ γ ) {
    for ( (X → θAδ) ∈ G.P ) {
      lookIntoBeta := First_k(β) ⊕_k First_k(δ) ⊕_k Follow_k(X);
      lookIntoGamma := First_k(γ) ⊕_k First_k(δ) ⊕_k Follow_k(X);
      if( lookIntoBeta ∩ lookIntoGamma ≠ ∅ )
        return false;
    }
  }
  return true;
}
```

As example we can use the grammar G which is LL(3) but not strong LL(3) as shown before.

The following exercises provide further opportunities for practicing with finite automata, item PDAs and regular expressions. They might not be discussed in full detail in the tutorials.

Exercise 2.5 Item-PDAs Revisited

Let the pushdown automaton $P = (\{a, b\}, \{q_0, q_1, q_2, q_3\}, \Delta, q_0, \{q_3\})$, where

$$\Delta = \{(q_0, a, q_0q_1), (q_0, b, q_0q_2), (q_0, \#, q_3), (q_1, a, q_1q_1), (q_1, b, \epsilon), (q_2, a, \epsilon), (q_2, b, q_2q_2)\}$$

and $\# \notin \Sigma$ symbolizes the end of the input word, be given.

Provide a context-free grammar that generates the language L accepted by P. If possible, provide also a regular expression for L. Otherwise provide sufficient arguments why this is not possible.

Solution: The context-free grammar generating the language L accepted by the given PDA looks like this:

$$S \rightarrow aSb \mid bSa \mid \epsilon \mid SS$$

Here we can see, that L consists of as many a's as there are b's. To prove that it is not possible to provide a regular expression for L, we can show that it is not regular by using the pumping lemma.

We can also argument in a very simple way why it is not possible to give a regular expression describing the language L. If it would be possible to give a regular expression, then it would also be possible to describe a finite state machine accepting the language. Let such a finite state machine have an arbitrary high but finite number z of states. That finite state machine would have to remember at least the difference between the number of times it encountered a and it encountered b so far. To accept the word $a^z b^z$, which is in language L, we would already need $z + 1$ states corresponding to the differences 0 to z. So there cannot be such a finite state machine accepting the language and therefore also no regular expression describing it.

Exercise 2.6 Regular Expressions and Languages

The lecture defined regular expressions using the metacharacters $\underline{\emptyset}$ and $\underline{\varepsilon}$. Show that they are the neutral elements with respect to the alternative and concatenation operations in regular expressions. This means show that:

- $\underline{(}r_1\underline{|}\underline{\emptyset}\underline{)}$ describes the same language as r_1
- $\underline{(}r_1\underline{\varepsilon}\underline{)}$ describes the same language as r_1

only by reasoning about the described languages as shown in the lecture. Assume the regular expression r_1 to denote the language R_1.

Solution:

$$\begin{aligned} L(\underline{(}r_1\underline{|}\underline{\emptyset}\underline{)}) &= L(r_1) \cup L(\underline{\emptyset}) \\ &= L(r_1) \cup \emptyset \\ &= L(r_1) \end{aligned}$$

$$\begin{aligned} L(\underline{(}r_1\underline{\varepsilon}\underline{)}) &= L(r_1)L(\underline{\varepsilon}) \\ &= L(r_1)\{\varepsilon\} \\ &= \{xy | x \in L(r_1) \wedge y \in \{\varepsilon\}\} \\ &= \{x\varepsilon | x \in L(r_1)\} \\ &= \{x | x \in L(r_1)\} \\ &= L(r_1) \end{aligned}$$

Exercise 2.7 Finite Automata Reloaded

In this exercise we take a closer look at recognizing common language structures like comments. Consider comments in XML which start with $<!--$ and end with the first occurrence of $-->$. However, XML comments are not nestable. So the first $-->$ ends the comment no matter how many $<!--$ it contained. We can define the construct $<!-- \ until \ -->$ to describe such comments.

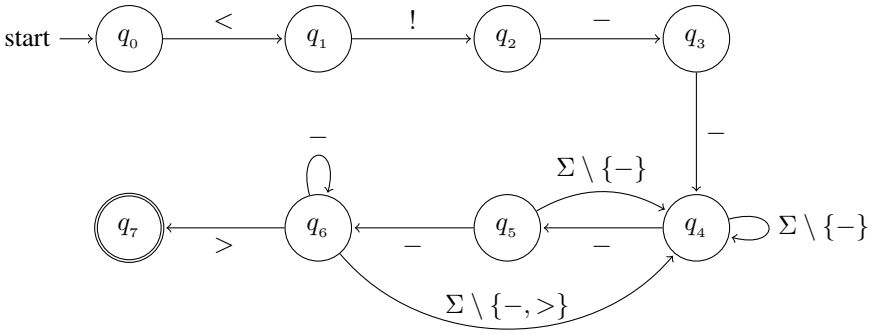
Create a minimal deterministic finite automaton that accepts XML comments over an alphabet Σ, where $\{<, >, -, !\} \subseteq \Sigma$. You may label an automaton edge with $\Sigma \setminus \{x, y\}$ to express that there are in fact edges for all of the alphabet's symbols except $\{x, y\}$.

Solution: You can either directly create the DFA from scratch or follow the design flow described in the lecture.

In case you design it directly, you should at least briefly describe why the automaton is deterministic (no edges labeled with the same character starting from the same node, no ε-transitions) and verify that it is minimal (for example by applying the minimization algorithm and see that it does no get any smaller).

In case you follow the design flow from the lectures, you already used all the (provably correct) algorithms and can furthermore be sure that the resulting DFA in the end will accept exactly the intended language.

Note that there is only one (up to renaming unique) minimal DFA accepting the language.


start
q_0
$<$
q_1
!
q_2
$-$
q_3
$-$
$\Sigma \setminus \{-\}$
q_7
$>$
q_6
$-$
q_5
$-$
q_4
$\Sigma \setminus \{-\}$
$-$
$\Sigma \setminus \{-, >\}$