Exercise Sheet Solutions 1

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.

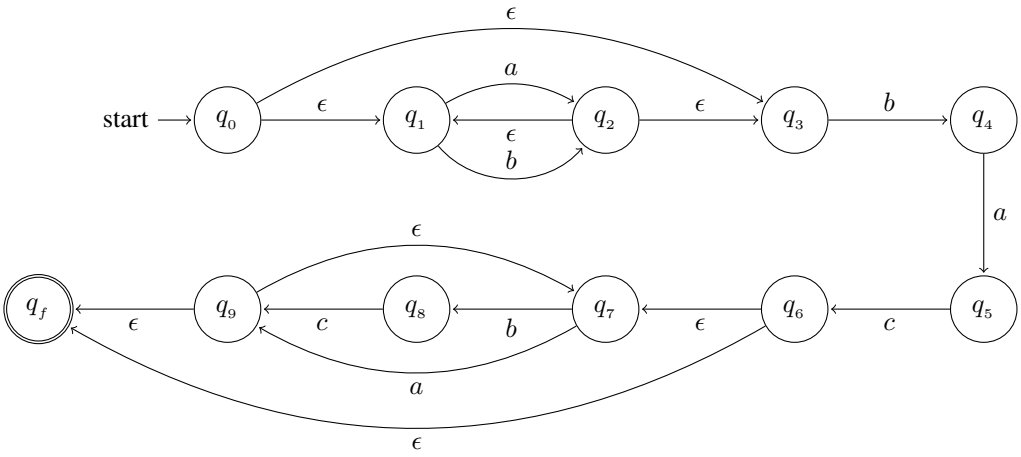
Exercise 1.1 Regular Expressions and Finite Automata

1. For the regular expression $(a|b)^*bac(a|bc)^*$ over the alphabet $\Sigma = \{a, b, c\}$ construct:
   (a) a nondeterministic finite automaton,
   (b) a deterministic finite automaton, and
   (c) a minimal deterministic finite automaton

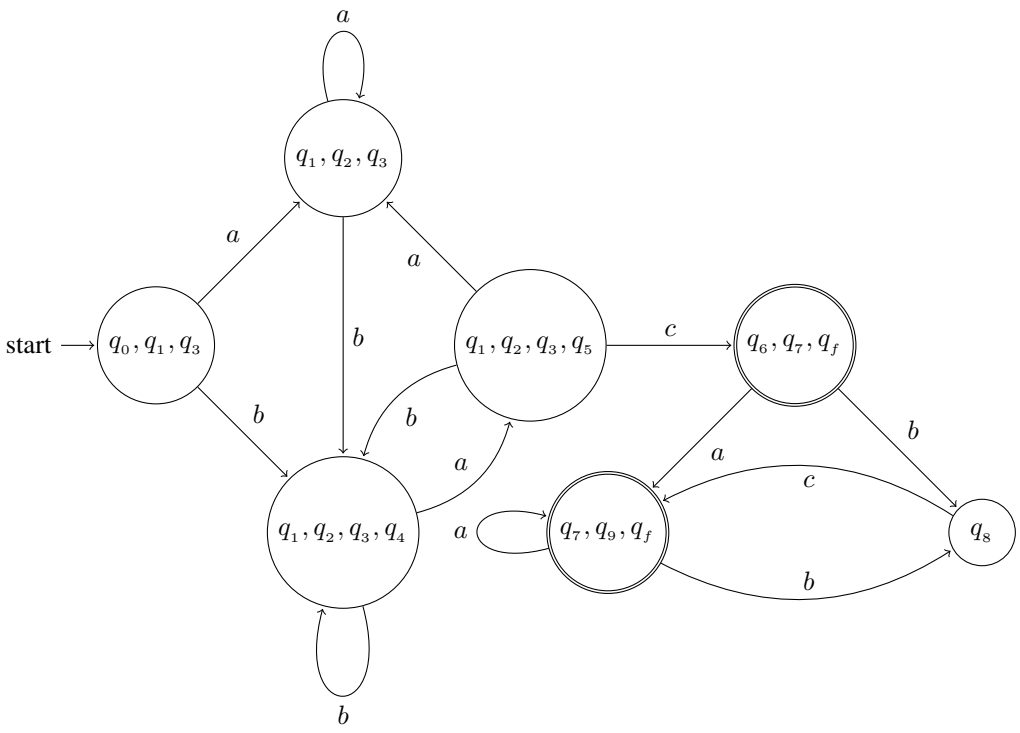
   following the proof sketches from the lecture.
2. At first sight, the construction of an NFA from a regular expression as presented in the lecture seems to be too complicated in the case of r^* (Slide 18). Give examples where simplifying the rule by merging states
   (a) q and q_1 fails.
   (b) q_2 and p fails.
3. Write a regular expression that accepts only valid decimal floating point expressions as defined in §6.4.4.2 of the C language definition (see below).

Solution:

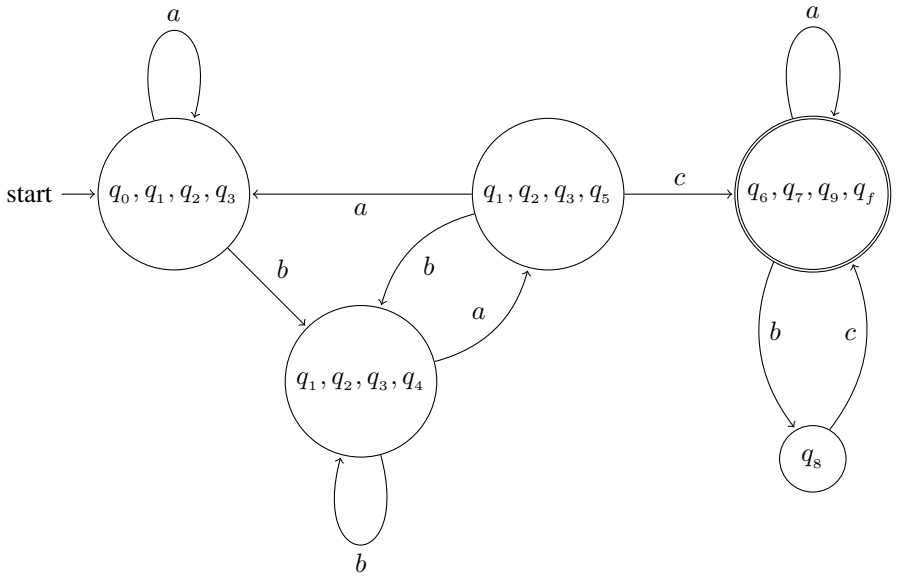
1. (a) Nondeterministic finite automaton:



   (b) Deterministic finite automaton



(c) Minimal deterministic finite automaton



2. (a) The construction for $(b|a*)$ would accept the input word $aaab$ which is not in the language.
   (b) The construction for $(b|a*)$ would accept the input word $baaa$ which is not in the language.

3. Let $D := (0|1|2|3|4|5|6|7|8|9)$.

   We translate the relevant rules individually:

digit-sequence	$DD*$
sign	$(+\|-)$
floating-suffix	$(f\|l\|F\|L)$
exponent-part	$(e\|E)$ $(\varepsilon|$*sign*$)$ *digit-sequence*
fractional-constant	$((\varepsilon|$*digit-sequence*$)$.*digit-sequence*$)$
	$\|$ (*digit-sequence*.$)$
decimal-floating-constant	(*fractional-constant* $(\varepsilon|$*exponent-part*$)$ $(\varepsilon|$*floating-suffix*$)$)
	$\|$ (*digit-sequence exponent-part* $(\varepsilon|$*floating-suffix*$)$)

Overall, we get by substituting (and slight simplifications):

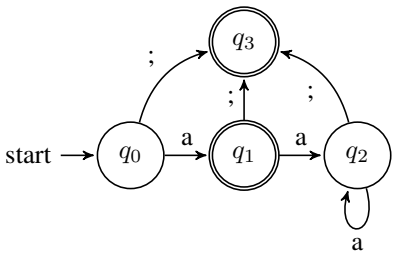
decimal-floating-constant	$(((D*.DD*) \mid (DD*.))\ (\varepsilon\|((e\|E)\ (\varepsilon\|+\|-)\ DD*))\ (\varepsilon\|f\|l\|F\|L))$
	$\mid (DD*\ ((e\|E)\ (\varepsilon\|+\|-)\ DD*)\ (\varepsilon\|f\|l\|F\|L))$

Exercise 1.2 Greed

1. Provide an automaton and a corresponding input word, such that the maximal munch strategy has runtime in $\Omega(n^2)$. Prove your claim!
2. Is there a sequence of regular expressions $\alpha_1, \ldots, \alpha_n$ over Σ and a word $w \in \Sigma^*$, such that
   - there exist $w_1, \ldots, w_k \in \Sigma^*$ and $t_1, \ldots, t_k \in \{1, \ldots, n\}$, such that $w = w_1 \ldots w_k$ and $w_i \in \alpha_{t_i}$ (that is, there is a match of the complete word), and
   - for each such dissection there exist $j \in \{1, \ldots, k-1\}$, $x \in \Sigma^+$, $y \in \Sigma^*$, and $s \in \{1, \ldots, n\}$, such that $w_{j+1} \ldots w_k = xy$, $w_j x \in \alpha_s$, and there is no dissection $y = w'_1 \ldots w'_m$ and $t'_1, \ldots, t'_m \in \{1, \ldots, n\}$, such that $w'_i \in \alpha_{t'_i}$ (that is, maximal munch scanning does not take us to a match)?

Solution:

1. Consider the example regular expression from the lecture: $(a|a^*;)$ and the following automaton that recognizes the same language:



   Given the input string a^n the maximal-munch strategy needs $\Omega(n^2)$ steps to recognize all symbols. The only possible dissection of this input into symbols is having one symbol per character a. The maximal-munch strategy however reads all remaining characters for each a until finding that no ; follows and back-tracking to the first a read. This results in $n + (n-1) + \ldots + 1 = \frac{n \cdot (n+1)}{2} \in \Omega(n^2)$ steps.

2. Consider the following sequence of regular expressions: (ax) (xb) (a) and the input word axb. The only possible dissection is by a and xb. Greedy scanning will however recognize ax and will then be left with b, which is not accepted by any combination of regular expressions.

Exercise 1.3 Minimizing Automata

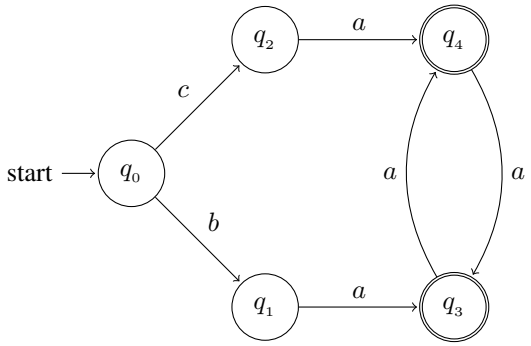
Consider the following alternative algorithm for minimizing automata: Let $\langle \Sigma, Q, \Delta, q_0, F \rangle$ be a DFA. We merge two states $q_1, q_2 \in Q$ iff $q_1 \in F \Leftrightarrow q_2 \in F$ and for all $a \in \Sigma$, for all $p \in Q$ holds: $(q_1, a, p) \in \Delta \Leftrightarrow (q_2, a, p) \in \Delta$. We apply this rule until no further states can be merged.

Does this algorithm correctly minimize arbitrary deterministic finite automata? Find a proof or a counter example to back up your claim!

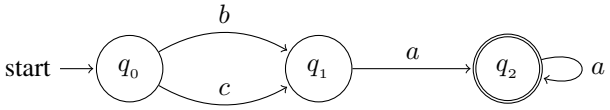
Solution: First of all, we will point out the main difference of the two algorithms. The algorithm presented in the lecture starts with two sets: F (containing all accepting states of the DFA) and $Q - F$. To refine this partition it is necessary to show inequality, what means the $\vdash_M^*$ leads from one state to a state in the same part of the partition.

The other one starts with the set Q (containing all states). To merge two states it is necessary to show equality, that means the transition of each state over a letter a must lead to the *same* state. It is harder to show the equality (concerning concrete states) than the inequality (concerning sets of states). This is problematic when the DFA M contains cycles.

For example for the following DFA the algorithm fails.



The algorithm of lecture transforms the automaton to the following one, whereas the new algorithm fails to do so.



Project task A General Information

In the practical project you will implement a compiler for a subset of C. The milestones of the project roughly reflect the structure of a compiler, which also structures the lecture. Although the milestones will not be graded, you will receive feedback that hints at existing shortcomings in your compiler. Each milestone comes with a soft deadline that indicates the expected time schedule. The final submission will be graded based on the percentage of tests passed and a code review where the group members have to justify their work.

Technical requirements and restrictions:

- The compiler itself must be written in C++.
- Your project must be buildable with the provided Makefile, which produces a binary `c4`.
- Upon acceptance of a source program, `c4` must terminate with return code 0. Rejection of a source program must be indicated by return code 1 and an error message with the following format on `stderr`:

  `<filename>:<line>:<column>: error: <message>`

  Here, `<filename>` is the name of the input file as provided to your compiler, `<line>` and `<column>` are the line number and the column number of the error's location, and `<message>` is some meaningful error message.
- The individual assignments will refine the requirements.

Project task B Lexer

To get started:

- Make sure that you have logged in to the Forum at `https://discourse.cdl.uni-saarland.de` and followed the "First Steps". You should then have a Gitlab account, be in a group there with your team mates and you should have a fresh project with the name `c4` (for "Compiler Construction C Compiler", C^4) living in your group's namespace.
- When in your project's root folder, merge the template project into your sources:

```
git remote add template git@cc.cdl.uni-saarland.de:template/c4.git
git fetch --all
git merge --allow-unrelated-histories template/master
```

- Examine the contents of the starter kit. It provides a Makefile which you can use to build the project. The produced binary is called `c4`.
- Get the language specification from `http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf`.
- You can now begin to work with your new project.

Implement the lexical analysis. The relevant section is §6.4. The lexer uses the maximal munch strategy (§6.4p4). The project does not incorporate a preprocessor. Therefore we are only considering *token*s, not *preprocessing-token*s. Furthermore, *universal-character-name*s (§6.4.3) are omitted. For *constant*s (§6.4.4) we process only *decimal-constant*s, 0 and *character-constant*s. In this connection we ignore *integer-suffix*es and prefixes for the encoding (`L`, ...). *character-constant*s are restricted to single characters. For *string-literal*s (§6.4.5) the *encoding-prefix* is also omitted. Also we elide *octal-escape-sequence*s and *hexadecimal-escape-sequence*s.

The phases of translation are presented in §5.1.1.2. We leave out several parts. In phase 1 trigraphs are elided. For the input and execution encodings we use ISO-8859-1. Phases 2, 4 and 6 are left out. It is not necessary to implement the remaining phases in detail. Some of them may be combined.

To test this part, output the tokens. For this use the switch `--tokenize`, e.g. `c4 --tokenize test.c`. Output one token per line: First the position in the source (terminated by a colon), a single space, then the kind of token, another single space, and last the textual representation of the token. Make sure to output string- or character-constants exactly like they appear in the input stream, so do not replace escape sequences or digraphs in the output. For naming the tokens, use the non-terminals on the right hand side of the productions of *token* (§6.4p1). Consider the example input file test.c:

```
42␣␣if
␣␣"bla\n"x+
```

Output:

```
test.c:1:1: constant 42
test.c:1:5: keyword if
test.c:2:2: string-literal "bla\n"
test.c:2:9: identifier x
test.c:2:10: punctuator +
```

Remarks and hints:

- The soft deadline for this milestone is 2017-11-03.
- Keep it simple!
- Files that contain lexical errors must be rejected with an error message showing the location of the error.
- The running time of your lexer must be proportional to the input size.
- Write more test cases, especially some that test the rejection capabilities of your lexer.
- §6.4.2.1p4 suggests a neat way for the implementation of the detection of keywords.
- Do not intermingle lexing with the output of the tokens. The interface to read tokens will be used again by the parser.
- You may also use an existing lexer generator to create (parts of) the source code for this task.