

Exercise Sheet 1

Exercise 1.1 Regular Expressions and Finite Automata

- For the regular expression $(a|b)^*bac(a|bc)^*$ over the alphabet $\Sigma = \{a, b, c\}$ construct:
 - a nondeterministic finite automaton,
 - a deterministic finite automaton, and
 - a minimal deterministic finite automaton
 following the proof sketches from the lecture.
- 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
 - q and q_1 fails.
 - q_2 and p fails.
- 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).

Exercise 1.2 Greed

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

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!

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 should print an 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 *tokens*, not *preprocessing-tokens*. Furthermore, *universal-character-names* (§6.4.3) are omitted. For *constants* (§6.4.4) we process only *decimal-constants*, `0` and *character-constants*. In this connection we ignore *integer-suffixes* and prefixes for the encoding (`L, ...`). *character-constants* are restricted to single characters. For *string-literals* (§6.4.5) the *encoding-prefix* is also omitted. Also we elide *octal-escape-sequences* and *hexadecimal-escape-sequences*.

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.