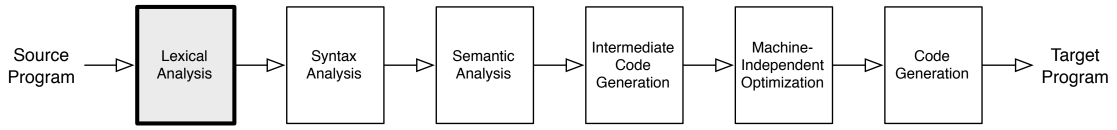
CS 301 Spring 2021 T4
Tutorial Assignment
15 April

Regular Expressions, Finite Automata, Lexical Analysis

1 Plan



Our first focus is on two topics:

- **Compilation Overview.** The first reading expands on our exploration of the compilation pipeline with TINY, considers how programming language design and computer language ideas. architecture affect compiler design, and surveys key programming
- **Lexical Analysis.** The first step in compilation is *lexical analysis*, turning a string of source code characters into a stream of meaningful *lexemes* or *tokens* in the language. The reading introduces the theory of lexical analysis (regular expressions, transition diagrams, finite automata) and how this connects to implementation. The first project stage applies these ideas in developing a full lexical specification as input to a tool for automatic lexer generation.

2 Readings

"Alternative" readings are an optional second perspective if the primary option is not a good fit for you.

- Skim Dragon 1 or EC 1
- Dragon 3.1.0–3.1.2, 3.3, 3.5–3.7
  Alternative: EC 2.1–2.4.3, 2.4.5
- Dragon 3.8, 3.9.6, 3.10
  Alternative: EC 2.4.4-2.4.5, 2.5.1
- *Regular Expression Matching Can Be Simple And Fast (but is slow in Java, Perl, PHP, Python, Ruby, ...).* Russ Cox, January 2007. `https://swtch.com/~rsc/regexp/regexp1.html`
  Read for basic ideas and commentary. The C code is less important.
- **Extra Depth** if you are curious about the general algorithm behind the ad hoc method used to transform the C-comment DFA into a regular expression: see *Kleene's construction* in EC 2.6.1.

3 Exercises

"Extra Depth" exercises are for the curious. In most tutorial meetings we will not reach them. Remember that solving even all the non-extra exercises is not necessarily expected.

1. (From Dragon Exercise 3.3.2) Describe (in English) the languages denoted by the following regular expressions:
   (a) $a(a|b)^*$
   (b) $((\epsilon|a)b^*)^*$
   (c) $(a|b)^*a(a|b)(a|b)$

(d) a*ba*ba*ba*

(e) (aa|bb)*((ab|ba)(aa|bb)*(ab|ba)(aa|bb)*)*

2. Write a regular expression for HTTP and FTP URLs. A URL consists of four parts: the protocol (`http://` or `ftp://`), the domain name or the IP address of a host, an optional port number, and an optional pathname for a file. For simplicity, we assume that:
   - A domain name is a list of non-empty alphabetical strings separated by periods.
   - An IP address is four non-negative integers of at most three digits each, separated by periods.
   - A port number is a positive integer following a colon. (*e.g.*, :8080)
   - The pathname is a Unix-style absolute pathname. The allowed symbols are letters, digits, period, slash. A sequence of two consecutive slashes `//` is forbidden, *i.e.*, no empty directory name.
   - A URL may end with a slash as long as it does not create the sequence `//`.

3. Write the DFAs for each of the following:

   (a) Binary numbers that contain the substring 011.

   (b) Binary numbers that are multiples of 3 and have no consecutive 1's. Your solution can accept or reject the empty string – either is fine. (You may find it easiest to create DFAs for each of the two requirements and then think about how to combine them.)

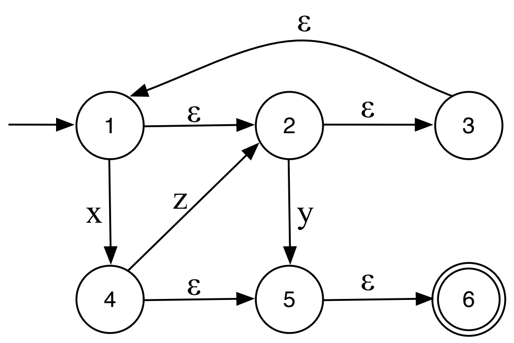
4. A block comment in the C language begins with the two-character sequence `/*`, followed by the body of the comment, and then the sequence `*/`. The body of the comment may not contain the sequence `*/`, although it may contain the sequence `/*`, or the characters `*` and `/`. We use the notation $(E)^*$ for the Kleene closure of E; all other occurrences of `*` refer to the character itself, not to the Kleene operator.

   (a) Show that the following regular expression does not correctly describe C comments:

$$\texttt{/*}\;(\texttt{/})^*\left([\,^\wedge\texttt{*/}] \;\middle|\; [\,^\wedge\texttt{*}]\texttt{/} \;\middle|\; \texttt{*}[\,^\wedge\texttt{/}]\right)^*\;(\texttt{*})^*\;\texttt{*/}$$

   (b) Draw the DFA that accepts C comments and then use it to write the regular expression that correctly describes C comments.

5. Convert the following NFA to a DFA. For each DFA state, indicate the set of NFA states to which it corresponds. Make sure you show the initial state and the final/accept states in the constructed DFA.



6. Consider the following lexical analysis specification:

```
(aba)+   { return Tok1; }
(a(b)+a) { return Tok2; }
(a|b)    { return Tok3; }
```

In case of tokens with the same length, the token whose pattern occurs first in the above list is returned.

(a) Use **Thompson's construction** to build an NFA that accepts strings that match any of the above three patterns.

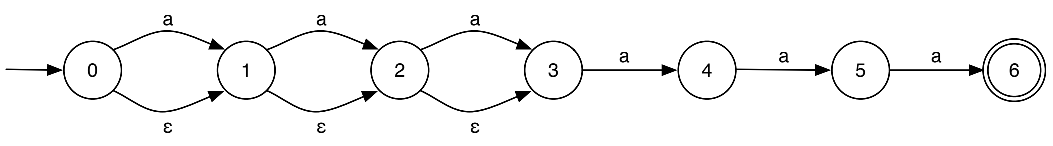
(b) Use the **subset construction** to transform your NFA into a DFA. Label the DFA states with the set of NFA states to which they correspond. Indicate the final/accept states in the DFA and label each of these states with the (unique) token being returned in that state.

(c) Show the steps in the functioning of the lexer for the input string `abaabbaba`. Indicate what tokens the lexer returns for successive calls to `next_token()`. For each of these calls indicate the DFA states being traversed in the automaton.

7. **Extra Depth:** Russ Cox describes how inefficient some regular expression pattern matchers can be.

   To motivate the design and to see how a little bit of theory can dramatically improve software design, we start by comparing the two simulation algorithms in section "Regular Expression Search Algorithms" section of Cox's article.

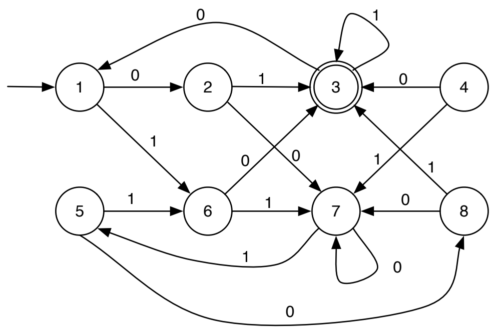
   Here is the NFA corresponding to "`a?a?a?aaa`":



   (a) Enumerate all paths taken through the NFA using the backtracking search algorithm on input `aaab`. How many are there? Given the NFA for $(\texttt{a}?)^n\texttt{a}^n$, how many paths may need to be explored to test an input string of length $n+1$? Give a Big-O bound and a one or two sentence explanation.

   (b) Show the steps performed by Algorithm 3.22 in Dragon. (This is a more precise and succinct description of Cox's second algorithm). It suffices to show the states in S each time line (3) is executed. You will find it useful to compute the ϵ-closure for each state in the NFA.

8. **Extra Depth:** Minimize the states of the following DFA. Label each state in the minimized DFA with the set of states from the original DFA to which it corresponds.


1
2
3
4
5
6
7
8
0
1
0
1
0
1
0
0
1
1
1
0
1
0
0
0
1