CS 164 Programming Language and Compilers **Fall 2015**

Discussion 2: Lexical Analysis

TAs: Melanie Cebula, Ben Mehne, Matt Miller

- **Announcement**
  - WA1 due Wed (9/16) in class.
  - PA2 due Thu (9/17) at 11:59pm.
  - WA2/PA3 out Wed (9/16)
- **Outline**
  1. Understanding **regular expression**
  2. Regular expression with **backreference**
  3. Understanding **finite automata**.

---

1 Regular Expression

For lexical analysis, we care about regular languages. A regular language is a language which can be recognized while using a finite amount of memory. Specifically, a regular language can be described by a regular expression.

1. Vocabulary for Lexical Analysis:
   - **Alphabet:** a finite (we will not be considering infinite) set of symbols (characters).
   - **String:** a finite sequence of symbols from some alphabet.
   - **Language:** a set of strings.
   - **Token:** a pair consisting of a token name (syntactic category) and an optional attribute (semantic information).
   - **Pattern:** a description of the form that lexemes of a token might take.
   - **Lexeme:** a string that matches the pattern of a token.

2. What are the five core constructs used to define regular expressions/languages?
   Answer: Alphabet, concatenation, union, Kleene-star, and empty-string

3. What language is denoted by each of the following regular expressions? Try writing down a few simple strings and give a concise description of the language.
   (a) `[_a-zA-Z][_a-zA-Z0-9]*`
   Answer: The language of strings that are made up of alphanumerics and underscores and that start with a letter or underscore (Identifiers in many languages)

   (b) `((`ϵ`|a)b*)*` (Could you simplify this expression?)
   Answer: All possible sequences with characters a and b

   (c) `(0+1+)+|(1+0+)+`
   Answer: Binary strings with different first and last bits.

4. Write a regular expression for the following languages:
   (a) Binary numbers.
   Answer: `0|1(0|1)*`.

   (b) Even binary numbers. (read from left to right – i.e.,“1101” is 13).
   Answer: “`[01]*0`” or “`1[01]*0|0`”, depending on whether we allow leading zeros.

   (c) All strings of lowercase letters in with the letters are in ascending lexicographic order.
   Answer: `a*b*c*....y*z*`

   (d) Java-like comments, consisting of a string surrounded by `/*` and `*/` without an intervening `*/` unless it is inside double quotes (`"`). You can escape meta-characters with a backslash.
   Answer: A simple example, only allowing letters in comments, would be:
   `\/\*([\w\s]?("([\w\s]?(\*\/)?)*")?)*\*\/`

   (e) Strings with balanced parentheses (e.g., ((()))).
   Answer: It is not possible.

---

2 Regular Expression with Backreference

Backreferences match the same text as previously matched by a capturing group. In the basic GNU regular expression, a subexpression enclosed in parenthesis is a capturing group. For example, in regular expression `(\w)\1`, subexpression `\w` is a capturing group, and the matched text is referred as the first reference `\1`. Therefore, `(\w)\1` could match any pair of repeated letters: “aa”, “bb”, etc. As you might guess, `\2` refers to the second matched group and so forth.

5. Write a regular expression for the following languages:
   (a) Suppose now you have a large CSV(Comma Separated Values) file, each row of which has the form `firstname,lastname,email`. You want to find out all uninteresting people who use boring email address of the form `firstname+lastname@xxx.com` and avoid making friends with them. Try to write a normal (without backreference) regular expression that describes this pattern, or state it's impossible.
   Answer: Not possible. Backreferences require backtracking, which cannot be done with finite memory.

   (b) Now try to write expression for the mentioned pattern using backreference.
   Answer: `(\w+),(\w+),\1\2@`

---

3 Finite Automata

Finite automata[1] can be used to implement regular languages. A finite automaton is defined as a tuple $(\Sigma, S, n, F, \delta)$, where Σ is the alphabet, S is the set of state, $n \in S$ is the start state, $F \subseteq S$ is the set of accepting states, and $\delta : S \times \Sigma \to S$ is the transition relation.

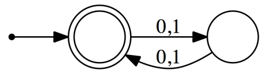
[1] *Automata* is plural. The singular form is *automaton*

Theorem 1 *Theorem: A language is regular if and only if it is accepted by a finite automaton.*

6. Miscellaneous questions:
   - How does NFA differ from a DFA?
   - When does NFA and DFA accept its input?
   - Which is more powerful?
   - Why would you use the non-determinism in an NFA?

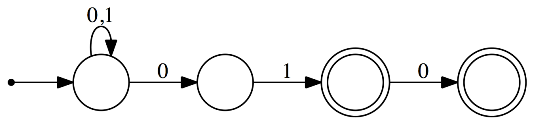
   Answers:
   - DFA (deterministic) means that for a given input, it can only be in and transition to one state at a time. NFA (nondeterministic) means that for a given input it can transition to, and be in multiple states at once.
   - A DFA accepts if the word ends in an accept state. An NFA accepts if for the set of final states the word could end up in, at least one of them is an accept state.
   - You can convert from a DFA to an NFA and vice versa. But you can think of a DFA as a special case NFA. For each NFA, there is a DFA such that both recognize the same formal language
   - Some problems are easier to reason about nondeterministically. Example: a(bab)*|a(ba)*; In an NFA we can have two different "a" transitions from the start state to the two subexpressions.

7. What language is accepted by the following DFA?



   Answer: Binary strings of even length.

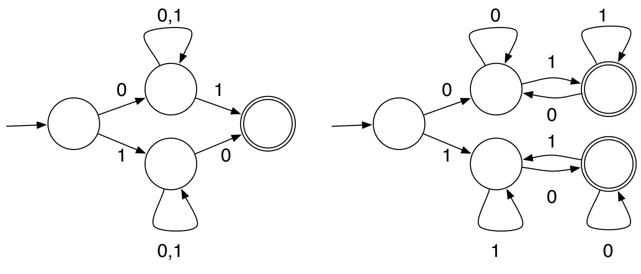
8. What language is accepted by the following NFA?



   Answer: Binary strings ending in 01 or 010

9. Construct DFA and NFA for binary strings with different first and last bits.

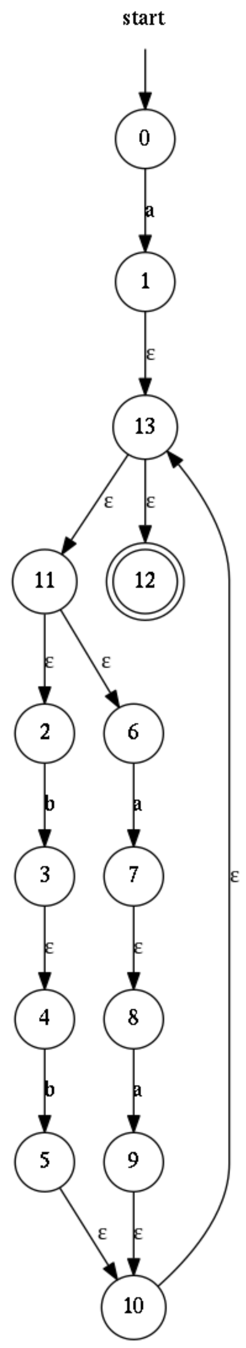
   Answer:

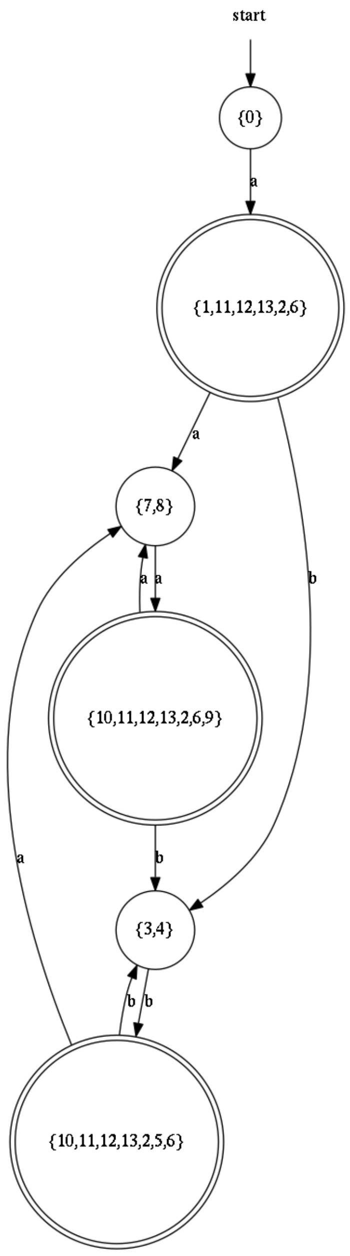


NFA DFA

10. Convert regular expression $a(bb|cc)*$ to a NFA using **Thompson construction**.

11. Convert the NFA from Exercise 10 to a DFA using **subset construction**.
Answers to 10 and 11:


start
0
a
1
ε
13
ε
ε
11
12
ε
ε
2
6
b
a
3
7
ε
ε
4
8
b
a
5
9
ε
ε
10
ε


start
{0}
a
{1,11,12,13,2,6}
a
b
{7,8}
a
a
{10,11,12,13,2,6,9}
b
a
{3,4}
b
b
{10,11,12,13,2,5,6}