CS 131 Compilers: Discussion 2: (Non)Deterministic Finite Automata

杨易为 季杨彪 尤存翰
{yangyw,jiyb,youch}@shanghaitech.edu.cn

2021 年 3 月 8 日

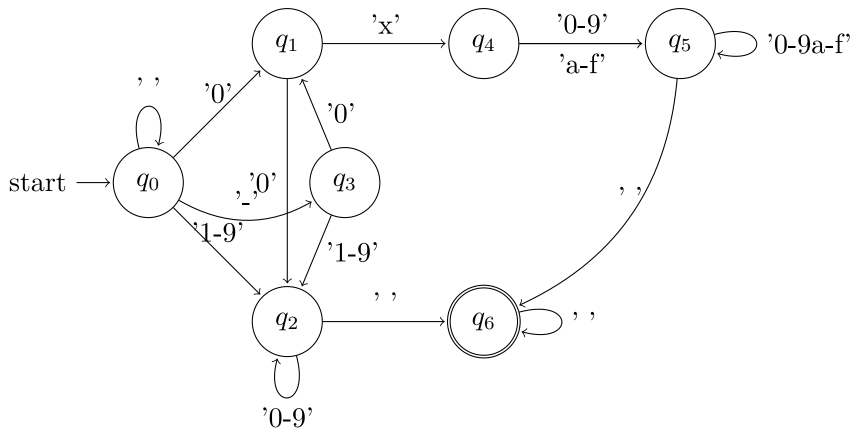
1 DFA and NFA

1.1 Introduction

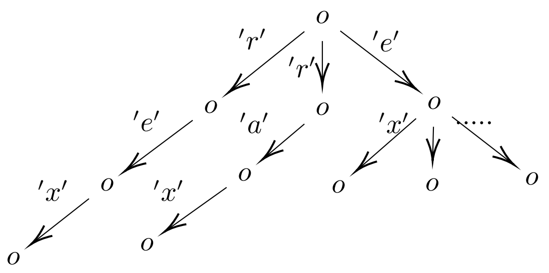
Finite state automata (FSA) are abstract machines that feature states and guarded tran-sitions from one state to another. An FSA can only be in one state a time, and its totalnumber of states isfinite. The machine takes transitions in response to inputs it receivessequentially; if an input matches the guard of a transition that departs from the currentstate that transition is said to be enabled; only enabled transitions can be taken. FSA thathave an accepting state provide the machinery to determine whether an input string is in aregular language. If no transitions are enabled by a given input then the entire input stringgets rejected. If all inputs are processed and the FSA is in an accepting state then theentire input string is accepted. Deterministic finite state automata (DFA) can only haveone enabled transition at a time while a non-deterministic finite state automata (NFA) canhave multiple.
Before NLP, we utilize state machine to accept languages. Two different perspectives can be considered, namely symbolic and random. chomsky's formal language theory embodies a symbolic approach. Based on this view, a language contains a sequence of symbols that must follow the syntactic rules of its generative grammar. This view reduces the structure of a language to a set of well-defined rules that allow for the decomposition of each sentence and word into structural components.

1.2 Examples of accepting x86 Assembly

https://github.com/yangminz/bcst_csapp
There are three main types of terminators for assembly instructions: Immediately and Scale for numbers, Reg and Op for characters, and other symbols, including spaces (' '), commas (','), and parentheses ('(',')'). For numeric terminators, we can use Deterministic Finite Automation (DFA) to determine and parse them according to Regular Grammar, see below Figure. It should be noted that the numbers in assembly instructions include positive and negative integers in hexadecimal and decimal, and We need to check whether the value overflows at each state transfer step.



For character type terminators, i.e. operands and registers, we can take a simple string matching (Matching). Here, we have a variety of data structures to employ. The most common approach is to use an array or list that stores all possible string values, e.g., {"rax","eax"} and {"mov","ld","sd"}, and iterate through this array and match each element. With such a data structure, the time complexity of finding and matching is $O(LE[strlen])$, where $E[strlen]$ is the average length of the string. Another strategy is to build a dictionary tree (Trie) or a prefix tree for the above array of strings.



By using prefix trees, searching and matching can be done without traversing the entire array, and the time complexity is reduced to $O(E[strlen])$. In the implementation, a prefix tree can be created for all instruction operator names and register names before executing the CPU instruction cycle, so that the matching time can be reduced during the execution of the instruction cycle. Each node of the prefix tree usually includes a char→(void *) mapped data structure, where char is the character and (void *) is the address of the subtree. As you can see, the prefix tree is very similar to the DFA, except that the prefix tree automatically builds the state transfer (so it is not streamlined enough), while the DFA requires us to calculate the state transfer in advance.

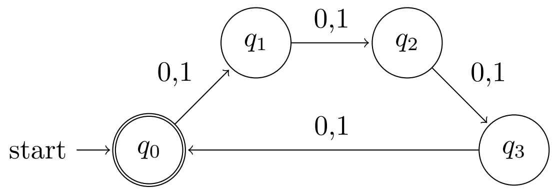
1.3 Examples of accepting yml

We cannot do it, because of indent can only be accepted with stack machine.

```
// Identity definitions
<whitespace> ::= "␣" | "\t"
<in-line-whitespace> ::= <whitespace> | <whitespace> <in-line-whitespace>
<digit> ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
<integer-number> ::= <digit> | <digit> <integer-number>
<end-of-line> ::= "\r\n" | "\r" | "\n"
<non-whitespace> ::=
<non-whitespace-word> ::= <non-whitespace> | <non-whitespace> <non-whitespace-word>

// Basic document structure
<yaml-stream> ::= <yaml-document-list>
<yaml-document-list> ::= <yaml-document> | <yaml-document> <yaml-document-list>
<yaml-document> ::= <yaml-directive-list> <yaml-directives-end> <yaml-content-list>
        <yaml-document-end>
<yaml-directive-list> ::= <yaml-directive> <yaml-directive-list> | ""
```

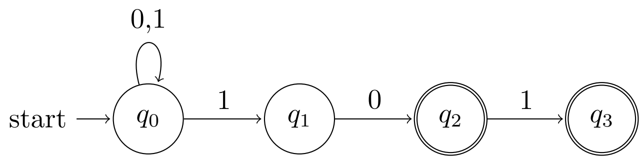
```
<yaml-directive> ::= <yaml-directive-yaml> | <yaml-directive-tag>
<yaml-directive-yaml> ::= "%YAML" <in-line-whitespace> <integer-number> "." <integer-number>
          <end-of-line> | ""
<yaml-directive-tag> ::= "%TAG" <in-line-whitespace> <yaml-tag-handle> <in-line-whitespace>
          <yaml-tag-prefix> <end-of-line> <yaml-directive-tag> | ""
<yaml-tag-handle> ::= "!" | "!!" | "!" <non-whitespace-word> "!"
<yaml-tag-prefix> ::= "!" <non-whitespace-word> | <non-whitespace-word>
<yaml-directives-end> ::= "---" | ""
<yaml-content-list> ::= <yaml-content> <yaml-content-list> | <yaml-content> <end-of-line>
<yaml-document-end> ::= "..." | ""

// Internals of an individual document
<yaml-content> ::=
```

1.4 What language is accepted by the following DFA?



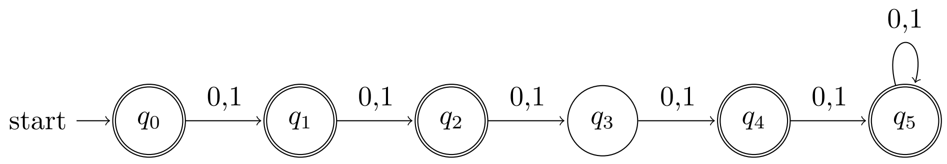
Answer:

1.5 What language is accepted by the following NFA?



Answer:

1.6 What language is accepted by the following NFA?



Answer:

1.7 Construct the NFA that accepts

1.7.1 $x(zy?|(yz)*)$

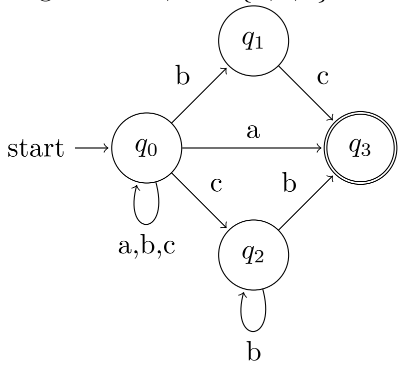
Answer:

1.7.2 $(10) * 1 * 01 * 0$

Answer:

1.8 Minimized the NFA

1. Original NFA, $\Sigma = \{a, b, c\}$:



Answer:

2 Pumpingg Lemma

Pumping Lemma: For any DFA (or NFA or regular expression) that accepts an infinite number of strings, there is some minimum length, M, such that any string with length greater than or equal to M that the machine accepts must have the form uxv, where u, x, and v are strings, x is not empty, the length of ux is $\leq M$, and the machine accepts all strings of the form ux^nv, for all $n \geq 0$.

2.1 Proof of not regular

Let $A = \{1^jz \mid z \in \{0,1\}^*$ and z contains at most j i's, for any $j \geq 1\}$. Prove, by the pumping lemma, that A is not regular.
Answer: