Lexical Analysis

Lecture 2

Announcements

- WA1
  - Assigned today (03 June)
- PA1
  - Assigned today

Lexical Analysis

- What do we want to do? Example:

```
if (i == j)
    Z = 0;
else
    Z = 1;
```

- The input is just a string of characters:

  \tif (i == j)\n\t\tz = 0;\n\telse\n\t\tz = 1;

- Goal: Partition input string into substrings
  - Where the substrings are tokens

What's a Token?

- A syntactic category
  - In English:

    noun, verb, adjective, …

  - In a programming language:

    Identifier, Integer, Keyword, Whitespace, …

Tokens

- Tokens correspond to sets of strings.

- Identifier: *strings of letters or digits, starting with a letter*
- Integer: *a non-empty string of digits*
- Keyword: *"else" or "if" or "begin" or …*
- Whitespace: *a non-empty sequence of blanks, newlines, and tabs*

What are Tokens For?

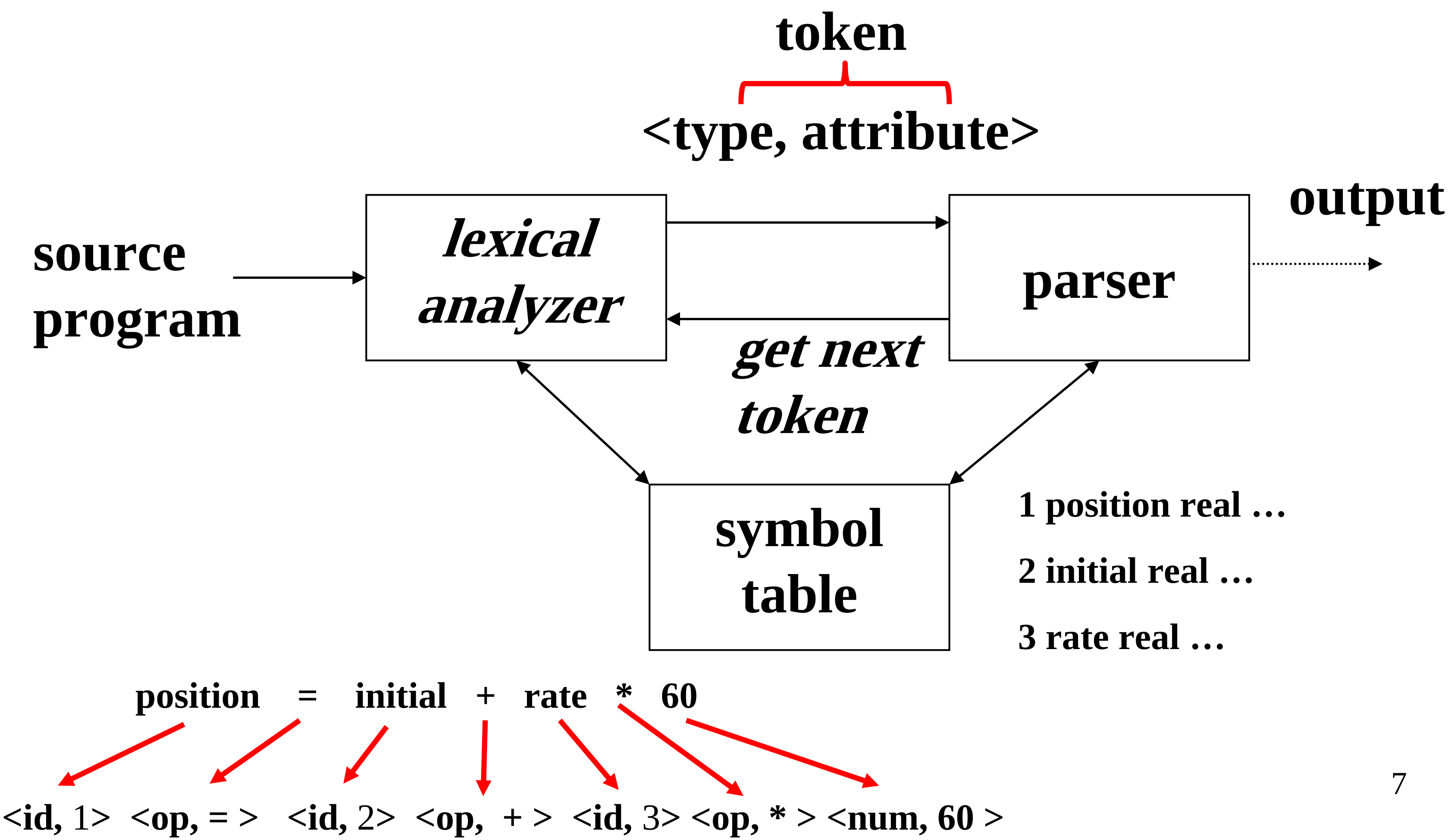
- Classify program substrings according to role

- Output of lexical analysis is a stream of tokens . . .

- . . . which is input to the parser

- Parser relies on token distinctions
  - An identifier is treated differently than a keyword

Lexical Analyzer in Perspective



Example

- Recall

  \tif   (i == j)\n\t\tz = 0;\n\telse\n\t\tz = 1;

  W K W (I   R I)   W   I = N ;   W   K   W   I = N ;

- Useful tokens for this expression:

  Number, Keyword, Relation, Identifier, Whitespace, (,), =, ;

- N.B., (,), =, ; are tokens, not characters, here

Designing a Lexical Analyzer: Step 1

- Define a finite set of tokens
  - Tokens describe all items of interest
  - Choice of tokens depends on language, design of parser

Designing a Lexical Analyzer: Step 2

- Describe which strings belong to each token

- Recall:
  - Identifier: *strings of letters or digits, starting with a letter*
  - Integer: *a non-empty string of digits*
  - Keyword: *"else" or "if" or "begin" or …*
  - Whitespace: *a non-empty sequence of blanks, newlines, and tabs*

Lexical Analyzer: Implementation

- An implementation must do two things:

1. Recognize substrings corresponding to tokens

   The *lexemes*

2. Return the token class of each lexeme

   $\underbrace{\text{<Token class, lexeme>}}_{\text{Token}}$

Lexical Analyzer: Implementation

- The lexer usually discards “uninteresting” tokens that don’t contribute to parsing.

- Examples: Whitespace, Comments

True Crimes of Lexical Analysis

- Is it as easy as it sounds?

- Not quite!

- Look at some history . . .

Lexical Analysis in FORTRAN

- FORTRAN rule: Whitespace is insignificant
- E.g., VAR1 is the same as VA   R1
- A terrible design!

Example

- Consider
  - DO 5 I = 1,25
  - DO 5 I = 1.25

$\xrightarrow{\hspace{4em}}$
Lookahead

Fortran loop

```
DO 5 I = 1,25
   ...
5 ...
```

Fortran assignment

```
DO 5 I = 1.25
```

Lexical Analysis in FORTRAN (Cont.)

- Two important points:
  1. The goal is to partition the string. This is implemented by reading left-to-write, recognizing one token at a time

  2. “Lookahead” may be required to decide where one token ends and the next token begins

Lookahead

- Even our simple example has lookahead issues

  i vs. if

  = vs. ==

- Footnote: FORTRAN Whitespace rule motivated by inaccuracy of punch card operators

Lexical Analysis in PL/I

- PL/I keywords are not reserved

IF ELSE THEN THEN = ELSE; ELSE ELSE = THEN

Keyword Keyword Keyword

Lexical Analysis in PL/I (Cont.)

- PL/I Declarations:

  DECLARE (ARG1,. . ., ARGN)

- Can't tell whether DECLARE is a keyword or array reference until after the).
  - Requires arbitrary/unbounded lookahead!

Review

- The goal of lexical analysis is to
  - Partition the input string into lexemes
  - Identify the token of each lexeme

- Left-to-right scan => lookahead sometimes required

Next

- We still need
  - A way to describe the lexemes of each token
  - A way to resolve ambiguities
    - Is if two variables i and f?
    - Is == two equal signs = =?

Regular Languages

- There are several formalisms for specifying tokens

- *Regular languages* are the most popular
  - Simple and useful theory
  - Easy to understand
  - Efficient implementations

Languages

Def. Let Σ be a set of characters. *A language over* Σ is a set of strings of characters drawn from Σ

Notation

- Languages are sets of strings.
- Need some notation for specifying which sets we want
- The standard notation for regular languages is *regular expressions.*

Atomic Regular Expressions

- Single character

$$'c' = \{"c"\}$$

- Epsilon

$$\varepsilon = \{""\}$$

Compound Regular Expressions

- Union

$$A + B = \{ s \mid s \in A \text{ or } s \in B \}$$

- Concatenation

$$AB = \{ ab \mid a \in A \text{ and } b \in B \}$$

- Iteration

$$A^* = \bigcup_{i \geq 0} A^i \quad \text{where } A^i = A \ldots i \text{ times} \ldots A$$

Regular Expressions

- **Def.** The *regular expressions over* Σ are the smallest set of expressions including

$$
\begin{array}{ll}
\varepsilon & \\
'c' & \text{where } c \in \Sigma \\
A + B & \text{where } A, B \text{ are rexp over } \Sigma \\
AB & \text{"} \qquad\qquad\qquad \text{"} \qquad\qquad\qquad \text{"} \\
A^{*} & \text{where } A \text{ is a rexp over } \Sigma
\end{array}
$$

Syntax vs. Semantics

- To be careful, we should distinguish syntax and semantics.

$$
\begin{aligned}
L(\varepsilon) &= \{""\} \\
L('c') &= \{"c"\} \\
L(A+B) &= L(A) \cup L(B) \\
L(AB) &= \{ab \mid a \in L(A) \text{ and } b \in L(B)\} \\
L(A^*) &= \bigcup_{i \geq 0} L(A^i)
\end{aligned}
$$

Example: Keyword

Keyword: *"else" or "if" or "begin" or …*

'else' + 'if' + 'begin' + . . .

Note: 'else' abbreviates 'e''l''s''e'

Example: Integers

Integer: *a non-empty string of digits*

$$\text{digit} = \text{'0'}+\text{'1'}+\text{'2'}+\text{'3'}+\text{'4'}+\text{'5'}+\text{'6'}+\text{'7'}+\text{'8'}+\text{'9'}$$

$$\text{integer} = \text{digit digit}^*$$

Abbreviation: $A^+ = AA^*$

Example: Identifier

Identifier: *strings of letters or digits, starting with a letter*

letter = 'A' + . . . + 'Z' + 'a' + . . . + 'z'

identifier = letter (letter + digit)*

letter = [a-zA-Z]

Is (letter* + digit*) the same?

Example: Whitespace

Whitespace: *a non-empty sequence of blanks, newlines, and tabs*

$$(\text{' '} + \text{'\textbackslash n'} + \text{'\textbackslash t'})^{+}$$

Example: Email Addresses

- Consider *anyone@cs.stanford.edu*

$$\Sigma = \text{letters} \cup \{.,@\}$$

$$\text{name} = \text{letter}^{+}$$

$$\text{address} = \text{name } '@' \text{ name } '.' \text{ name } '.' \text{ name}$$

Example: Unsigned Pascal Numbers

digit	=	'0' +'1'+'2'+'3'+'4'+'5'+'6'+'7'+'8'+'9'
digits	=	digit$^+$
opt_fraction	=	('.' digits) + ε = ('.' digits)?
opt_exponent	=	('E' ('+' + '-' + ε) digits) + ε
num	=	digits opt_fraction opt_exponent

Summary

- Regular expressions describe many useful languages

- Regular languages are a language specification
  - We still need an implementation

- Next: Given a string *s* and a rexp *R*, is

$$s \in L(R)?$$

Lexical Specification

Notation

- There is variation in regular expression notation

- At least one: $A^+ \equiv AA^*$
- Union: $A \mid B \equiv A + B$
- Option: $A + \varepsilon \equiv A?$
- Range: 'a'+'b'+…+'z' $\equiv$ [a-z]
- Excluded range:

  complement of [a-z] $\equiv$ [^a-z]

Regular Expressions in Lexical Specification

- Last: a specification for the predicate

$$s \in \underbrace{L(R)}_{\text{Set of strings}}$$

- But a yes/no answer is not enough!
- Instead: partition the input into tokens

$$c_1c_2c_3 \mid c_4c_5c_6c_7 \mid \ldots$$

- We adapt regular expressions to this goal

Regular Expressions => Lexical Spec. (1)

1. Write a rexp for the lexemes of each token
   - Number = digit +
   - Keyword = 'if' + 'else' + …
   - Identifier = letter (letter + digit)*
   - OpenPar = '('
   - …

Regular Expressions => Lexical Spec. (2)

2. Construct R, matching all lexemes for all tokens

R = Keyword + Identifier + Number + …

$= R_1 + R_2 + \ldots$

Regular Expressions => Lexical Spec. (3)

3. Let input be $x_1 \ldots x_n$

   For $1 \leq i \leq n$ check

   $$x_1 \ldots x_i \in L(R)$$

4. If success, then we know that

   $x_1 \ldots x_i \in L(R_j)$ for some j

5. Remove $x_1 \ldots x_i$ from input and go to (3)

Ambiguities (1)

- There are ambiguities in the algorithm

- How much input is used? What if
  - $x_1 \ldots x_i \in L(R)$ and also
  - $x_1 \ldots x_K \in L(R)$

    $k \neq i$

- Rule: Pick longest possible string in $L(R)$
  - The "maximal munch"

Ambiguities (2)

- Which token is used? What if
  - $x_1 \ldots x_i \in L(R_j)$ and also
  - $x_1 \ldots x_i \in L(R_k)$
    $k \neq i$

$$R = R_1 + R_2 + R_3 + \ldots$$

Keyword = 'if' + 'else' + …

Identifier = letter (letter + digit)*

- Rule: use rule listed first (j if $j < k$)
  - Treats "if" as a keyword, not an identifier

Error Handling

- What if

  No rule matches a prefix of input ?

  $x_1 \ldots x_i \notin L(R_j)$

- Problem: Can't just get stuck …

- Solution:
  - Write a rule matching all "bad" strings
  - Put it last (lowest priority)

Finite Automata

- Regular expressions = specification
- Finite automata = implementation

- A finite automaton consists of
  - An input alphabet Σ
  - A set of states S
  - A start state n
  - A set of accepting states $F \subseteq S$
  - A set of transitions $state \rightarrow^{input} state$

Finite Automata

- Transition

$$s_1 \to^a s_2$$

- Is read

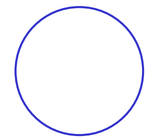
  In state s_1 on input "a" go to state s_2

- If end of input and in accepting state => accept

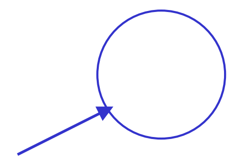
- Otherwise => reject
  - Terminates in a state s that is NOT an accepting state ($s \notin F$)
  - Gets stuck

Finite Automata State Graphs

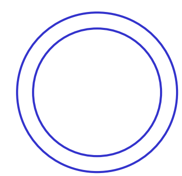
- A state



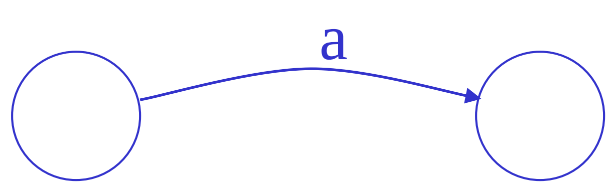
- The start state



- An accepting state

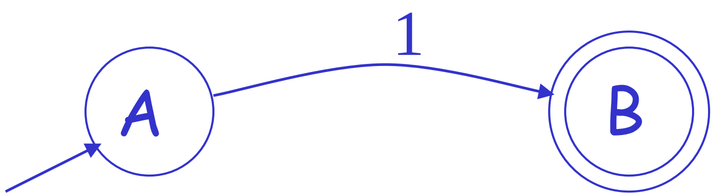


- A transition



A Simple Example

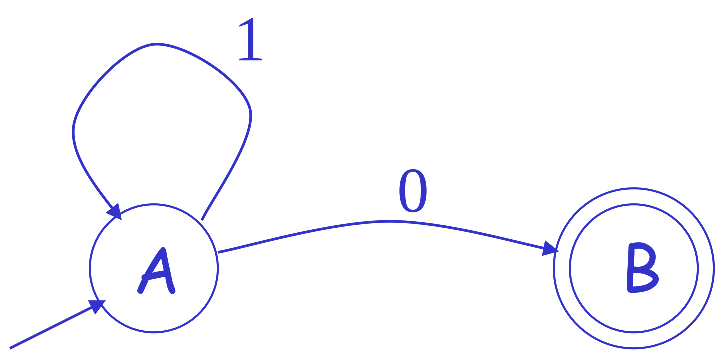
- A finite automaton that accepts only "1"



- Accepts '1' : ↑1, 1↑
- Rejects '0' : ↑0
- Rejects '10' : ↑1, 1↑0

Another Simple Example

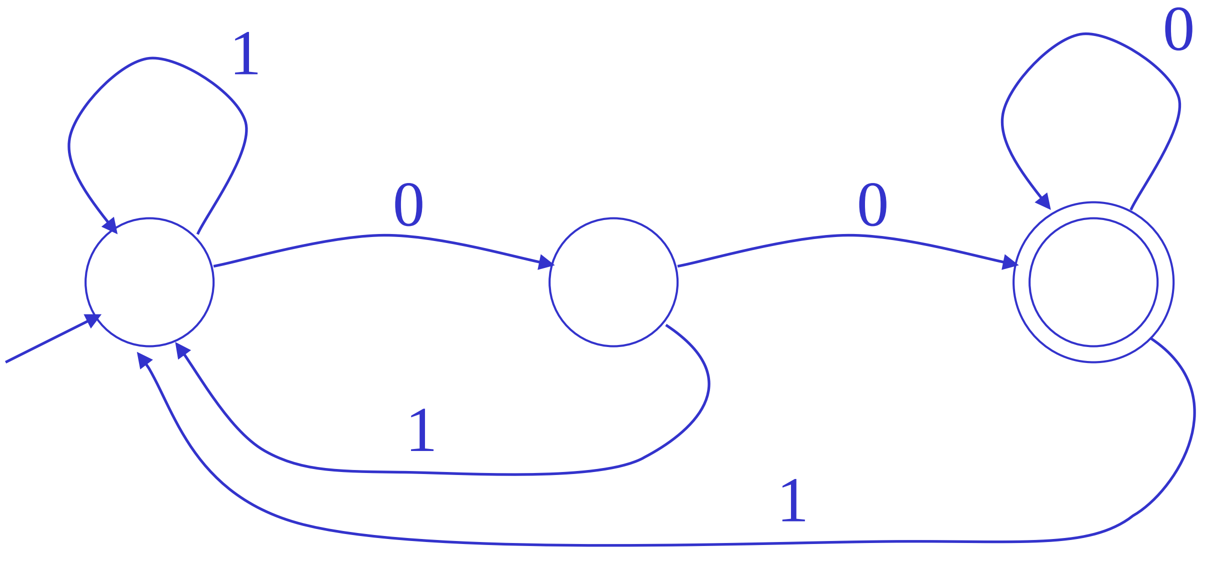
- A finite automaton accepting any number of 1's followed by a single 0
- Alphabet: {0,1}



- Accepts '110': ↑110, 1↑10, 11↑0, 110↑
- Rejects '100' : ↑100, 1↑00, 10↑0

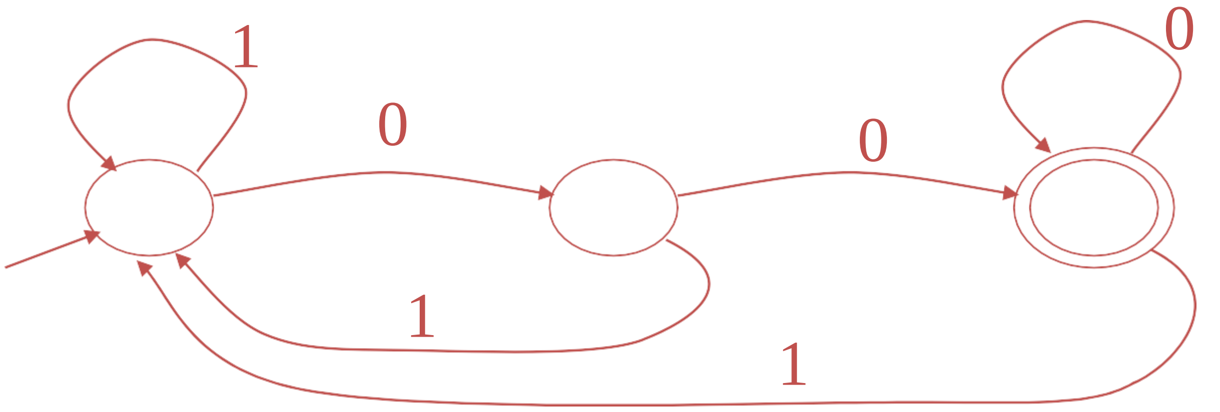
And Another Example

- Alphabet {0,1}
- What language does this recognize?



And Another Example

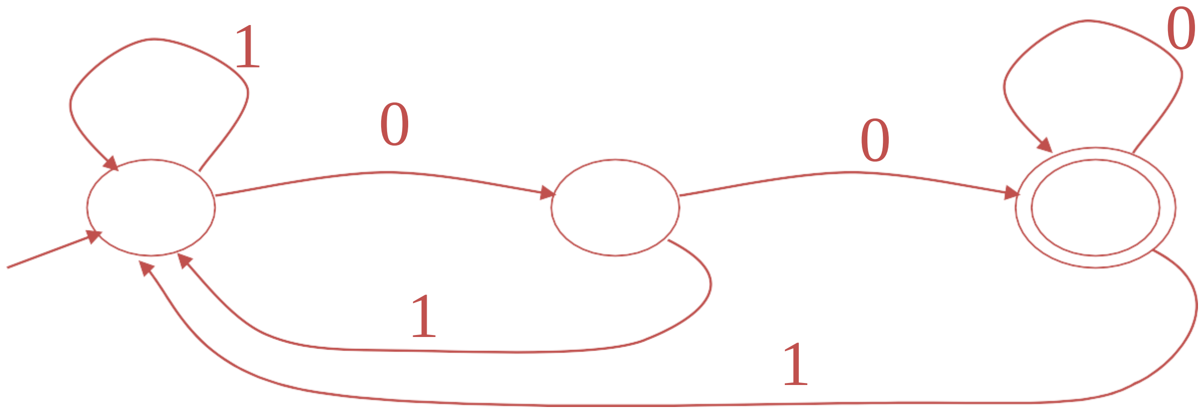
Select the regular language that denotes the same language as this finite automaton



- $(0 + 1)^*$
- $(1^* + 0)(1 + 0)$
- $1^* + (01)^* + (001)^* + (000^*1)^*$
- $(0 + 1)^*00$

And Another Example

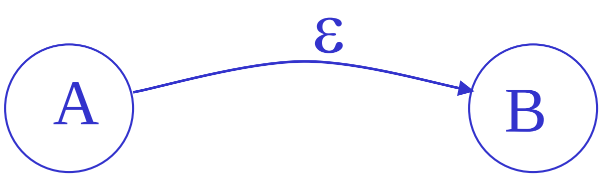
Select the regular language that denotes the same language as this finite automaton



- ○ $(0 + 1)^*$
- ○ $(1^* + 0)(1 + 0)$
- ○ $1^* + (01)^* + (001)^* + (000^*1)^*$
- ○ $\boxed{(0 + 1)^*00}$

Epsilon Moves

- Another kind of transition: ε-moves



- Machine can move from state A to state B without reading input

Deterministic and Nondeterministic Automata

- Deterministic Finite Automata (DFA)
  - One transition per input per state
  - No ε-moves
- Nondeterministic Finite Automata (NFA)
  - Can have multiple transitions for one input in a given state
  - Can have ε-moves

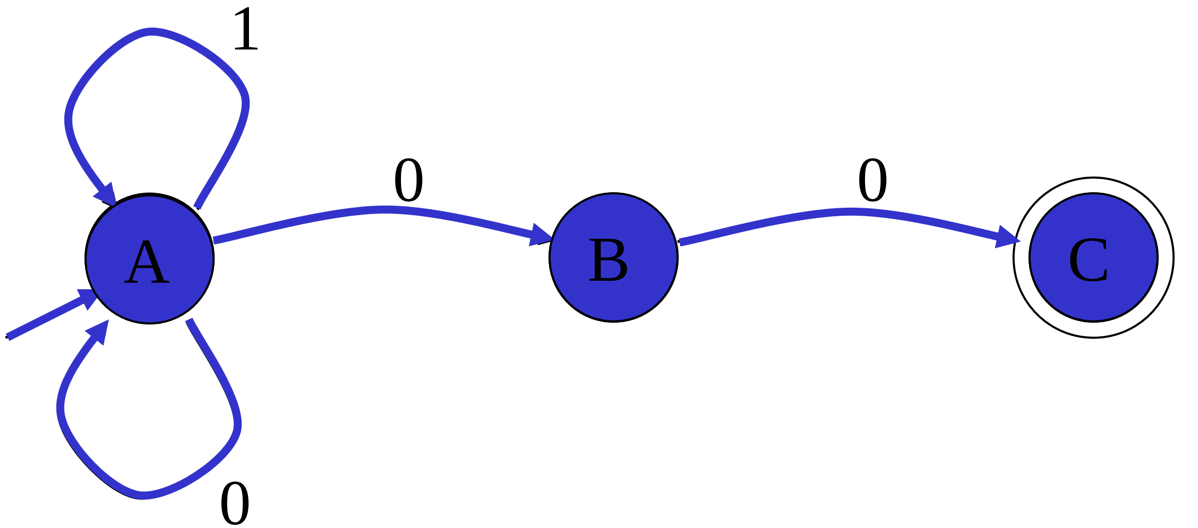
Execution of Finite Automata

- A DFA can take only one path through the state graph
  - Completely determined by input

- NFAs can choose
  - Whether to make ε-moves
  - Which of multiple transitions for a single input to take

Acceptance of NFAs

- An NFA can get into multiple states



- Input: 1 0 0
- Possible States: {A} {A, B} {A, B, C}

Rule: NFA accepts if it <u>can</u> get to a final state

NFA vs. DFA (1)

- NFAs and DFAs recognize the same set of languages (regular languages)

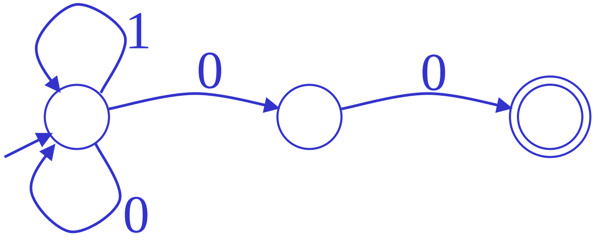
- DFAs are faster to execute
  - There are no choices to consider

- NFAs are, in general, smaller
  - Sometimes exponentially smaller

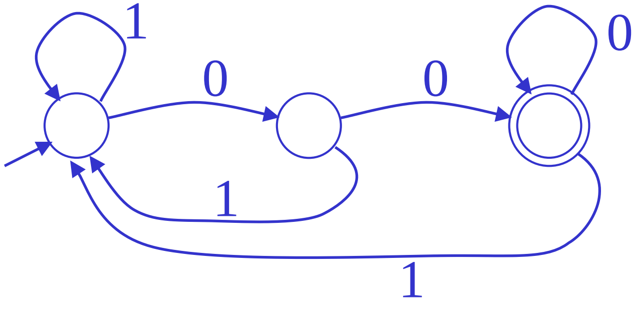
NFA vs. DFA (2)

- For a given language NFA can be simpler than DFA

NFA



DFA



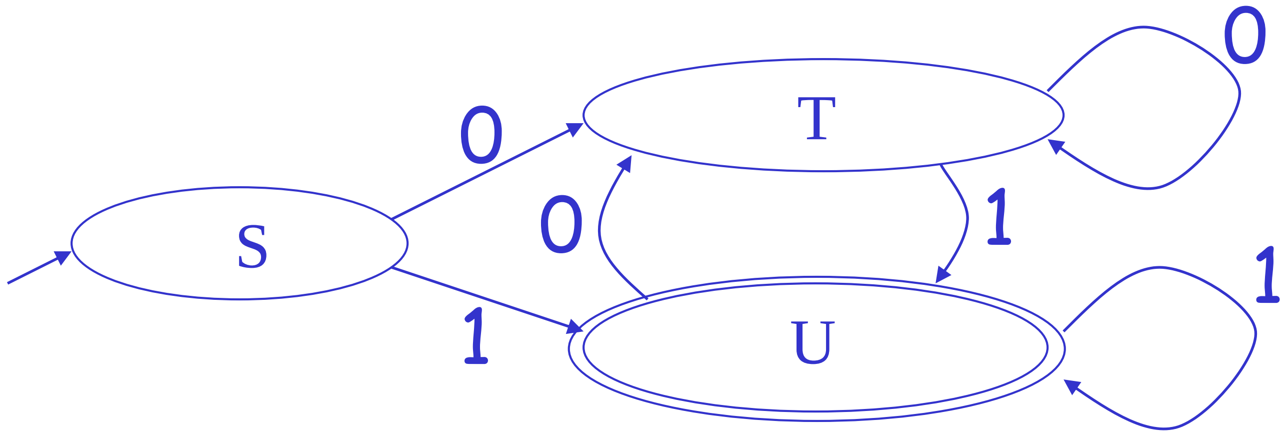
- DFA can be exponentially larger than NFA

Implementation

- A DFA can be implemented by a 2D table T
  - One dimension is "states"
  - Other dimension is "input symbol"
  - For every transition $S_i \to^a S_k$ define $T[i,a] = k$

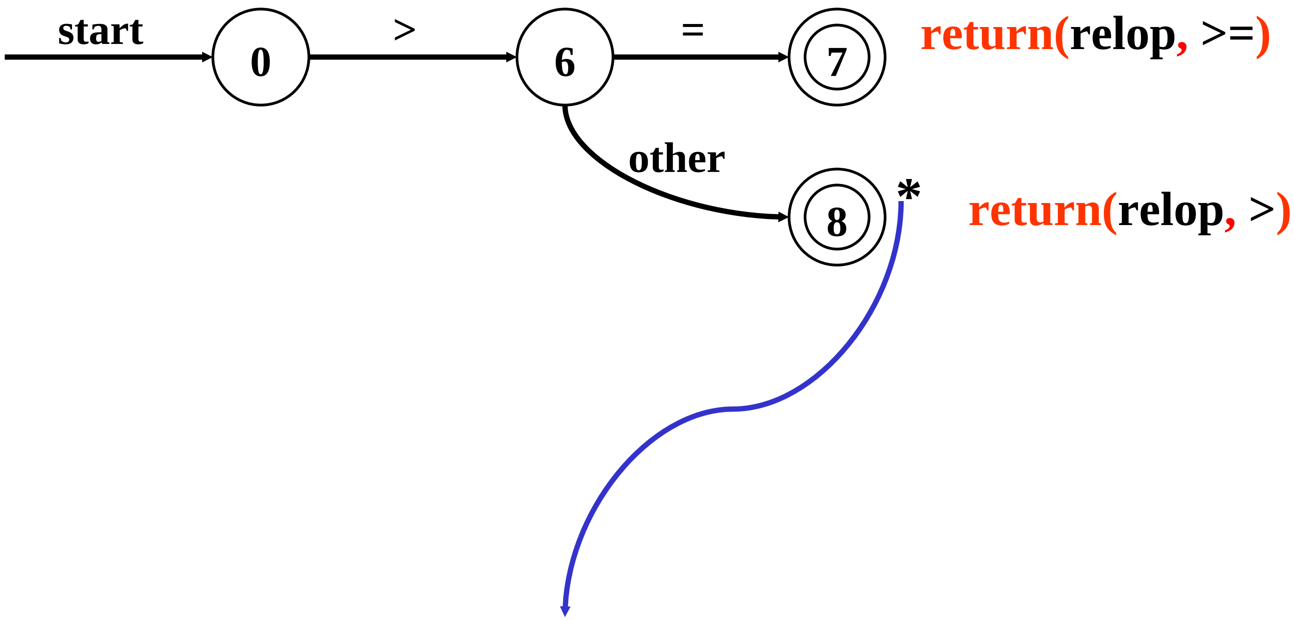
- DFA "execution"
  - If in state S_i and input a, read $T[i,a] = k$ and skip to state S_k
  - Very efficient

Table Implementation of a DFA



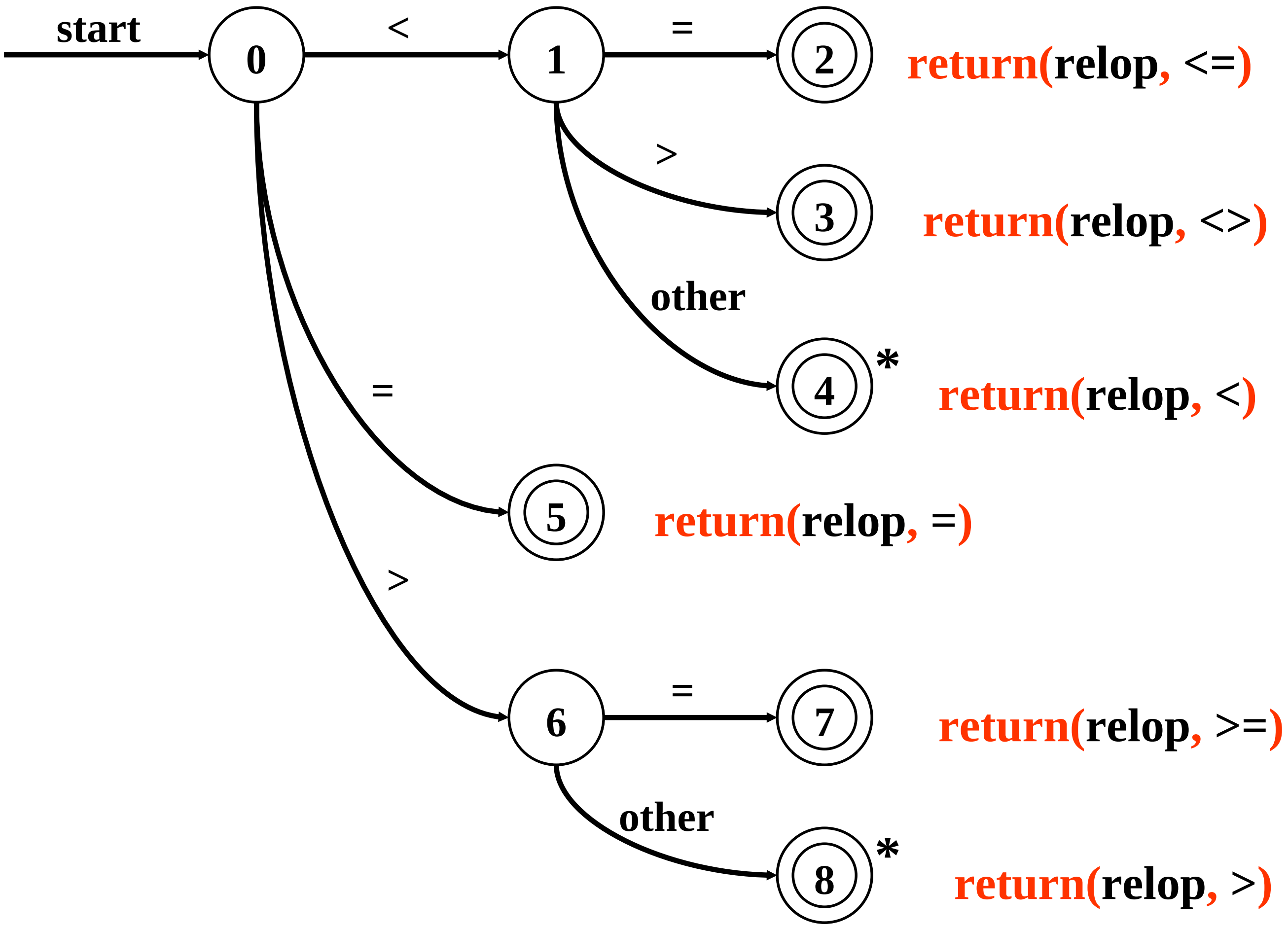
	0	1
S	T	U
T	T	U
U	T	U

DFA for recognizing two relational operators

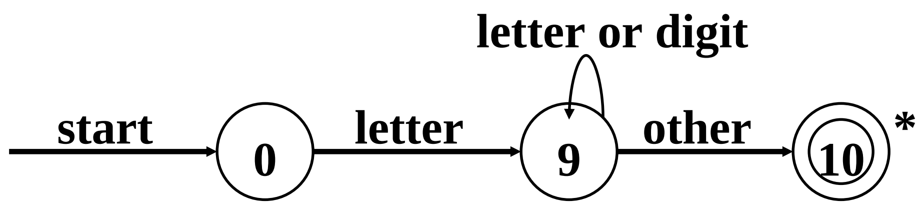


We've accepted ">" and have read "other" character that must be unread. That is moving the input pointer one character back.

DFA of Pascal relational operators



DFA for recognizing id and keyword

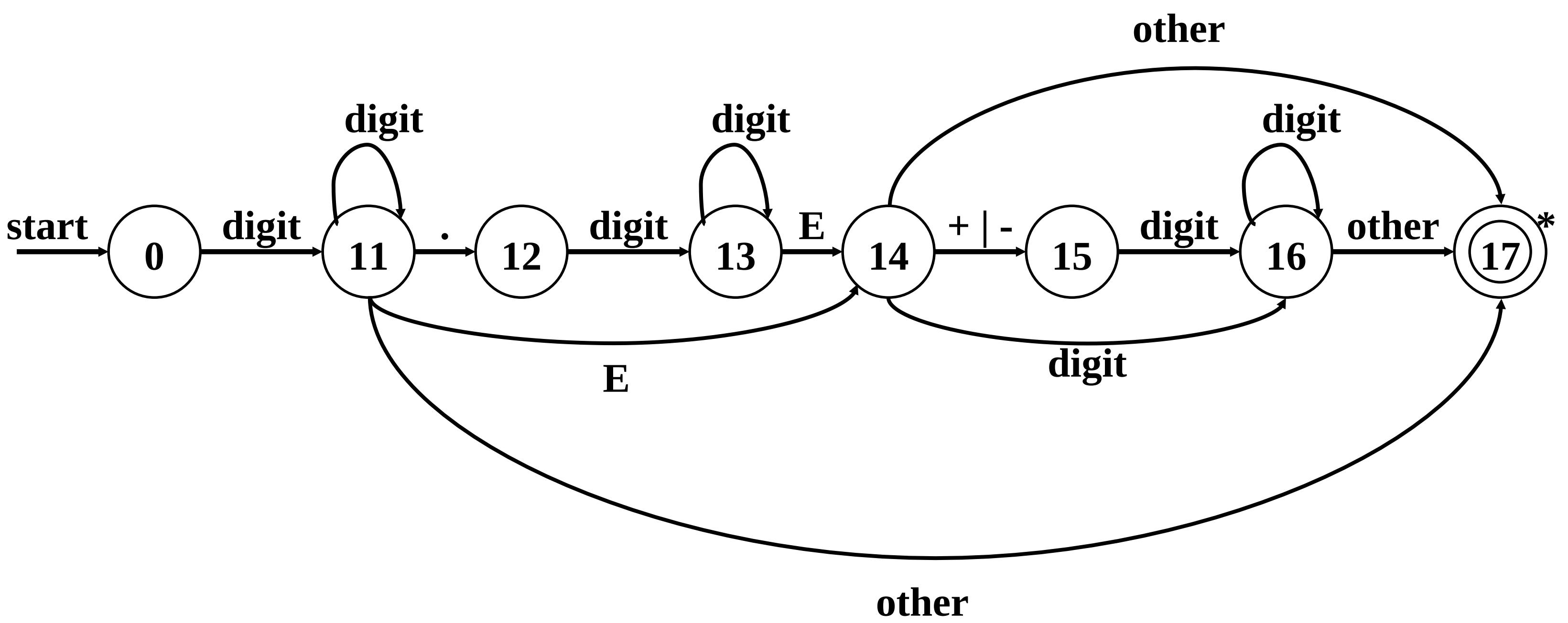


return(get_token(), install_id())

returns either a keyword or an identifier based on the type of the token

Either returns pointer to the symbol table (if token is an id) or "0" (if token is a keyword)

DFA of Pascal Unsigned Numbers



return(num, install_num())

Lexical errors

- Some errors are out of power of lexical analyzer to recognize:

  fi (a == f(x)) …

- However, it may be able to recognize errors like:

  □d = 2r

- Such errors are recognized when no pattern for tokens matches a character sequence

Error recovery

- Panic mode: successive characters are ignored until we reach to a well formed token

- Delete one character from the remaining input
- Insert a missing character into the remaining input
- Replace a character by another character
- Transpose two adjacent characters

Question?

For the code fragment below,
choose the correct number of tokens in each class that appear in the code fragment

```
x=0;\n\twhile (x>10){\n\tx++;\n}
```

- W = 9; K = 1; I = 3; N = 2; O = 9
- W = 11; K = 4; I = 0; N = 2; O = 9
- W = 9; K = 4; I = 0; N = 3; O = 9
- W = 11; K = 1; I = 3; N = 3; O = 9

W: Whitespace
K: Keyword
I: Identifier
N: Number
O: Other Tokens:
{ } ( ) < ++ ; =

Question?

How many distinct strings are in the language of the following regular expression:

$$(0 + 1 + \varepsilon)(0 + 1 + \varepsilon)(0 + 1 + \varepsilon)(0 + 1 + \varepsilon)$$

- 31
- 64
- 32
- 81

Question?

The language of the regular expression (abab)* is equivalent to the language of which of the following regular expressions?

Choose all that apply

- $(ab)^*$
- $(aba\ (baba)^*\ b) + \varepsilon$
- $(ab\ (abab)^*\ ab) + \varepsilon$
- $(a\ (ba)^*\ b) + \varepsilon$

Question?

Consider the string **abbbaacc** . Which of the following lexical specifications produces the tokenization: ab/bb/a/acc

Choose all that apply

- c*
b+
ab
ac*
- a(b + c*)
b+
- ab
b+
ac*
- b+
ab*
ac*

Question?

Using the lexical specification below, how is the string “**dictatorial**” tokenized?

Choose all that apply

dict (1)
dictator (2)
[a-z]* (3)
dictatorial (4)

- 1, 3
- 3
- 4
- 2, 3

Question?

Given the following lexical specification:

a(ba)*
b*(ab)*
abd
d+

Which of the following statements is true?

Choose all that apply

- babad will be tokenized as: bab/a/d
- ababdddd will be tokenized as: abab/dddd
- dddabbabab will be tokenized as: ddd/a/bbabab
- ababddababa will be tokenized as: ab/abd/d/ababa

Question?

Given the following lexical specification:

(00)*
01+
10+

Which strings are NOT successfully processed by this specification?

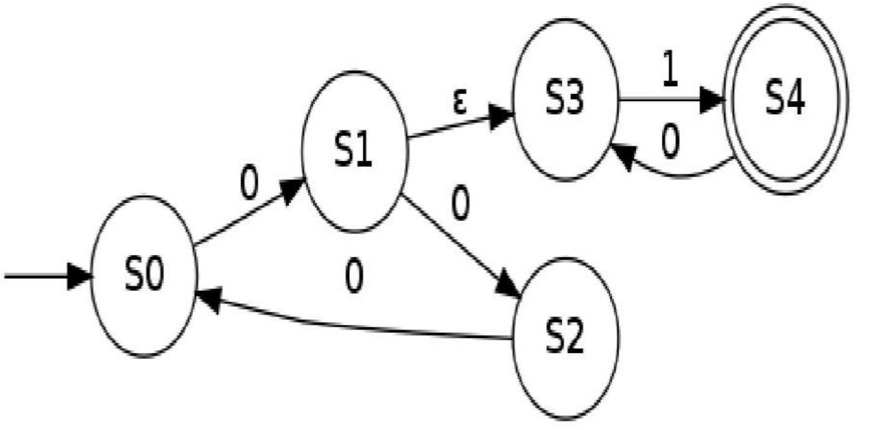
Choose all that apply

- ○ 011110
- ○ 01100100
- ○ 01100110
- ○ 0001101

Question?

Which of the following regular expressions generate the same language as the one recognized by this NFA?

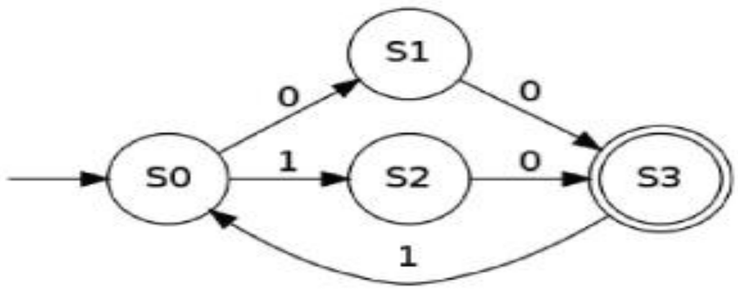
Choose all that apply

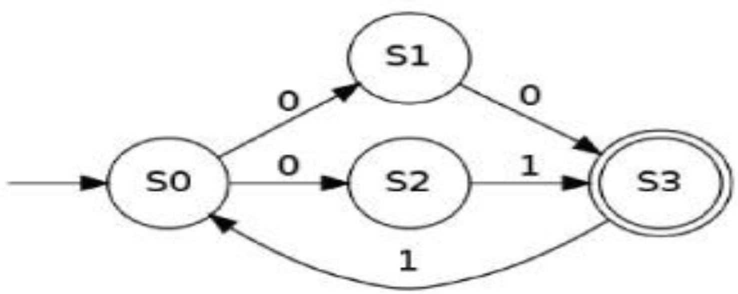


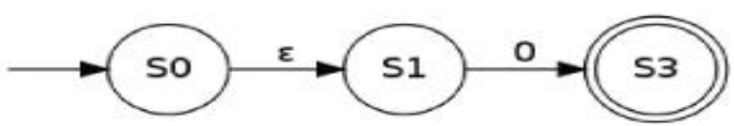
- (000)*(01)+
- 0(000)*1(01)*
- (000)*(10)+
- 0(00)*(10)*
- 0(000)*(01)*

Question?

Which of the following automata are DFA? Choose all that apply

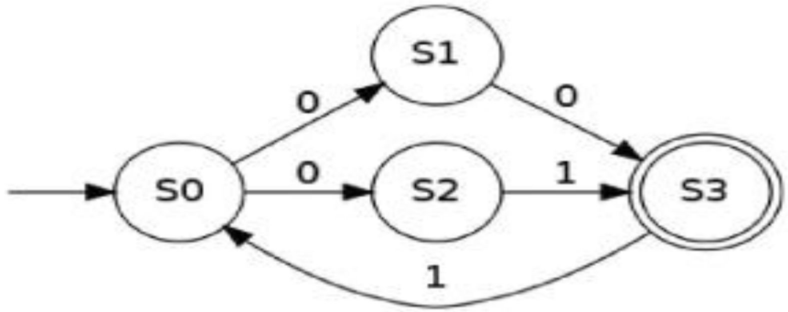


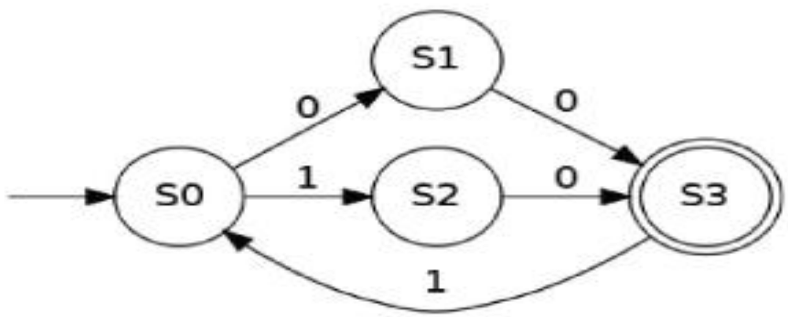


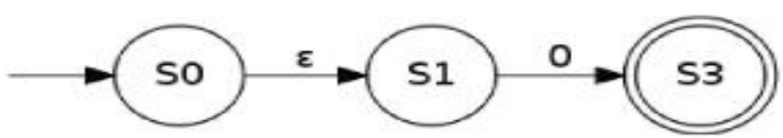


Question?

Which of the following automata are NFA? Choose all that apply







Question?

In a programming language, comments start with a '/*' and end with '*/'. Comments can contain any character strings except for '*/' unless it is surrounded by two double quote character ("). The followings are examples of valid and invalid comments:

Valid Comment	Invalid Comment
/*abc"*/"de*/	/*abc*/de*/
/**/	/*ab"*/ " */
/*21"*/ ""*/"43*/	/*21"*/"*/"43*/
/*a2"*/	/*a2"*

1-Write a regular expression that defines the set of comments in this language.

2-Draw a DFA that recognizes the set of comments in this language.