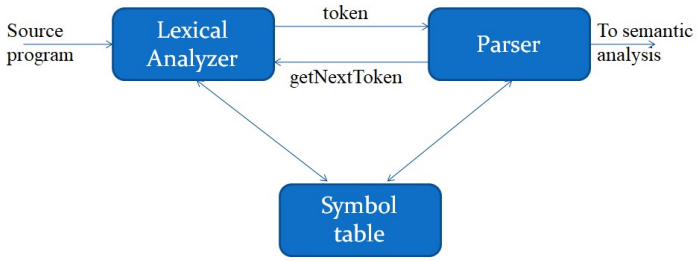
Lexical analysis

Role of Lexical Analyzer

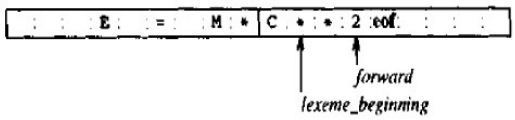
- The lexical analysis is the ***first phase*** of a compiler. Its main task is to read the input characters and produce a sequence of tokens as output that the parser uses for syntax analysis.
- A ***token*** is a logically interrelated sequence of characters, such as an identifier, a keyword, a punctuation character, or a multi-character operator like >= of the language.
- The set of strings is described by a rule called a ***pattern*** associated with the token. The pattern is said to match each string in the set.
- A ***lexeme*** is a sequence of characters in the source program that is matched by the pattern for a token.



Input Buffering

- The lexical analyzer needs to look ahead several characters beyond the lexeme for a pattern before a match can be announced.
- A large amount of time can be consumed for moving characters, so specialized buffering techniques have been developed to reduce the amount of overhead required to process an input character.
- Generally, a buffer is divided into two N-character halves. Typically, *N* is the number of characters on one disk block, e.g., 1024 or 4096.
- We read *N* input characters into each half of the buffer with one system read command, rather than invoking a read command for each input character.
- If fewer than *N* characters remain in the input, then a special character ***eof*** is read into the buffer after the input characters.

An input buffer in two halves.



Input Buffering

- Two pointers to the input buffer are maintained. The string of characters between the two pointers is the current lexeme.
- Initially, both pointers point to the first character of the next lexeme to be found. One, called the forward pointer, scans ahead until a match for a pattern is found.
- Once the next lexeme is determined, the forward pointer is set to the character at its right end.
- After the lexeme is processed, both pointers are set to the character immediately past the lexeme.
- With this scheme, comments and white space can be treated as patterns that yield no token.
- If the forward pointer is about to move past the halfway mark, the right half is filled with *N* new input characters.
- If the forward pointer is about to move past the right end of the buffer, the left half is filled with N new characters and the forward pointer wraps around to the beginning of the buffer.

Specification of Token

- ***Regular expressions*** are an important notation for specifying patterns.
- Each ***pattern*** matches a set of strings, so regular expressions will serve as names for sets of strings.
- A ***string*** over some alphabet is a finite sequence of symbols drawn from that alphabet.
- The term ***language*** denotes any set of strings over some fixed alphabet.

Specification of Token

Regular Expressions: A formal recursive definition of regular expressions over Σ as follows:

1. Any terminal symbol (i.e. an element of Σ), ^, and Ø are regular expressions.
2. The union of two regular expressions R_1 and R_2, written as R_1+R_2, is also a regular expression.
3. The concatenation of two regular expressions R_1 and R_2, written as R_1R_2, is also a regular expression.
4. The iteration (or closure of) a regular expression R, written as R*, is also a regular expression.
5. If R is a regular expression, then (R) is also a regular expression.
6. The regular expressions over Σ are precisely those obtained recursively by the application of the rules 1-5 once or several times.

Recognition of Token

Consider the following grammar fragment:

stmt -> **if** expr **then** stmt | **if** expr **then** stmt **else** stmt | Ɛ

expr -> term **relop** term | term

term -> **id** | **num**

where the terminals ***if***, ***then***, ***else***, ***relop***, ***id***, and ***num*** generate sets of strings given by the following regular definitions

if -> **if**

then -> **then**

else -> **else**

relop -> **<** | **>** | **<=** | **>=** | **=** | **<>**

id -> letter (letter | digit)*

num -> digit+(.digit+)? (E (+ | -)? digit+)?

letter -> *a* | b | ….. | z | A | B | …. | Z

digit -> 0 | 1 | ….. | 9

Where **?** means zero or one times.

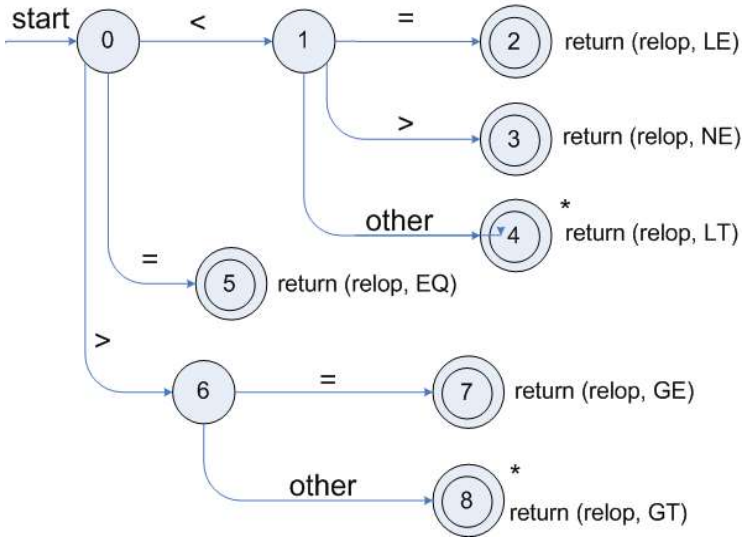
Recognition of Token

Transition Diagram

- As an intermediate step in the construction of a lexical analyzer, a stylized flowchart is produced, which is called ***transition diagram***.
- Positions in a transition diagram are drawn as ***circles*** and are called ***states***.
- The states are connected by arrows, called ***edges***.
- Edges leaving states have labels indicating the input characters that can next appear after the transition diagram has reached state s.
- One state is labeled the ***start state***; it is the initial state of the transition diagram where control resides when we begin to recognize a token.
- If there is an edge from the current state whose label matches this input character, we then go to the state pointed to by the edge. Otherwise, we indicate failure.
- The ***double circle*** indicates that it is an accepting state.

Recognition of Token

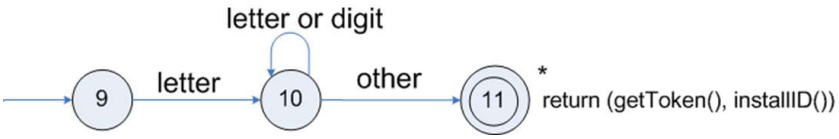
Transition diagram for relational operator



Recognition of Token

Transition diagram for identifiers and keywords

id -> letter (letter | digit)*

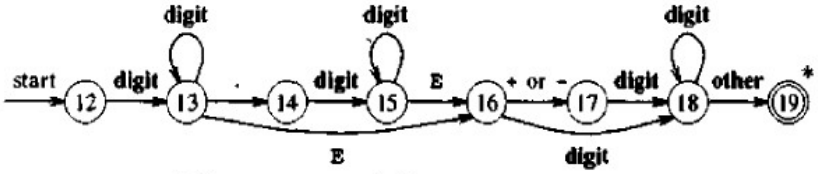


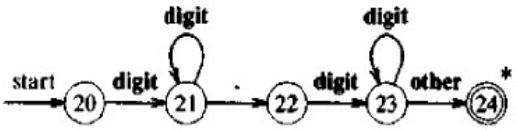
Draw a Transition diagram for identifier in C language
(An identifier starts with a letter A to Z, a to z, or an underscore '_' followed by zero or more letters, underscores, and digits (0 to 9).

Recognition of Token

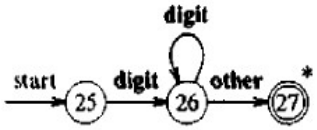
Transition diagram for unsigned numbers

num -> digit^{+}(.digit^{+})$^{?}$ E (+ | -)$^{?}$ digit^{+}





num -> digit^{+}.digit^{+}



num -> digit^{+}

Deterministic Finite Automation (DFA)

A deterministic finite automaton can be represented by a 5-tuple

$(Q, \sum, \delta, q_0, F)$, where

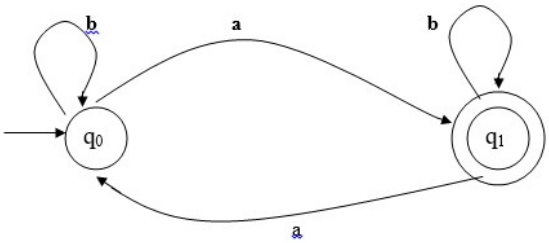
i. Q is a finite nonempty set of states;

ii. $\sum$ is a finite nonempty set of inputs called input alphabet;

iii. δ is a function which maps Q x $\sum$ into Q and is usually called direct transition function.

iv. $q_0 \in Q$ is the initial state; and

v. $F \subseteq Q$ is the set of final states. It is assumed that there may be more than one final state.



Nondeterministic Finite Automation (NDFA)

A nondeterministic finite automaton can be represented by a 5-tuple $(Q, \sum, \delta, q_0, F)$, where

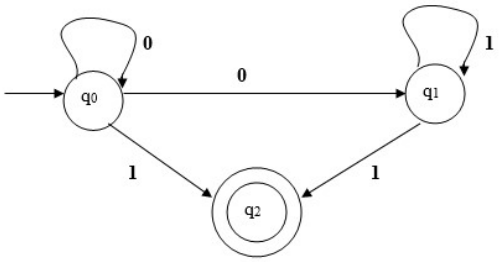
I. Q is a finite nonempty set of states;

II. $\sum$ is a finite nonempty set of inputs;

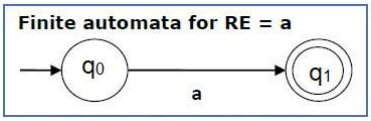
III. δ is the transition function mapping from Q x $\sum$ into 2^Q which is the power set of Q, the set of all subsets of Q;

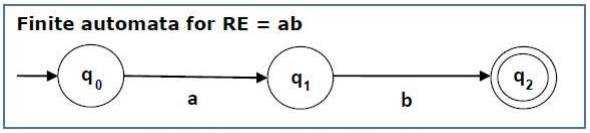
IV. $q_0 \in Q$ is the initial state; and

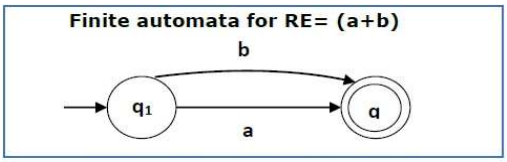
V. $F \subseteq Q$ is the set of final states.

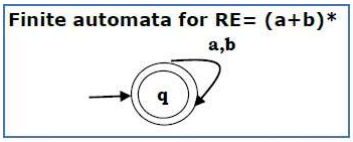


Regular Expression to NDFA



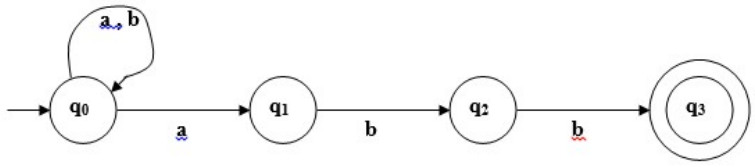




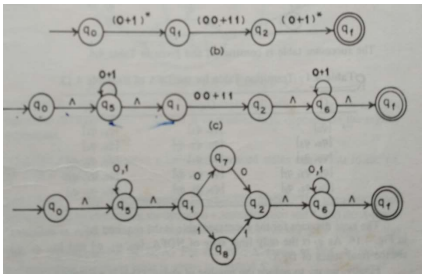


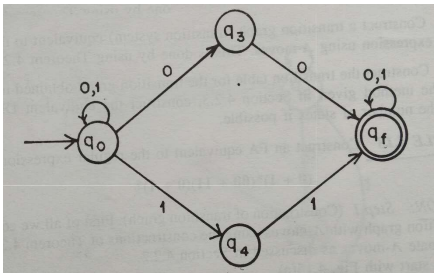
Regular Expression to NDFA

Construct an NFA for the regular expression R= (a / b)*abb



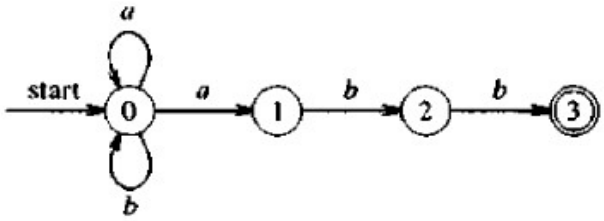
Construct an NFA equivalent to the regular expression R=(0+1)* (00+11) (0+1)*





NDFA to DFA Conversion

A nondeterministic finite automaton for $(a|b)^*abb$

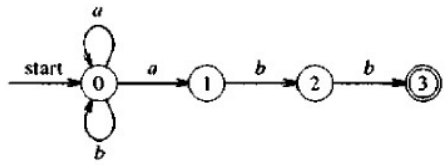


States/Σ	a	b
→ q_0	q_0,q_1	q_0
q_1		q_2
q_2		q_3
(q_3)		

States/Σ	a	b
→ q_0	q_0,q_1	q_0
q_0q_1	q_0,q_1	q_0,q_2
q_0q_2	q_0,q_1	q_0,q_3
(q_0q_3)	q_0,q_1	q_0

NDFA to DFA Conversion

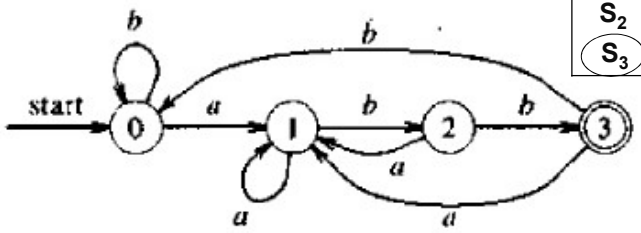
A nondeterministic finite automaton for $(a|b)^*abb$



States/Σ	a	b
→ q_0	q_0,q_1	q_0
q_0q_1	q_0,q_1	q_0,q_2
q_0q_2	q_0,q_1	q_0,q_3
q_0q_3	q_0,q_1	q_0

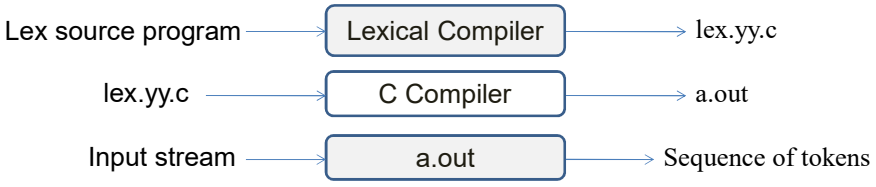
States/Σ	a	b
→ S_0	S_1	S_0
S_1	S_1	S_2
S_2	S_1	S_3
S_3	S_1	S_0

DFA accepting $(a|b)^*abb$.



Lex Compiler

- Lex compiler or ***Lex*** is a tool that is used to specify lexical analyzers for a variety of languages. It comes as an integrated utility of the UNIX operating system.
- The tool accepts a specification of the tokens in terms of extended regular expressions along with supporting routines.
- It produces a C-program (***lex.yy.c***) as an output, that can be compiled to obtain an executable version of the lexical analyzer.
- The lex.yy.c is compiled using a C compiler to produce the lexical analyzer (***a.out***).
- This lexical analyzer can now be fed with an input stream to determine the tokens in it.



Lex Specification

- A Lex program consists of three parts:

```
declarations
%%
translation rules
%%
auxiliary functions
```

- The ***declarations sections*** includes variables declarations, constant declarations, and regular definitions.
- The ***transaction rules*** sections contains the specification of tokens and the associated action. This section is of the form:

  r_1 {$action_1$}
  r_2 {$action_2$}
  …
  r_n {$action_n$}

  Here, each r_i is a regular expression.
- Each action specifies a set of statements to be executed whenever rule r_i matches the current input sequence.
- The ***auxiliary functions*** are the functions that may be used to write the action parts.

THANK YOU