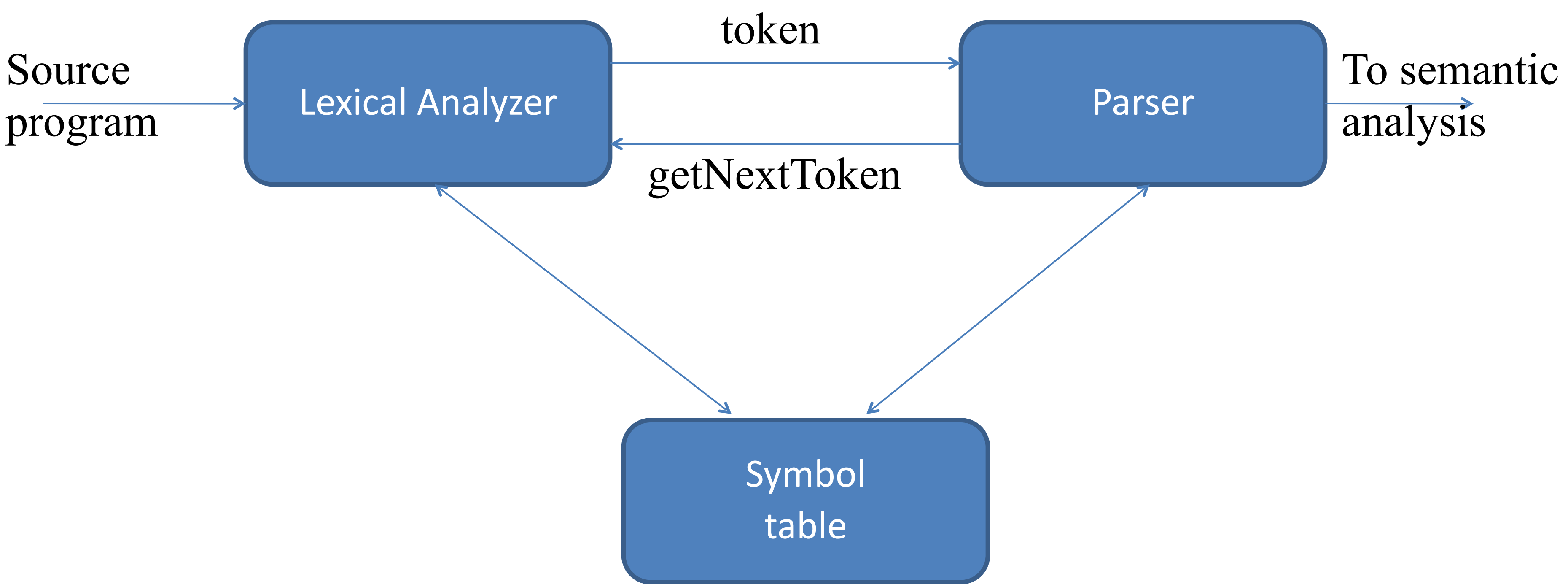
19Z602 COMPILER DESIGN
Unit-2

LEXICAL ANALYSIS : Need and Role of Lexical Analyzer - Input Buffering - Lexical Errors - Expressing Tokens by Regular Expression - Finite Automata: NFA- DFA - Converting NFA to DFA - Minimization of DFA- Converting Regular Expression to DFA. LEX Tool: Structure of LEX Program – Predefined Variables – Library routines – Design of Lexical Analyzer for a Sample Language

Outline

- Role of lexical analyzer
- Specification of tokens
- Recognition of tokens
- Lexical analyzer generator
- Finite automata
- Design of lexical analyzer generator

The role of lexical analyzer



Why to separate Lexical analysis and parsing

1. Simplicity of design

A parser embodying the conventions for comments and white space is **significantly more complex than one that can assume comments and white space have already been removed by a lexical analyzer.**

2. Improving compiler efficiency

A separate lexical analyzer allows us to construct a specialized techniques that serve only the lexical task, not the job of parsing.

Specialized buffering techniques for reading input characters can speed up the compiler significantly.

3. Enhancing compiler portability

Input device specific peculiarities can be restricted to the lexical analyzer.

Tokens, Patterns and Lexemes

- A token is a pair of token name and an optional token value. The token name is an **abstract symbol representing a kind of lexical unit.**
- A **pattern** is a **description** of the form that the lexemes of a token may take.
- A **lexeme** is a sequence of characters in the source program that matches the pattern for a token

Example

Token	Informal description	Sample lexemes
if	Characters i, f	if
else	Characters e, l, s, e	else
comparison	< or > or <= or >= or == or !=	<=, !=
id	Letter followed by letter and digits	pi, score, D2
number	Any numeric constant	3.14159, 0, 6.02e23
literal	Anything but “ sorrounded by “	“core dumped”

printf(“total = %d\n”, score);

Attributes for tokens

- E = M * C ** 2
  - <id, pointer to symbol table entry for E>
  - <assign-op>
  - <id, pointer to symbol table entry for M>
  - <mult-op>
  - <id, pointer to symbol table entry for C>
  - <exp-op>
  - <number, integer value 2>

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

Input buffering

- Sometimes lexical analyzer needs to look ahead some symbols to decide about the token to return
  - In C language: need to look after -, = or < to decide what token to return
  - In Fortran: DO 5 I = 1.25
- need to introduce a two buffer scheme to handle large look-aheads safely

Sentinels


```
Switch (*forward++) {
    case eof:
        if (forward is at end of first buffer) {
                reload second buffer;
                forward = beginning of second buffer;
        }
        else if {forward is at end of second buffer) {
                reload first buffer;
                forward = beginning of first buffer;
        }
        else /* eof within a buffer marks the end of input */
                terminate lexical analysis;
        break;
    cases for the other characters;
}
```

Specification of tokens

- In theory of compilation regular expressions are used to formalize the specification of tokens
- Regular expressions are means for specifying regular languages
- Example:
    - Letter_(letter_ | digit)*
- Each regular expression is a pattern specifying the form of strings

Recognition of tokens

- Starting point is the language grammar to understand the tokens:

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

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

term -> **id**
    | **number**

Recognition of tokens (cont.)

- The next step is to formalize the patterns:

Regular Expression for numerical constant

digit -> [0-9]

digits -> digit+

fraction -> (.digits)?

exponent -> (E(+|-)? Digits)?

number -> digits fraction exponent

Regular Expression for conditional branching statement (if-then-else)

letter -> [A-Za-z_]

digit -> [0-9]

digits -> digit+

id -> letter (letter|digit)*

if -> if

then -> then

else -> else

Relop -> < | > | <= | >= | = | <>

delim -> blank | \t | \n

ws -> delim+

term -> id | number

expr -> term relop term | term

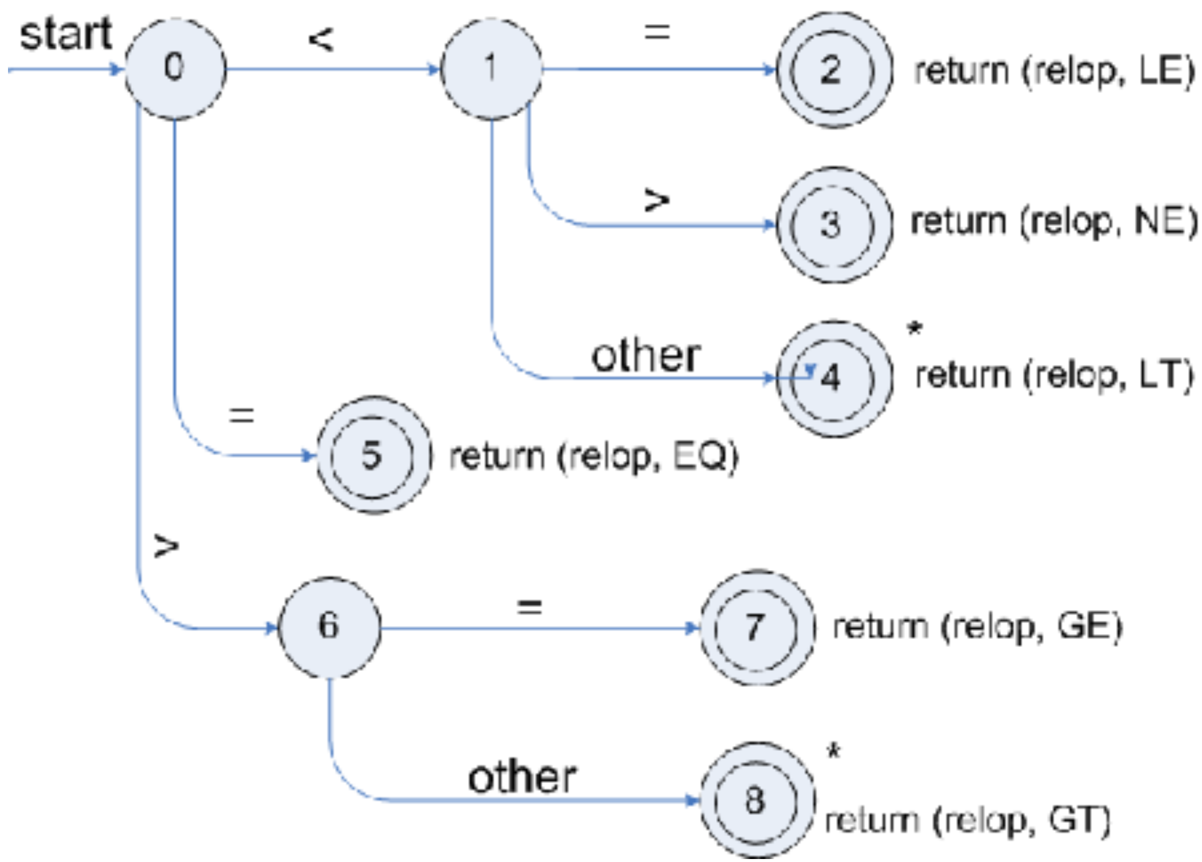
stmt -> if expr then stmt

| *if expr then stmt else stmt*

| ε

Transition diagrams

- Transition diagram for relop

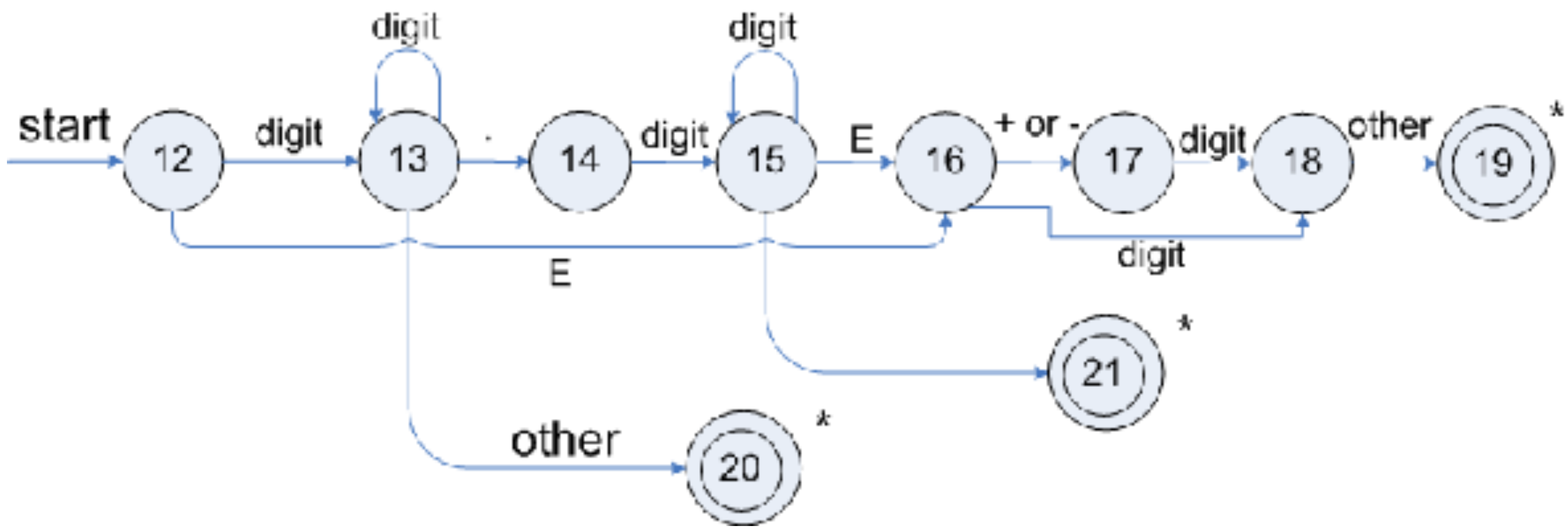


Transition diagrams (cont.)

- Transition diagram for reserved words and identifiers

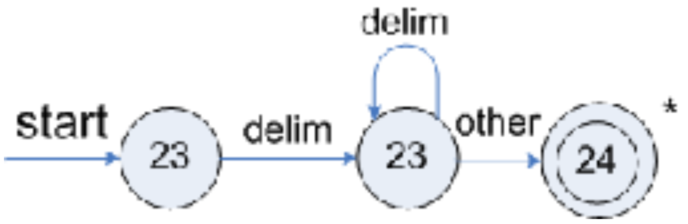
Transition diagrams (cont.)

- Transition diagram for unsigned numbers



Transition diagrams (cont.)

- Transition diagram for whitespace



Finite Automata

- Regular expressions = specification
- Finite automata = implementation
- Finite automata are ***recognizers*** , says ‘yes’ or ‘no’ about each string
- 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 $\xrightarrow{\text{input}}$ state

Finite Automata

- Transition

$$s_1 \xrightarrow{a} s_2$$

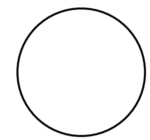
- Is read

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

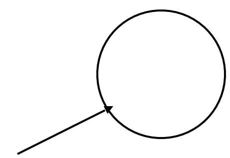
- If end of input
  - If in accepting state => accept, othewise => reject
- If no transition possible => reject

Finite Automata State Graphs

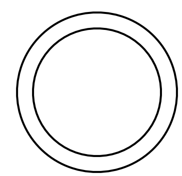
- A state



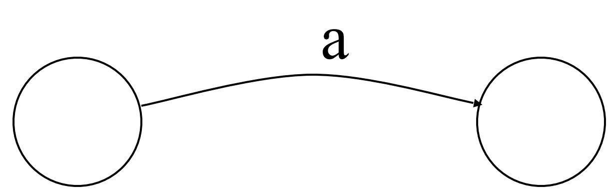
- The start state



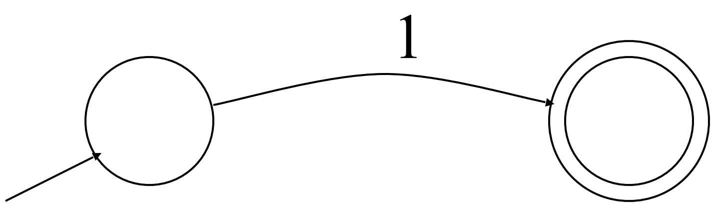
- An accepting state



- A transition

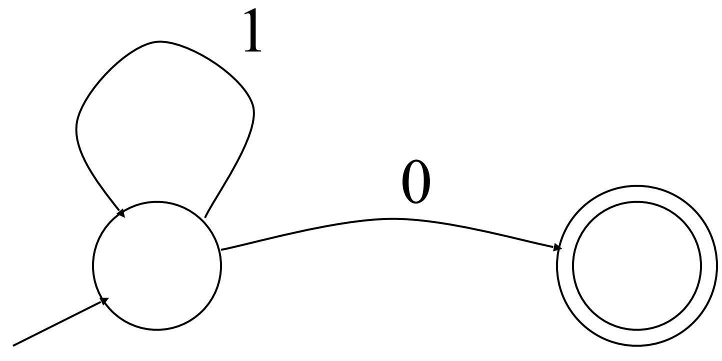


A Simple Example



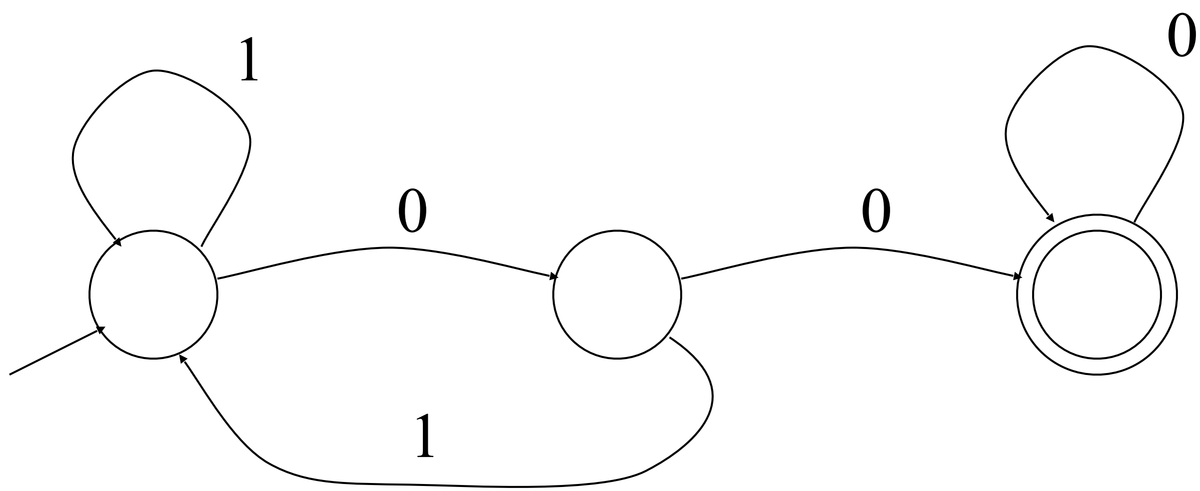
Another Simple Example

- A finite automaton accepting any number of 1's followed by a single 0
- Alphabet: {0,1}
- Check that "1110" is accepted but "110…" is not



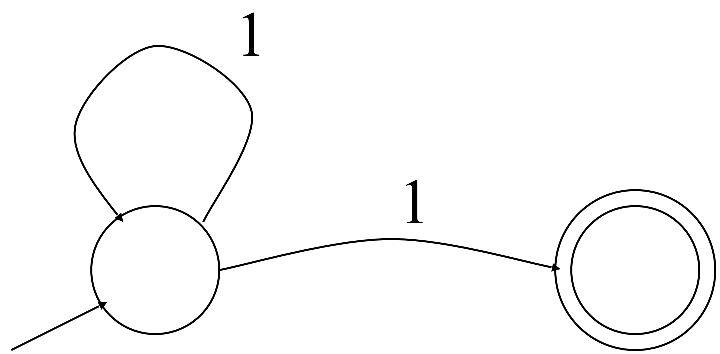
And Another Example

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



And Another Example

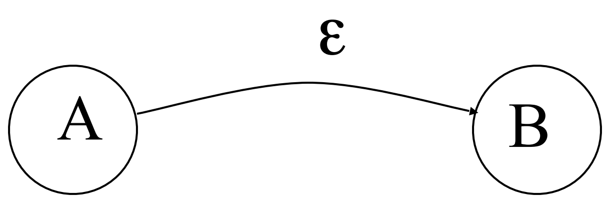
- Alphabet still { 0, 1 }



- The operation of the automaton is not completely defined by the input
  - On input “11” the automaton could be in either state

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
- *Finite* automata have *finite* memory
  - Need only to encode the current state

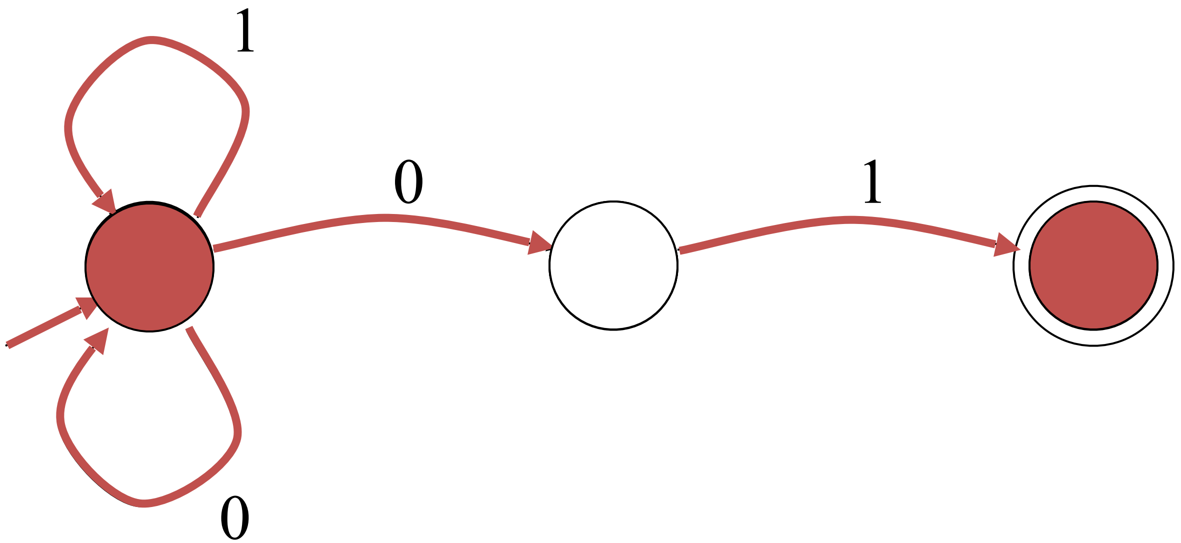
Execution of Finite Automata

- 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 1
- Rule: NFA accepts if it <u>can</u> get in a final state

NFA vs. DFA (1)

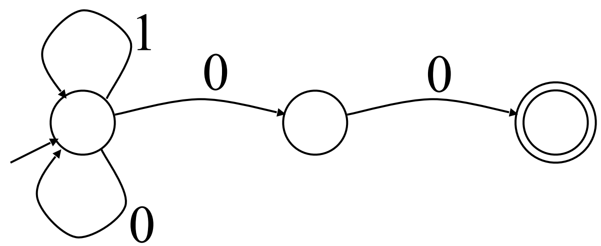
- NFAs and DFAs recognize the same set of languages (regular languages)

- DFAs are easier to implement
  - There are no choices to consider

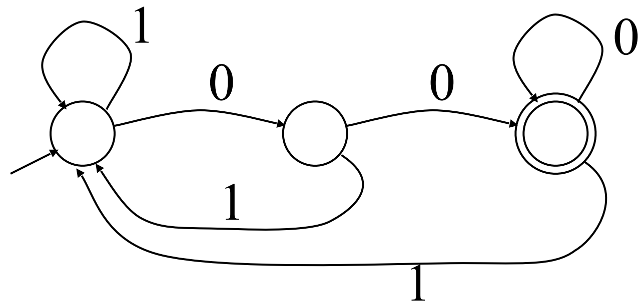
NFA vs. DFA (2)

- For a given language the NFA can be simpler than the DFA

NFA



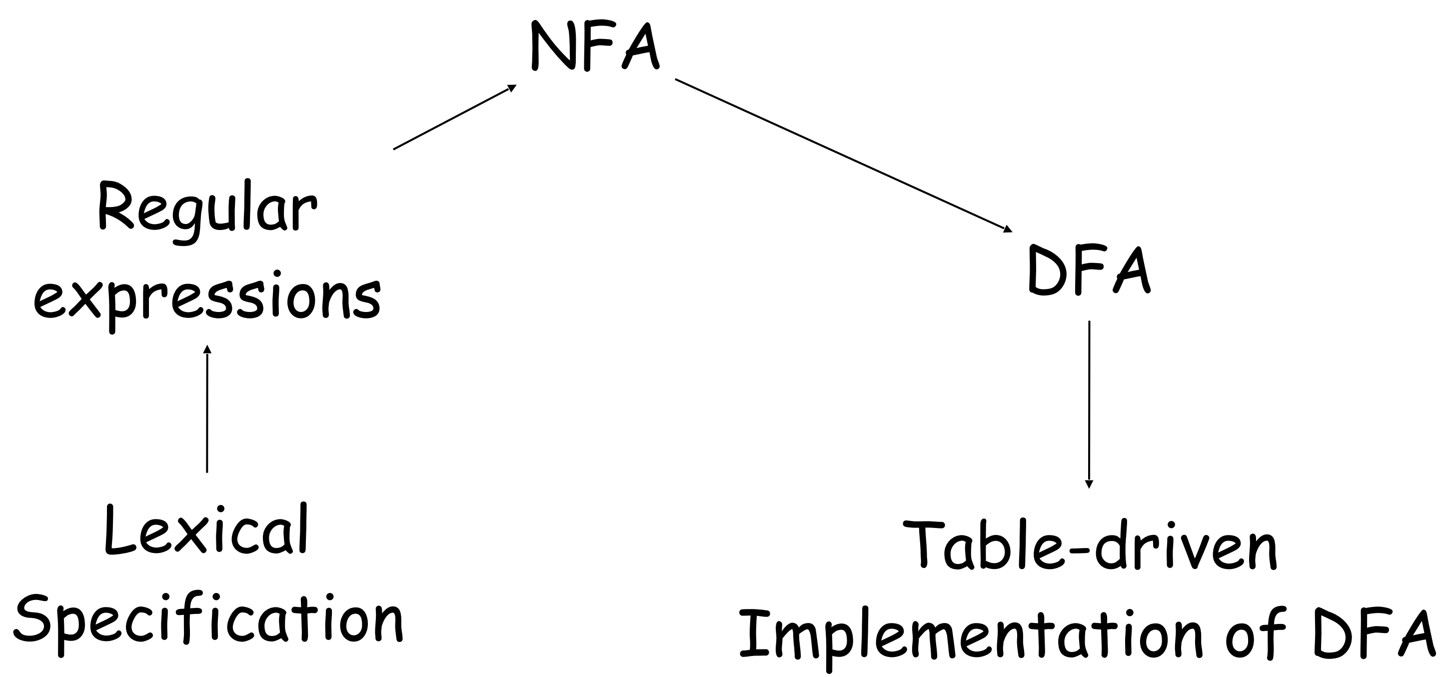
DFA



- DFA can be exponentially larger than NFA

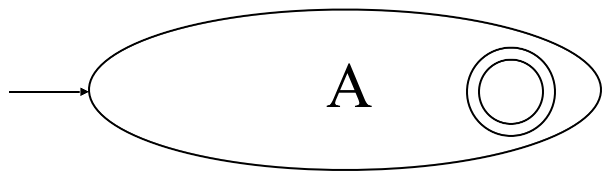
Regular Expressions to Finite Automata

- High-level sketch

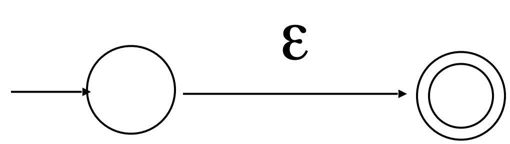


Regular Expressions to NFA (1)

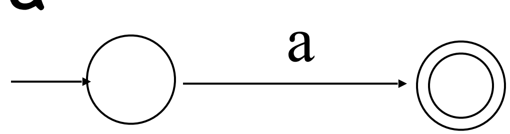
– **McNaughton-Yamada-Thompson Algorithm to convert Regular Expression to an NFA**
– Notation: NFA for RE A



- For ε

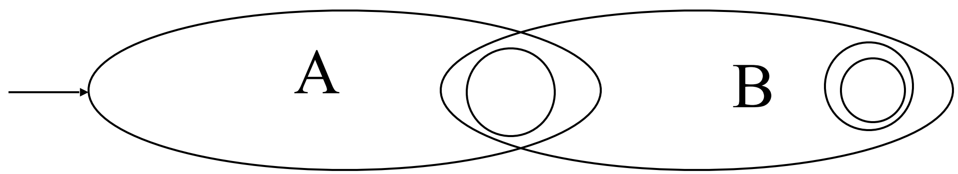


- For input a

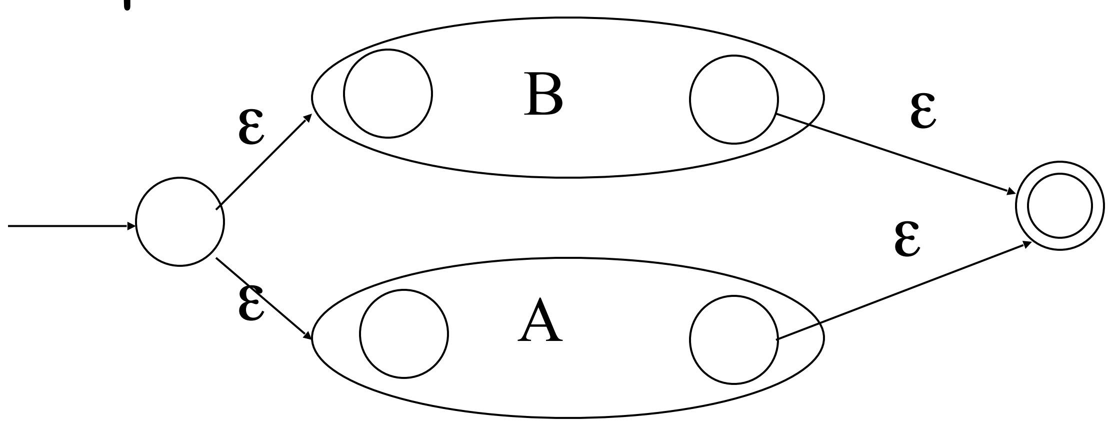


Regular Expressions to NFA (2)

- For AB

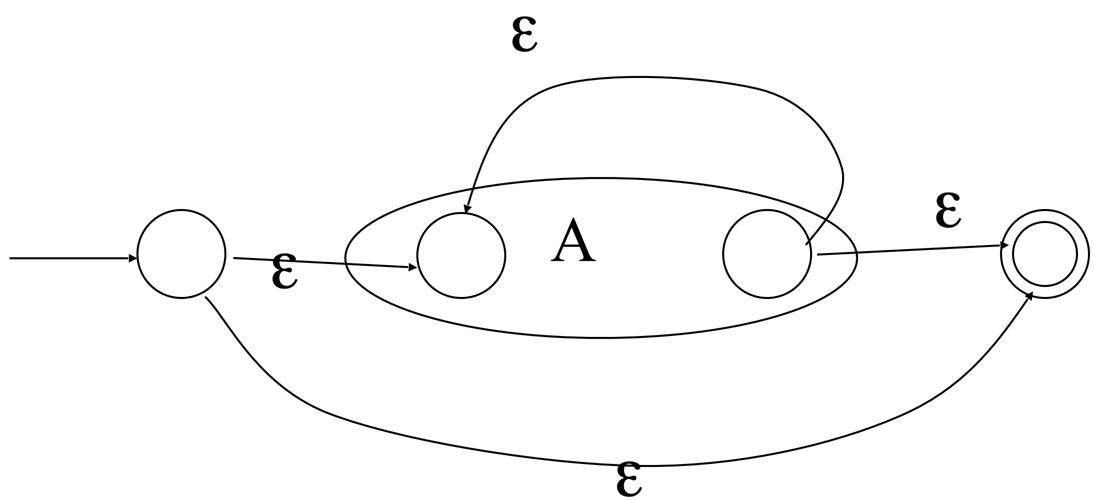


- For A | B



Regular Expressions to NFA (3)

- For A*

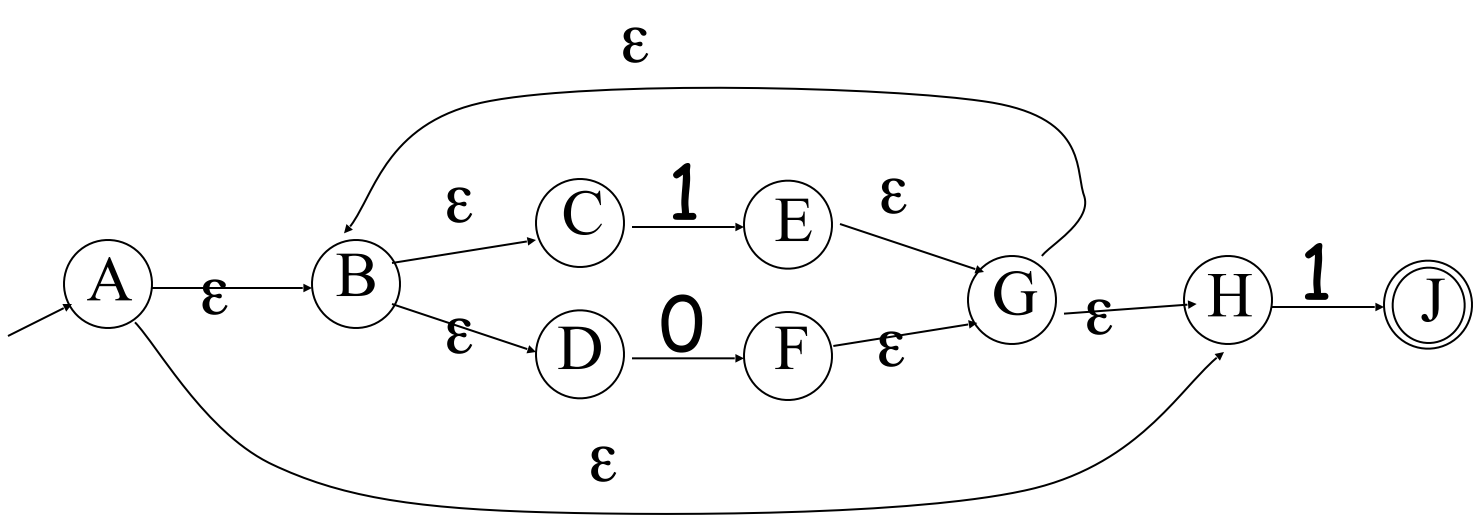


Example of RegExp -> NFA conversion

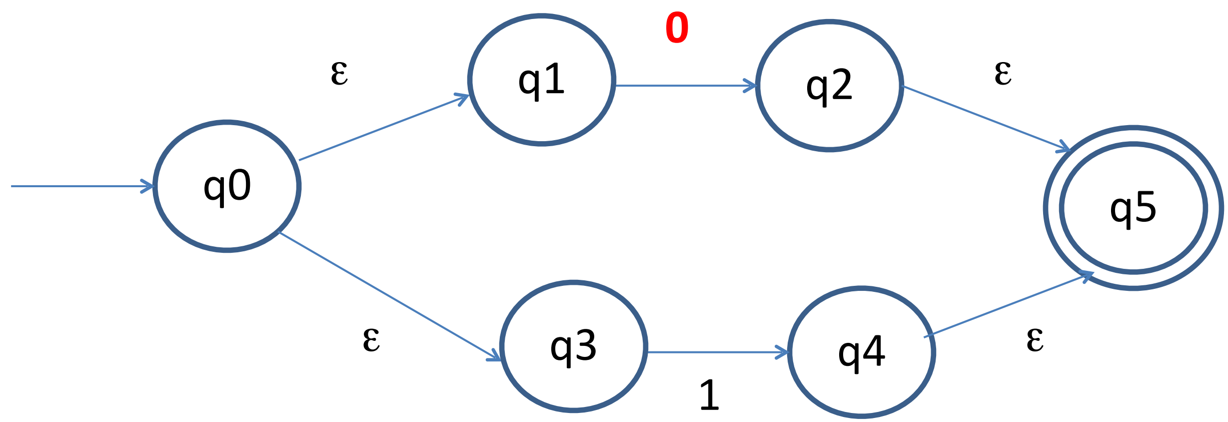
- Consider the regular expression

  (1 | 0)*1

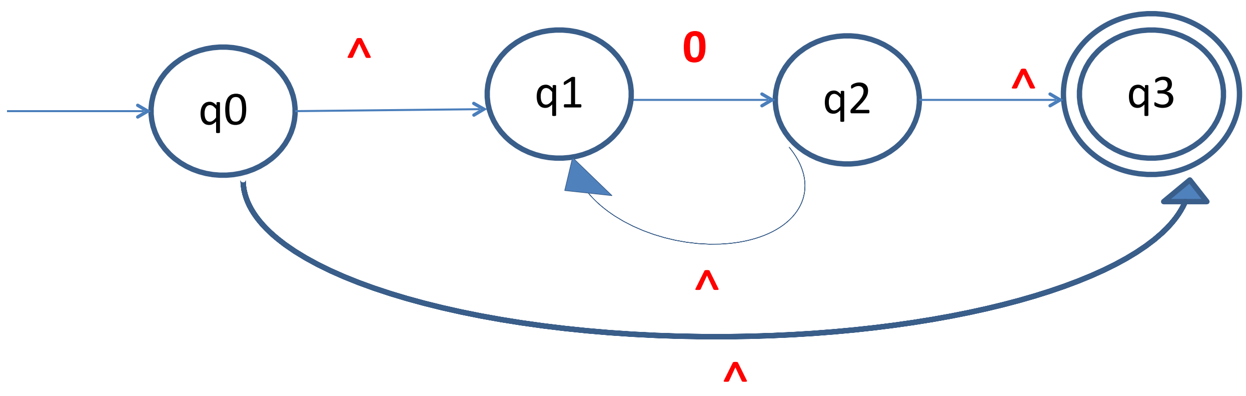
- The NFA is



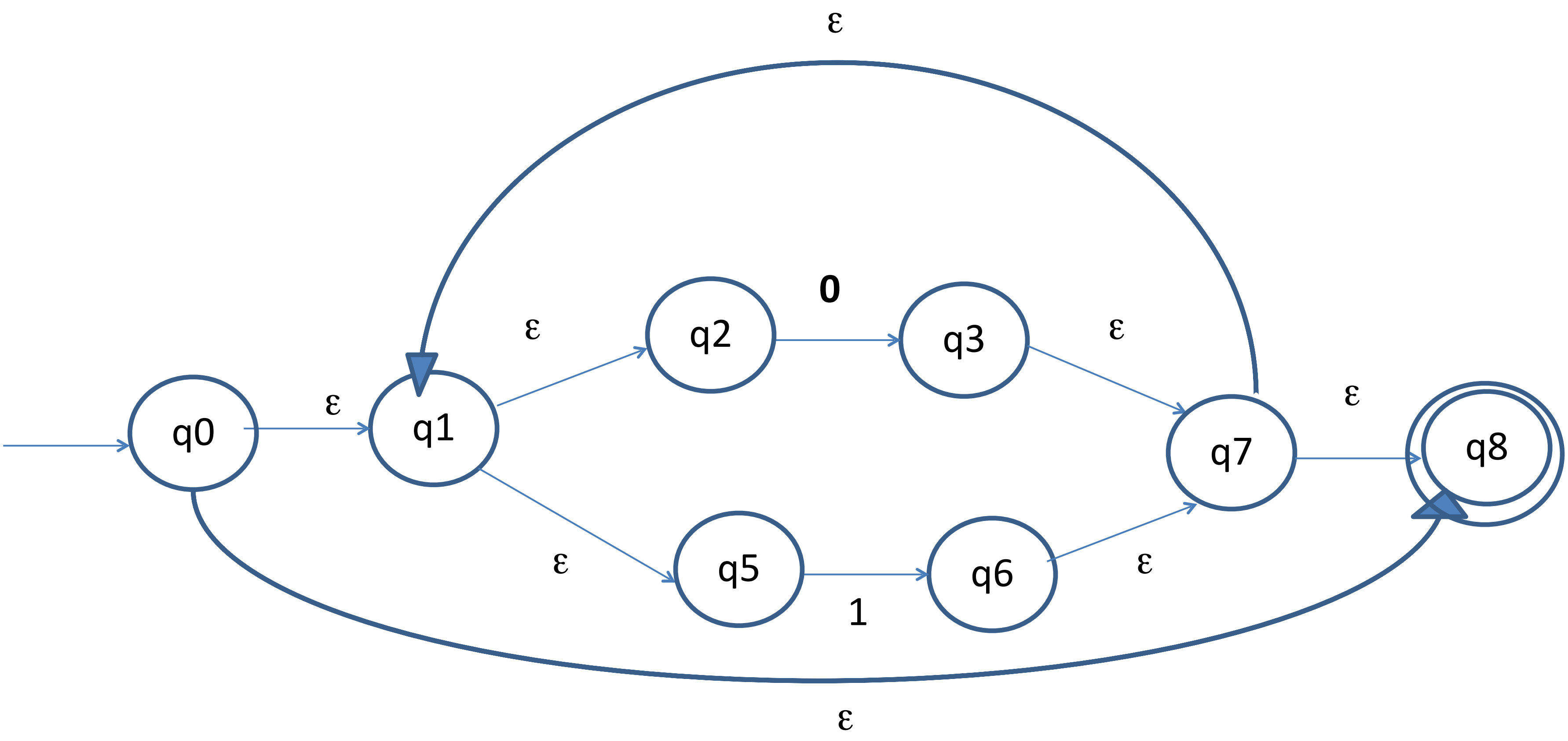
0 + 1



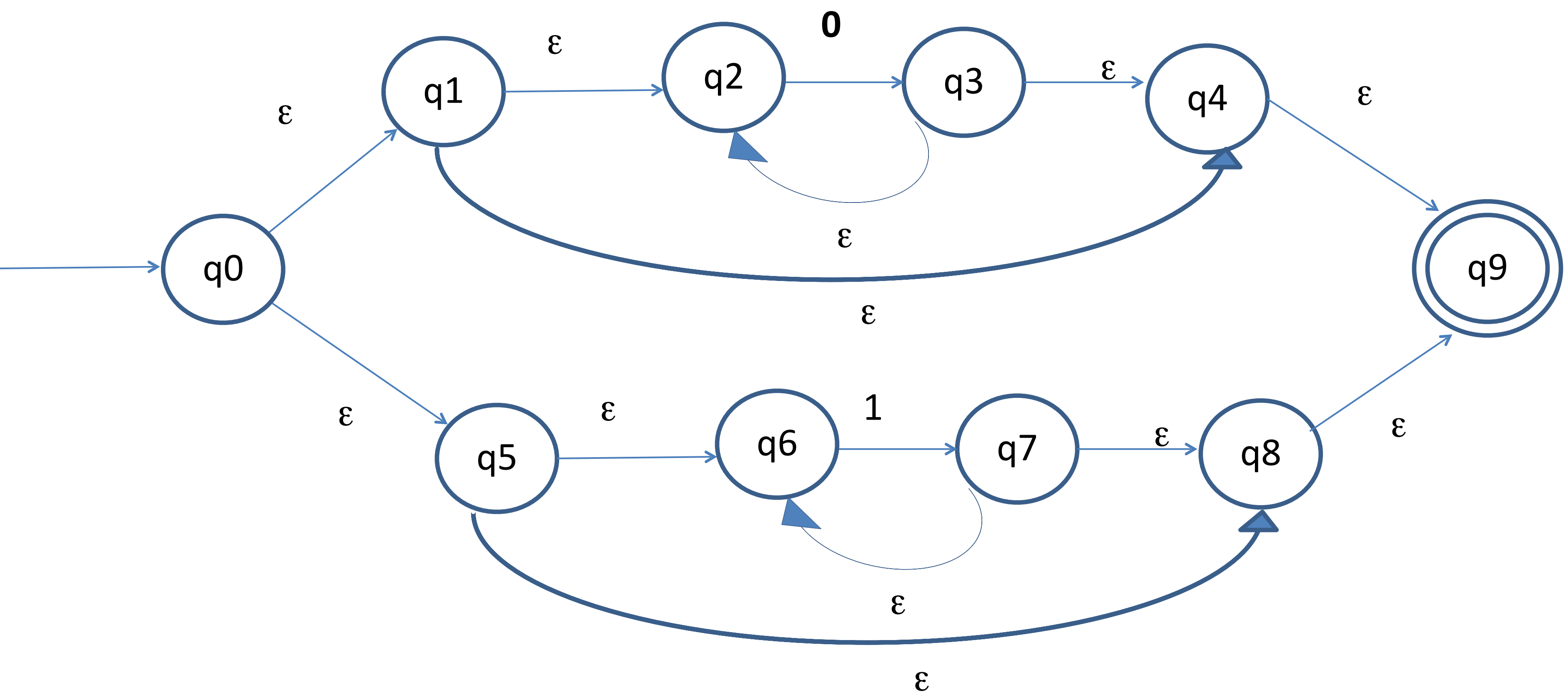
0*



(0 + 1) *

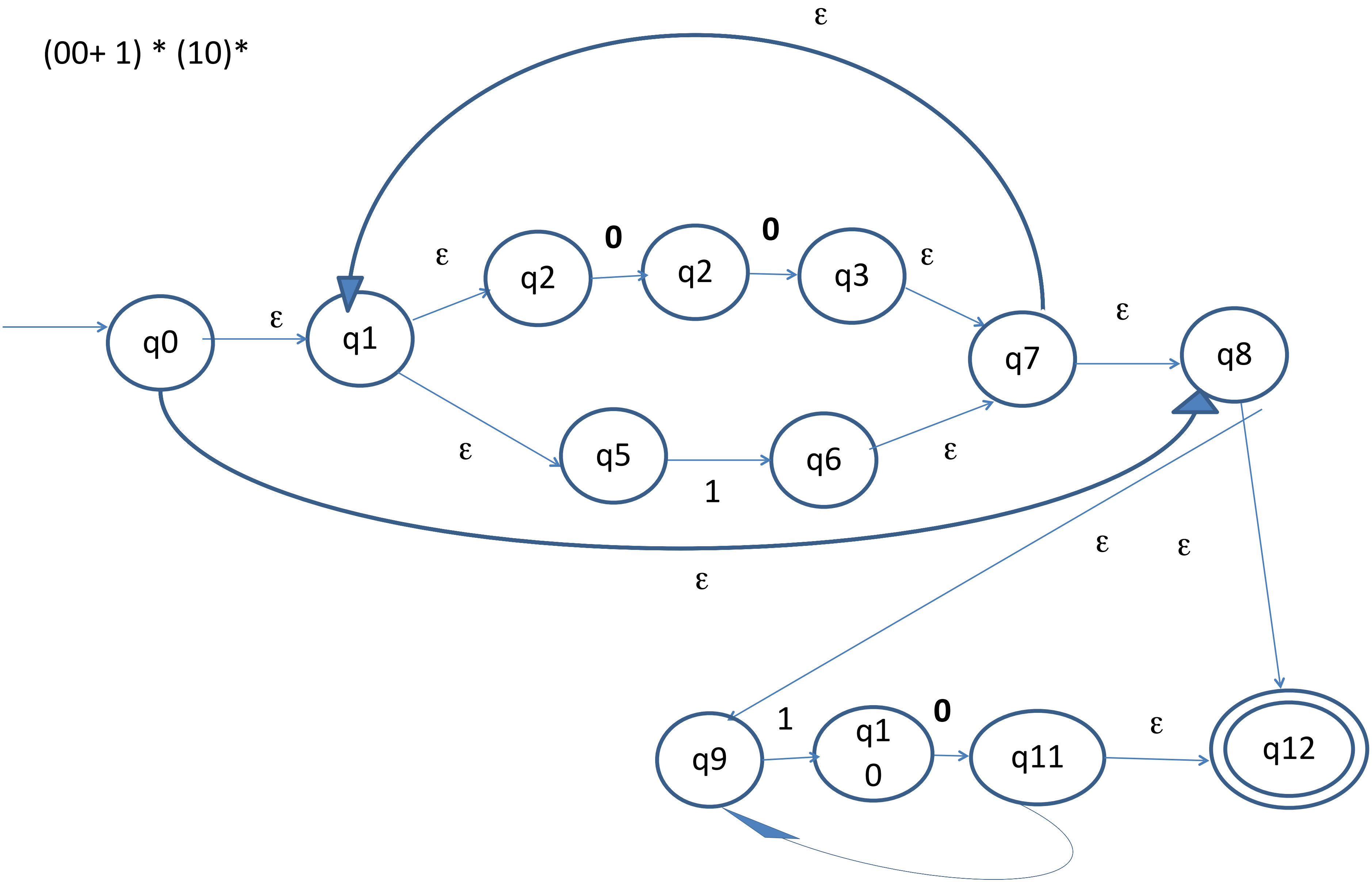


0 *+ 1 *



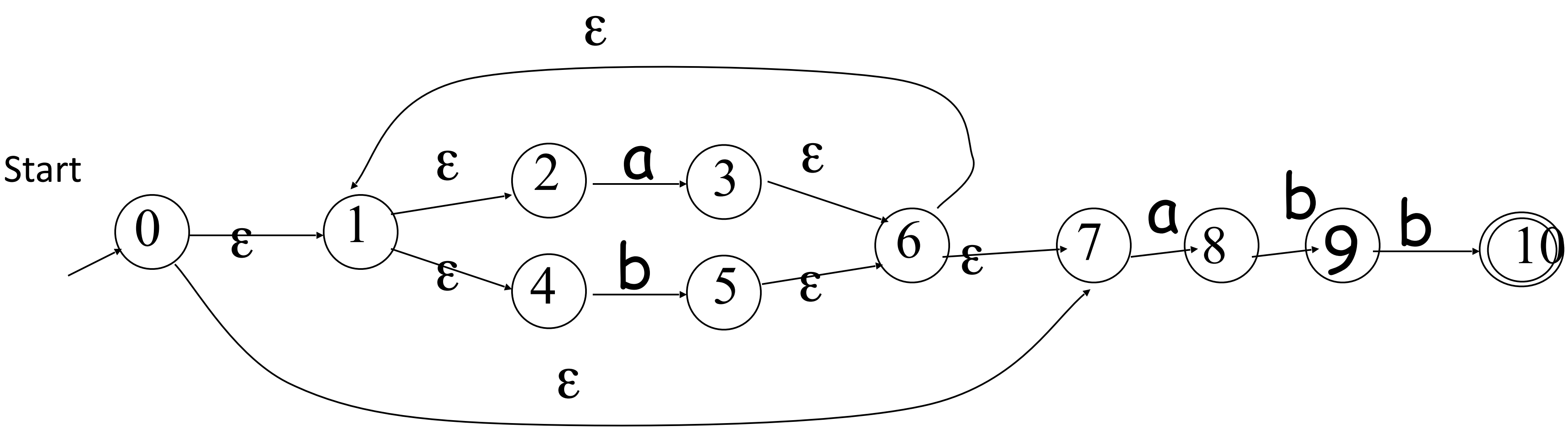
(00+ 1) * (10)*

(00+ 1) * (10)*



(a|b)*abb

NFA for (a|b)*abb



The subset Construction

initially, ϵ-*closure*(s_0) is the only state in *Dstates*, and it is unmarked;
while (there is an unmarked state T in *Dstates*) {
    mark T;
    for (each input symbol a) {
        $U = \epsilon\text{-}closure(move(T, a))$;
        if (U is not in *Dstates*)
            add U as an unmarked state to *Dstates*;
        $Dtran[T, a] = U$;
    }
}

ε-closure(s)- Set of NFA states reachable from NFA state **s** on ε-transitions alone
ε-closure(T) - Set of NFA states reachable from some NFA state **s** in set T on ε-transitions alone
εmove(T,a)- Set of NFA states to which there is a transition on input symbol **a** from some state s in T

Computing ε-closure(T)

```
push all states of T onto stack;
initialize ϵ-closure(T) to T;
while ( stack is not empty ) {
        pop t, the top element, off stack;
        for ( each state u with an edge from t to u labeled ϵ )
                if ( u is not in ϵ-closure(T) ) {
                        add u to ϵ-closure(T);
                        push u onto stack;
                }
}
```

ε-closure(0)= {0,1,2,4,7}=A
Dtran[A,a]=ε-closure(move(A,a)) – ε-closure({3,8})= {1,2,3,4,6,7,8} =B
Dtran[A,b]=ε-closure(move(A,b))= ε-closure({5})= {1,2,4,5,6,7}=C

Dtran[B,a]=ε-closure(move(B,a)) – ε-closure({3,8})= {1,2,3,4,6,7,8} =B
Dtran[B,b]=ε-closure(move(B,b))= ε-closure({5,9})= {1,2,4,5,6,7,9}=D

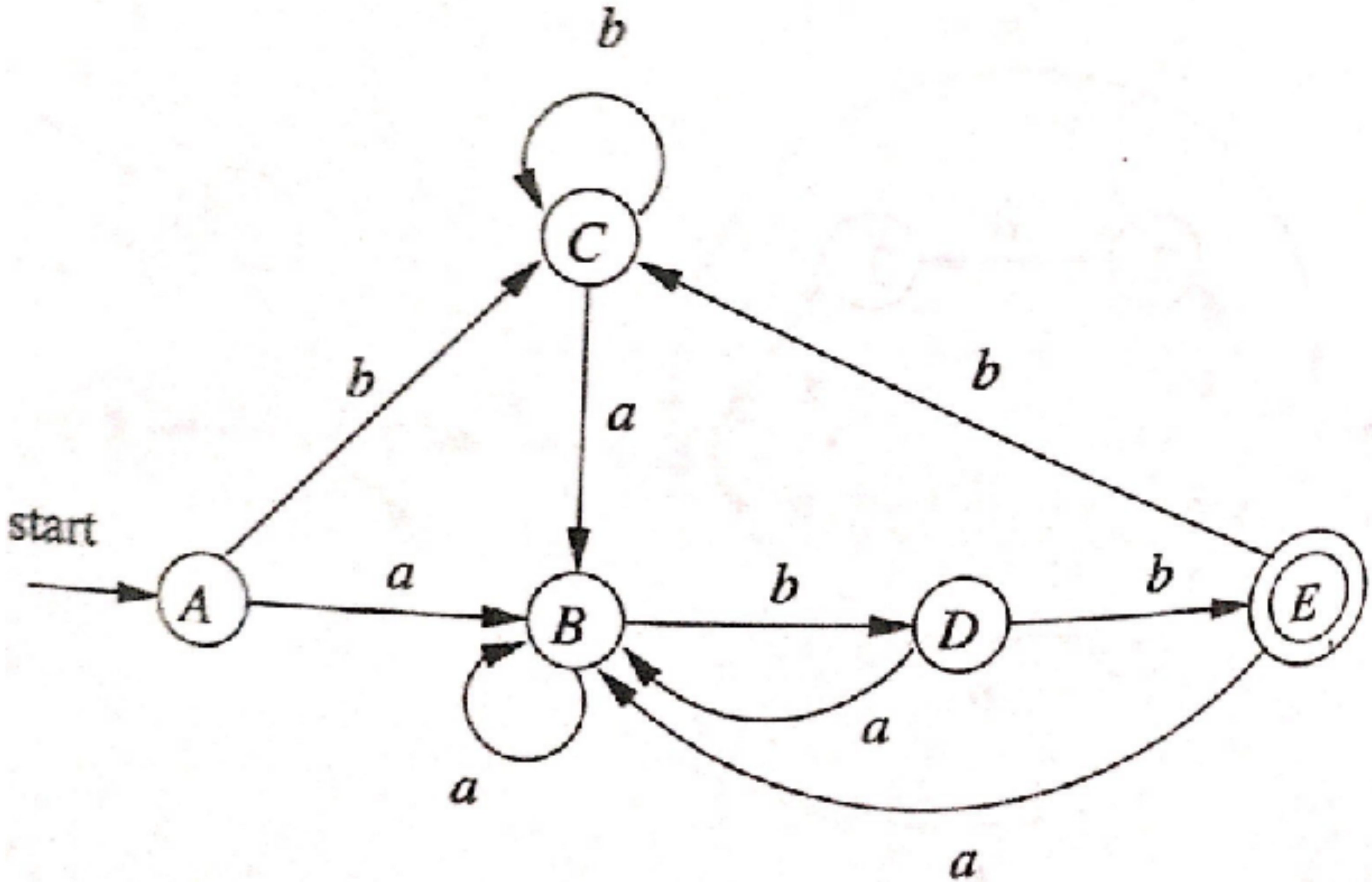
Dtran[C,a]=ε-closure(move(C,a)) – ε-closure({3,8})= {1,2,3,4,6,7,8} =B
Dtran[C,b]=ε-closure(move(C,b))= ε-closure({5})= {1,2,4,5,6,7}=C

Dtran[D,a]=ε-closure(move(D,a)) – ε-closure({3,8})= {1,2,3,4,6,7,8} =B
Dtran[D,b]=ε-closure(move(D,b))= ε-closure({5,10})= {1,2,4,5,6,7,10} =E

Dtran[E,a]=ε-closure(move(E,a)) – ε-closure({3,8})= {1,2,3,4,6,7,8} =B
Dtran[E,b]=ε-closure(move(E,b))= ε-closure({5})= {1,2,4,5,6,7}=C

NFA State	DFA State	a	b
{0,1,2,4,7}	A	B	C
{1,2,3,4,6,7,8}	B	B	D
{1,2,4,5,6,7}	C	B	C
{1,2,4,5,6,7,9}	D	B	E
{1,2,4,5,6,7,10}	E	B	C

Transition table Dtran for DFA



b
C
b
a
b
start
a
b
b
A
B
D
E
a
a
a

INPUT: A DFA D with set of states S, input alphabets , start state s0 , and set of accepting states F.
OUTPUT: A DFA D' accepting the same language as D and having as few states as possible.
METHOD:

1. Start with an initial partition Π with two groups, F and S — F, the accepting and non accepting states of D.
2. Apply the procedure of to construct a new partition Π_{new}
   initially, let $\Pi_{new} = \Pi$;
   for (each group G of Π)
   { partition G into subgroups such that two states s and t are in the same subgroup if and only if for all input symbols a, states s and t have transitions on a to states in the same group of Π /* at worst, a state will be in a subgroup by itself */
   replace G in Π_{new} by the set of all subgroups formed;
   }
3. If $\Pi_{new} = \Pi$, let $\Pi_{final} = \Pi$ and continue with step (4). Otherwise, repeat step (2) with Π_{new} in place of Π .
4. Choose one state in each group of Π_{final} as the representative for that group. The representatives will be the states of the minimum-state DFA D'. The other components of D' are constructed as follows:

(a) The start state of D' is the representative of the group containing the start state of D.

(b) The accepting states of D' are the representatives of those groups that contain an accepting state of D. Note that each group contains either only accepting states, or only non-accepting states, because we started by separating those two classes of states, and the procedure always forms new groups that are subgroups of previously constructed groups.

(c) Let s be the representative of some group G of Π_{final}, and let the transition of D from s on input ‘a’ be to state t. Let r be the representative of t’s group H. Then in D', there is a transition from s to r on input ‘a’. Note that in D, every state in group G must go to some state of group H on input a, or else, group G would have been split according to the procedure .

Transition table Dtran for DFA

NFA State	DFA State	a	b
{0,1,2,4,7}	A	B	C
{1,2,3,4,6,7,8}	B	B	D
{1,2,4,5,6,7}	C	B	C
{1,2,4,5,6,7,9}	D	B	E
{1,2,4,5,6,7,10}	E	B	C

{A,B,C,D} {E}

{A,B,C} {D} {E}

{A,C} {B} {D} {E}

Transition table of minimum-state DFA

State	a	b
A	B	A
B	B	D
D	B	E
E	B	A

Minimization Algorithm for DFA

Construct a partition = { A, Q - A } of the set of states Q ;

Π_{new} := new_partition(Π)

while (Π_{new} != Π)

Π := Π_{new} ;

Π_{new} := new_partition(Π)

Π_{final} := Π;

function new_partition()

for each set S of Π **do**

partition S into subsets such that two states p and q of S are in the same subset of S if and only if for each input symbol, p and q make a transition to (states of) the same set of Π .(The states which have the transition on each input symbol in the alphabet to the same group are grouped together)

The subsets thus formed are sets of the output partition in place of S.

If S is not partitioned in this process, S remains in the output partition.

End

Minimum DFA M_1 is constructed from Π_{final} as follows:

Select one state in each set of the partition Π_{final} as the representative for the set. These representatives are states of minimum DFA M_1.

Let p and q be representatives i.e. states of minimum DFA M_1. Let us also denote by p and q the sets of states of the original DFA M represented by p and q, respectively. Let s be a state in p and t a state in q. If a transition from s to t on symbol a exists in M, then the minimum DFA M_1 has a transition from p to q on symbol a.

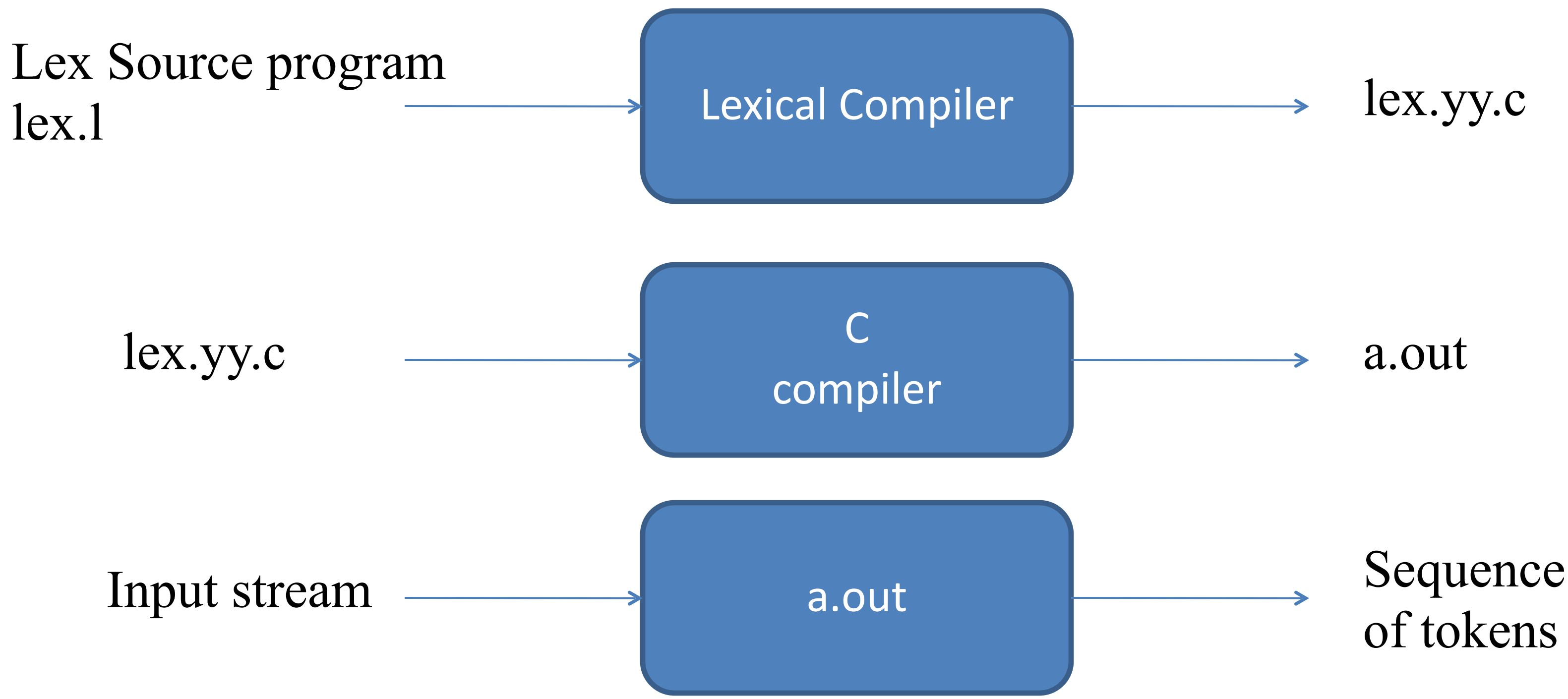
The start state of M_1 is the representative which contains the start state of M.

The accepting states of M_1 are representatives that are in A.

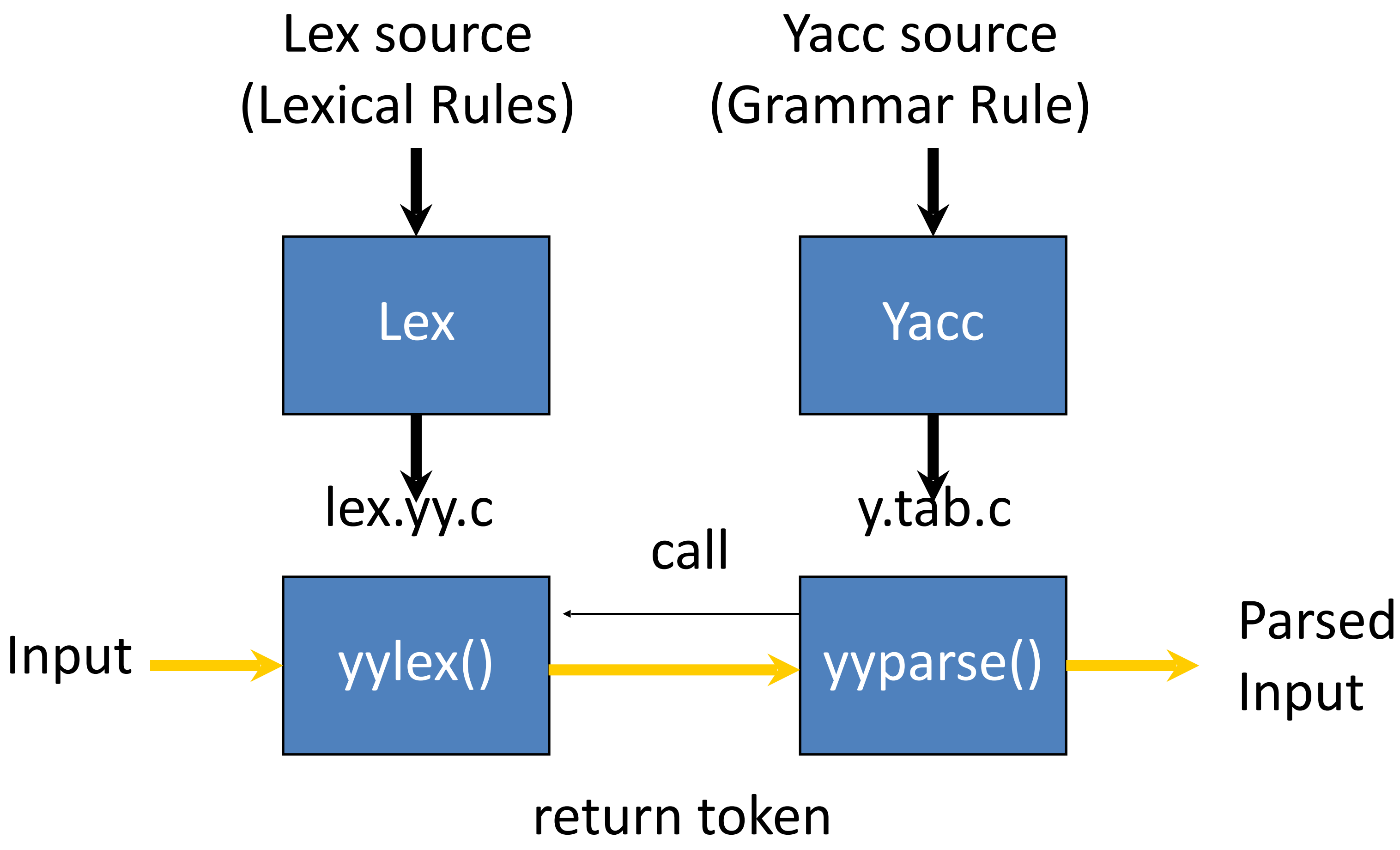
Lex -- a Lexical Analyzer Generator

Given tokens specified as regular expressions, Lex automatically generates a routine that recognizes the tokens.

Lexical Analyzer Generator - Lex



Lex with Yacc



Structure of Lex programs

The lex input file consists of three sections, separated by a line with %% in it:

declarations
%%
translation rules → Pattern {Action}
%%
auxiliary functions

Definitions Section

- The definitions section contains declarations of simple name definitions to simplify the scanner specification.
- Name definitions have the form:

```
name definition
```

- Example:

```
DIGIT       [0-9]
ID          [a-z][a-z0-9]*
```

Rules Section

Rules: <regular expression> <action>

Each regular expression specifies a token.

Default action for anything that is not matched: **copy to the output**

- The rules section of the lex input contains a series of rules of the form:

```
pattern action
```

- Example:

```
{ID} printf( "An identifier: %s\n", yytext );
```

- The *yytext* and *yylength* variable.
- If action is empty, the matched token is discarded.

Action

Action: C/C++ code fragment specifying what to do when a token is recognized.

- If the action contains a '{', the action spans till the balancing '}' is found, as in C.
- The *return* statement, as in C.
- In case no rule matches: simply copy the input to the standard output (A default rule).

User Code Section

- The user code section is simply copied to `lex.yy.c`
- The presence of this section is optional; if it is missing, the second `%%` in the input file may be skipped.
- In the definitions and rules sections, any indented text or text enclosed in `%{` and `%}` is copied exactly to the output (with the `%{}`'s removed).

- lex program examples:
  - ‘lex ex1.l’ produces the lex.yy.c file that contains a routine yylex().
    - The *int yylex()* routine is the scanner that finds all the regular expressions specified.
      - yylex() returns a non-zero value (usually token id) normally.
      - yylex() returns 0 when end of file is reached.
  - Need a drive to test the routine. Main.c is an example.
    - Have a *yywrap()* function in the lex file (return 1)
      - Something to do with compiling multiple files.

yylex() is a function of return type int. LEX automatically defines *yylex()* in *lex.yy.c* but does not call it. The programmer must call yylex() in the Auxiliary functions section of the LEX program. LEX generates code for the definition of yylex() according to the rules specified in the Rules section.

- LEX declares the function yywrap() of return-type int in the file *lex.yy.c* . LEX does not provide any definition for yywrap(). yylex() makes a call to yywrap() when it encounters the end of input. If yywrap() returns zero (indicating *false*) yylex() assumes there is more input and it continues scanning from the location pointed to by yyin. If yywrap() returns a non-zero value (indicating true), yylex() terminates the scanning process and returns 0 (i.e. "wraps up"). If the programmer wishes to scan more than one input file using the generated lexical analyzer, it can be simply done by setting yyin to a new input file in yywrap() and return 0.
- As LEX does not define yywrap() in lex.yy.c file but makes a call to it under yylex(), the programmer must define it in the Auxiliary functions section or provide %option noyywrap in the declarations section.

Review of Lex Predefined Variables

Name	Function
`char *yytext`	pointer to matched string
`int yyleng`	length of matched string
`FILE *yyin`	input stream pointer
`FILE *yyout`	output stream pointer
`int yylex(void)`	call to invoke lexer, returns token
`char* yymore(void)`	return the next token
`int yyless(int n)`	retain the first n characters in yytext
`int yywrap(void)`	wrapup, return 1 if done, 0 if not done
`ECHO`	write matched string
`REJECT`	go to the next alternative rule
`INITAL`	initial start condition
`BEGIN`	condition switch start condition

Usage

To run Lex on a source file, type

```
lex scanner.l
```

- It produces a file named lex.yy.c which is a C program for the lexical analyzer.
- To compile lex.yy.c, type

```
cc lex.yy.c -ll
```

- To run the lexical analyzer program, type

```
./a.out < inputfile > output
file
```

lex1.l

```
%{int count=0;
%}
chars [A-Za-z]
number [0-9]
delim [" "\n\t]
ws {delim}+
words {chars}+
numbers {number}+
%%
if printf("%s\n",yytext);
then printf("%s\n",yytext);
else printf("%s\n",yytext);
"<" printf("%s\n",yytext);
{words} {count++;}
{numbers} printf("digits %s\n",yytext);
%%
void main()
{
yylex();
printf("There are total %d words\n", count);
}
int yywrap()
{return 1;}
```

```
e1.l
%{
int count=0;
%}
chars [A-Za-z]
numbers [0-9]
delim [" "\n\t]
ws {delim}+
words {chars}+
%%{words} {count++;}
%%void main()
{
extern FILE* yyin;
yyin=fopen("input.txt","r");
yylex();
printf("%d",count);
}
int yywrap()
{return 1;}
```

input.txt

```
we are students from g if < else 3333
```

```
psg@psg-OptiPlex-3060:~$ flex lex1.l
psg@psg-OptiPlex-3060:~$ cc lex.yy.c -ll
psg@psg-OptiPlex-3060:~$ ./a.out <input.txt
   if
 <
 else
 3333
There are total 5 words
psg@psg-OptiPlex-3060:~$ ./a.out <input.txt >
out.txt
```

Example

```
%{
      /* definitions of manifest constants
      LT, LE, EQ, NE, GT, GE,
      IF, THEN, ELSE, ID, NUMBER, RELOP */
%}

/* regular definitions
delim         [ \t\n]
ws            {delim}+
letter        [A-Za-z]
digit [0-9]
id            {letter}({letter}|{digit})*
number        {digit}+(\.{digit}+)?(E[+-]?{digit}+)?

%%
{ws} {/* no action and no return */}
if            {return(IF);}
then{return(THEN);}
else {return(ELSE);}
{id}  {yylval = (int) installID(); return(ID); }
{number}      {yylval = (int) installNum(); return(NUMBER);}
…
```

```
Int installID() {/* funtion to install the
    lexeme, whose first character is
    pointed to by yytext, and whose
    length is yyleng, into the symbol
    table and return a pointer thereto
    */
}

Int installNum() { /* similar to
    installID, but puts numerical
    constants into a separate table */
}
```