

Lexical Analyzer — Scanner

ASU Textbook Chapter 3.1, 3.3, 3.4, 3.6, 3.7, 3.5

Tsan-sheng Hsu

tshsu@iis.sinica.edu.tw

`http://www.iis.sinica.edu.tw/~tshsu`

Main tasks

- Read the input characters and produce as output a sequence of **tokens** that the parser uses for syntax analysis.
- **Lexeme** : a sequence of characters matched by a given **pattern** for a **token** .

- Example:

Lexeme	pi	=	3.1416	;
token	ID	ASSIGN	FLOAT-LIT	SEMI-COL
- patterns:
 - ▷ *identifier (variable) starts with a letter and follows by letters, digits or “_”;*
 - ▷ *floating point number starts with a string of digits + a dot + another string of digits;*

Strings

■ Definitions and operations.

- **alphabet** : a finite set of characters (symbols);
- **string** : a finite sequence of characters from the alphabet;
- $|S|$: length of a string S ;
- **empty string**: ϵ ;
- xy : concatenation of string x and y
 $\epsilon x \equiv x \equiv x \epsilon$;
- **exponentiation**:
 - ▷ $s^0 \equiv \epsilon$;
 - ▷ $s^i \equiv s^{i-1}s, i > 0$.

Parts of a string

- **Parts of a string: example string “necessary”**
 - **prefix: deleting zero or more tailing characters;** eg: “nece”
 - **suffix: deleting zero or more leading characters;** eg: “ssary”
 - **substring: deleting prefix and suffix;** eg: “ssa”
 - **subsequence: deleting zero or more not necessarily contiguous symbols;**
eg: “ncsay”
 - **Proper** prefix, suffix, substring or subsequence: one that cannot equal to the original string;

Language

- **Language** : any set of strings over an alphabet.
- **Operations on languages:**
 - **union:** $L \cup M = \{s \mid s \in L \text{ or } s \in M\}$;
 - **concatenation:** $LM = \{st \mid s \in L \text{ and } t \in M\}$;
 - $L^0 = \{\epsilon\}$;
 - **Kleene closure** : $L^* = \bigcup_{i=0}^{\infty} L^i$;
 - **Positive closure** : $L^+ = \bigcup_{i=1}^{\infty} L^i$;
 - $L^* = L^+ \cup \{\epsilon\}$.

Regular expressions

- A regular expression r denotes a language $L(r)$, also called a regular set .
- Operations on regular expressions:

regular expression	language
\emptyset	empty set $\{\}$
ϵ	the set containing the empty string $\{\epsilon\}$
a	$\{a\}$ where a is a legal symbol
$r s$	$L(r) \cup L(s)$ — union
rs	$L(r)L(s)$ — concatenation
r^*	$L(r)^*$ — Kleene closure

- Example:

$a b$	$\{a, b\}$
$(a b)(a b)$	$\{aa, ab, ba, bb\}$
a^*	$\{\epsilon, a, aa, aaa, \dots\}$
$a a^*b$	$\{a, b, ab, aab, \dots\}$
C identifier	$(A B \dots) ((A B \dots) — (0 1 \dots) — “_”)^*$

Regular definitions

- For simplicity, give names to regular expressions.

- format: name \rightarrow regular expression.
- example 1: digit $\rightarrow 0|1|2|\dots|9$.
- example 2: letter $\rightarrow a|b|c|\dots|z|A|B|\dots$.

$$r^* \qquad r^+|\epsilon$$

$$r^+ \qquad rr^*$$

- Notational standards:

$$r^? \qquad r|\epsilon$$

$$[abc] \qquad a|b|c$$

$$[a-z] \qquad a|b|c|\dots|z$$

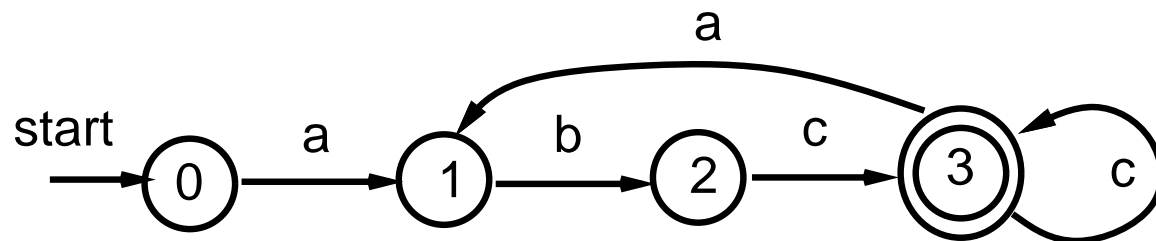
- Example: C variable name: $[A-Za-z][A-Za-z0-9_]^*$

Non-regular sets

- **Balanced or nested construct**
 - Example: if ... then ... else
 - Recognized by context free grammar .
- **Matching strings:**
 - $\{w cw\}$, where w is a string of a 's and b 's and c is a legal symbol.
- **Remark:** anything that needs to “memorize” something happened in the past.

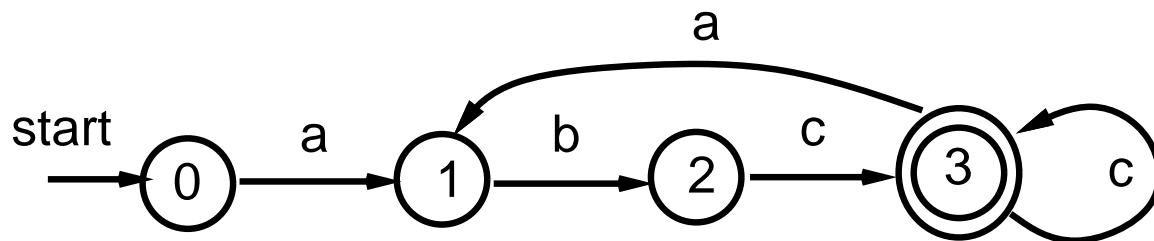
Finite state automata (FA)

- FA is a mechanism used to recognize tokens specified by a regular expression.
- Definition:
 - A finite set of states.
 - A set of transitions, labeled by characters.
 - A starting state.
 - A set of final (accepting) states.
- Example: **transition graph** for the regular expression $(abc^+)^+$



Transition graph and table for FA

■ Transition graph:



■ Transition table:

	<i>a</i>	<i>b</i>	<i>c</i>
0	1		
1		2	
2			3
3	1		3

- Rows are input symbols.
- Columns are current states.
- Entries are resulting states.
- Along with the table, a start state and a set of accepting states are also given.

This is also called a GOTO table.

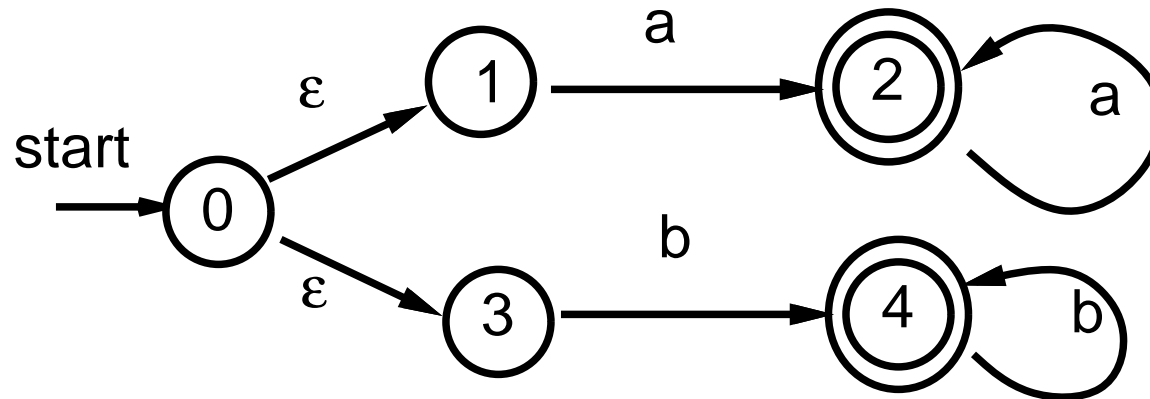
Types of FA's

■ Deterministic FA (DFA):

- has a unique next state for a transition;
- does not contain ϵ -transitions, that is a transition take ϵ as the input symbol.

■ Nondeterministic FA (NFA):

- has more than one next state for a transition;
- contains ϵ -transitions.
- Example: $aa^*|bb^*$.



How to execute a DFA

■ Algorithm:

$s \leftarrow$ starting state;

while there are inputs do

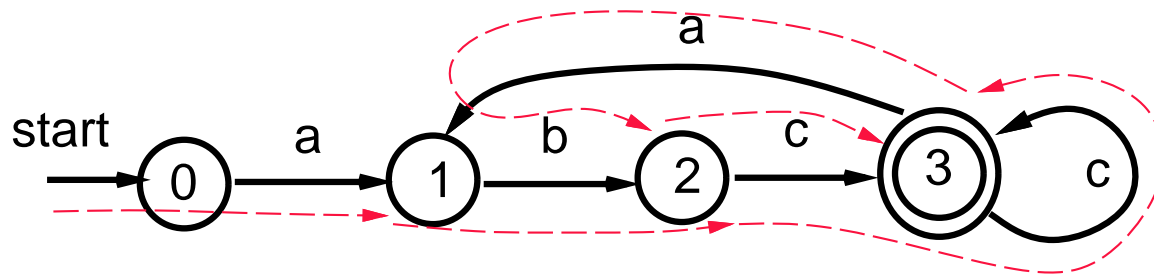
$s \leftarrow$ **Table**[$s, input$]

end while

if $s \in accpetingstates$ then **ACCEPT** else **REJECT**

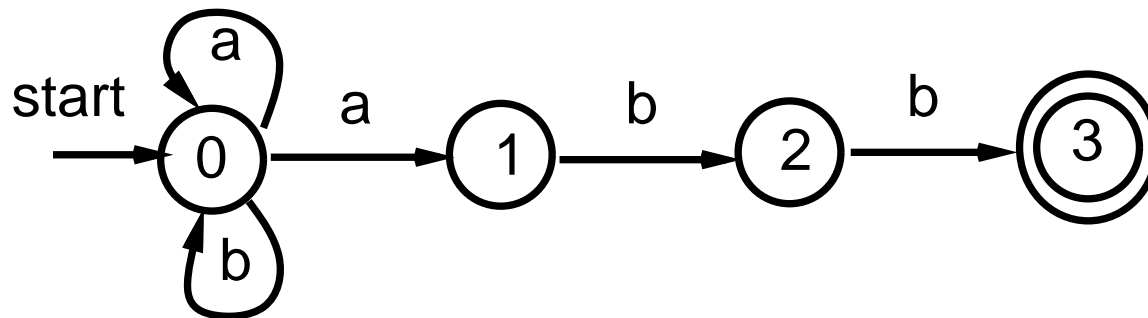
■ Example: input “abccabc”. The accepting path:

$0 \xrightarrow{a} 1 \xrightarrow{b} 2 \xrightarrow{c} 3 \xrightarrow{c} 3 \xrightarrow{a} 1 \xrightarrow{b} 2 \xrightarrow{c} 3$



How to execute an NFA (informally)

- An NFA accepts an input string x if and only if there is some path in the transition graph initiating from the starting state to some accepting state such that the edge labels along the path spell out x .
- Could have more than one path. (Note DFA has at most one.)
- Example: regular expression: $(a|b)^*abb$; input $aabb$



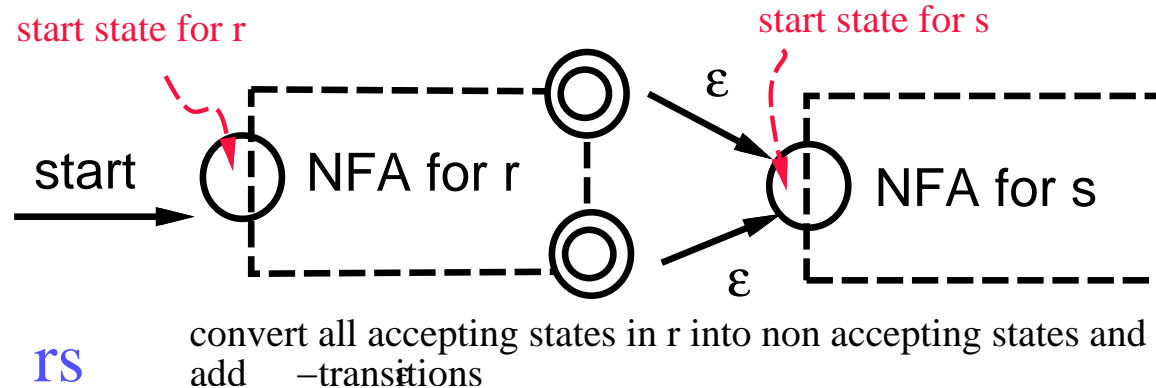
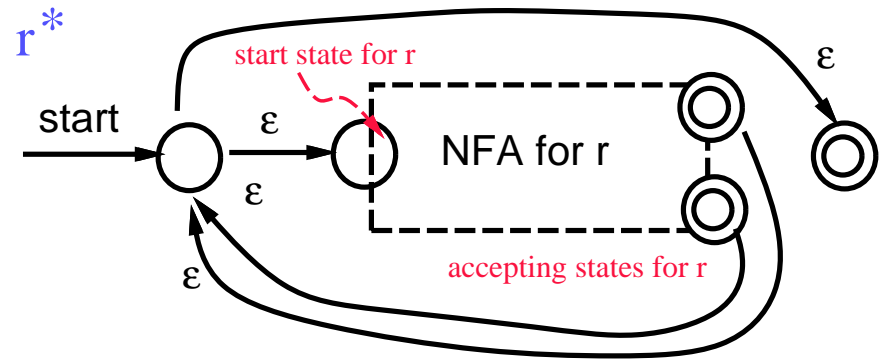
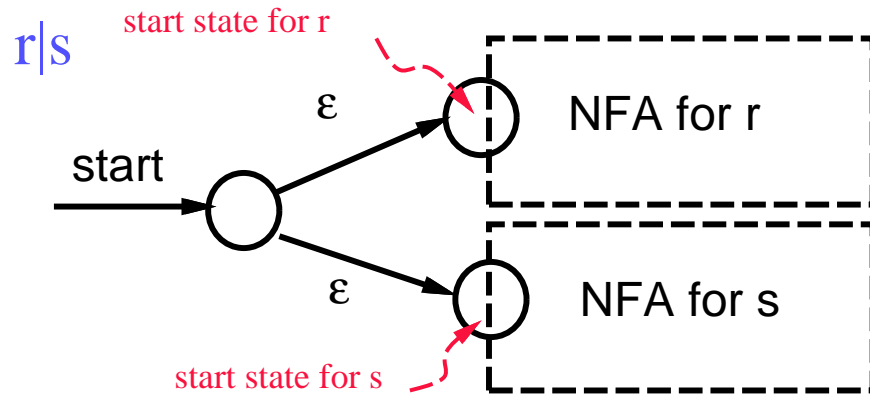
	a	b
0	{0,1}	{0}
1		{2}
2		{3}

$0 \xrightarrow{a} 0 \xrightarrow{a} 1 \xrightarrow{b} 2 \xrightarrow{b} 3$ **Accept!**

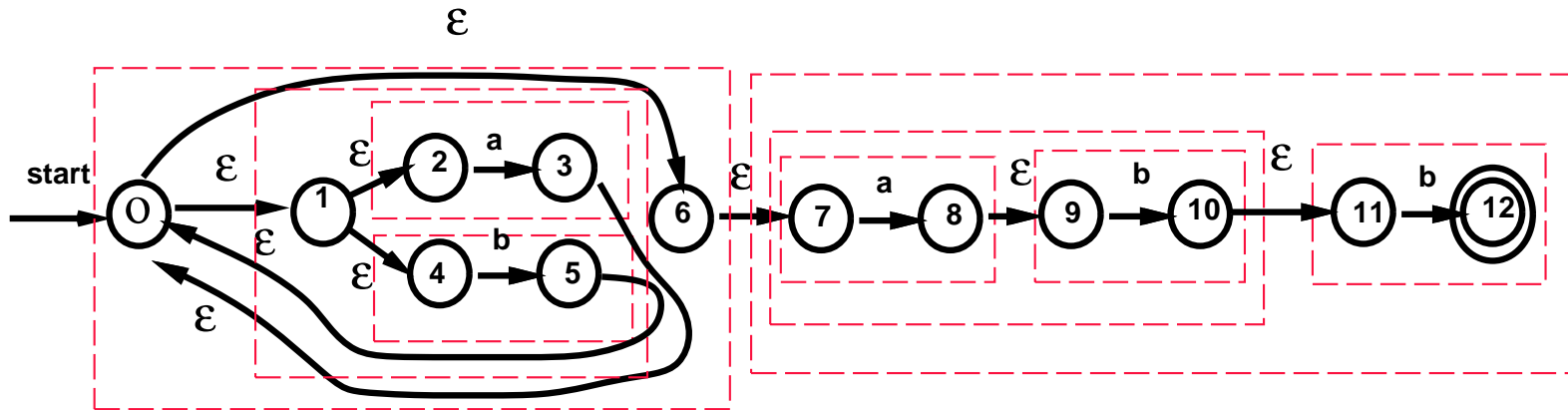
$0 \xrightarrow{a} 0 \xrightarrow{a} 0 \xrightarrow{b} 0 \xrightarrow{b} 0$ **Reject!**

From regular expressions to NFA's

- **Structural decomposition:**
 - atomic items: \emptyset , ϵ and a legal symbol.



Example: $(a|b)^*abb$



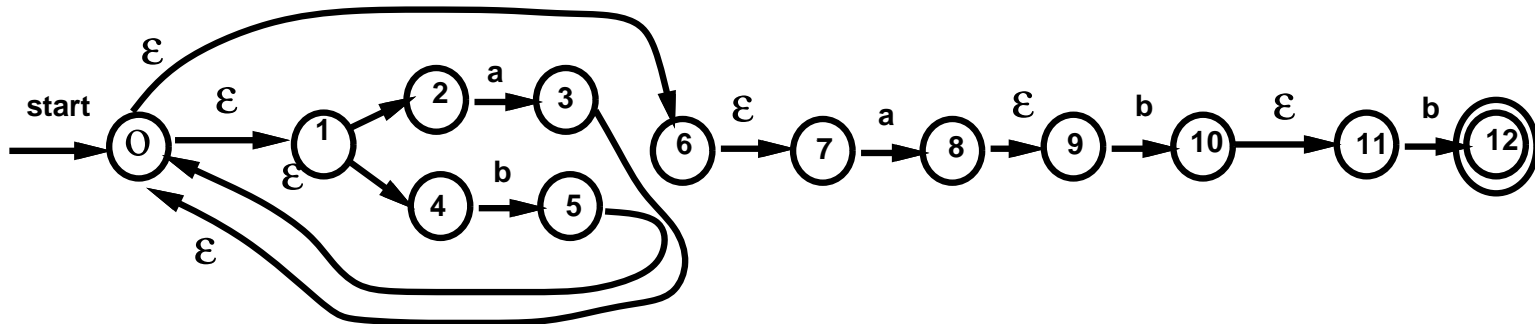
- This construction produces only ϵ -transitions, never multiple transitions for an input symbol.
- It is possible to remove all ϵ -transitions from an NFA and replace them with multiple transitions for an input symbol, and vice versa.

Construction theorems

- **Theorem #1:**
 - Any regular expression can be expressed by an NFA.
 - Any NFA can be converted into a DFA.
- **That is, any regular expression can be expressed by a DFA.**
- **How to convert an NFA to a DFA:**
 - Find out what is the set of possible states that can be reached from an NFA state using ϵ -transitions.
 - Find out what is the set of possible states that can be reached from an NFA state on an input symbol.
- **Theorem #2:**
 - Every DFA can be expressed as a regular expression.
 - Every regular expression can be expressed as a DFA.
 - DFA and regular expressions have the same expressive power.
- **How about the power of DFA and NFA?**

Converting an NFA to a DFA

- **Definitions:** let T be a set of states and a be an input symbol.
 - ϵ -closure(T): the set of NFA states reachable from some state $s \in T$ using ϵ -transitions.
 - $move(T, a)$: the set of NFA states to which there is a transition on the input symbol a from state $s \in T$.
 - Both can be computed using standard graph algorithms.
 - ϵ -closure($move(T, a)$): the set of states reachable from a state in T for the input a .
- **Example: NFA for $(a|b)^*abb$**



- ϵ -closure($\{0\}$) = $\{0, 1, 2, 4, 6, 7\}$, that is the set of all possible start states
- $move(\{2, 7\}, a) = \{3, 8\}$

Subset construction algorithm

- In the converted DFA, each state represents a subset of NFA states.

- $T \xrightarrow{a} \epsilon\text{-closure}(\text{move}(T, a))$

- **Subset construction algorithm :**

initially, we have an unmarked state labeled with $\epsilon\text{-closure}(\{s_0\})$, where s_0 is the starting state.

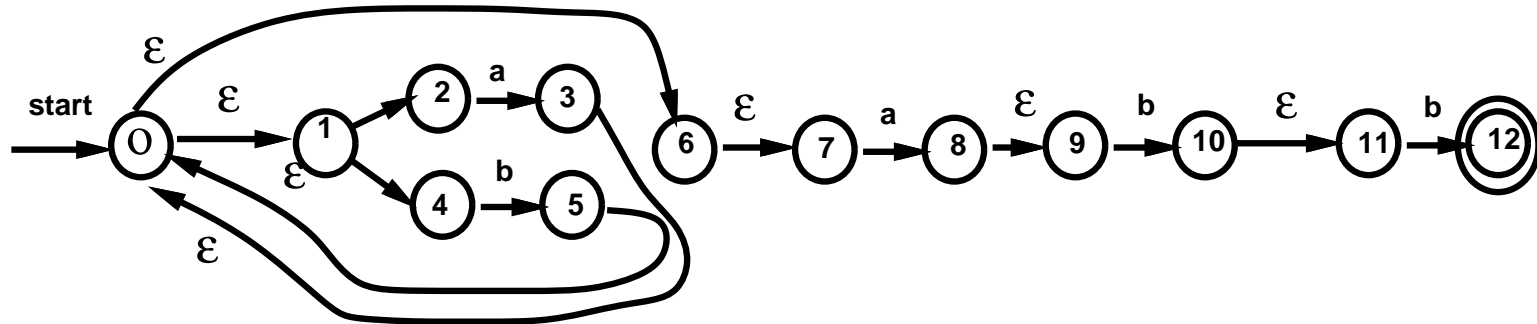
while there is an unmarked state with the label T **do**

- ▷ *mark the state with the label T*
- ▷ *for each input symbol a do*
- ▷ *$U \leftarrow \epsilon\text{-closure}(\text{move}(T, a))$*
- ▷ *if U is a subset of states that is never seen before*
- ▷ *then add an unmarked state with the label U*
- ▷ *end for*

end while

- **New accepting states:** those contain an original accepting state.

Example



First step:

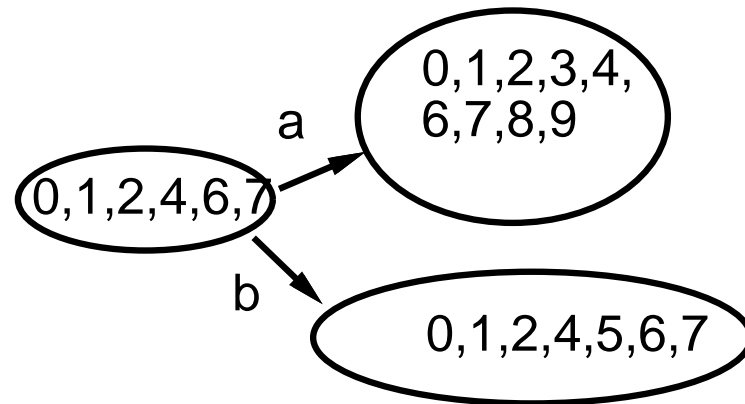
- ϵ -closure($\{0\}$) = $\{0, 1, 2, 4, 6, 7\}$

- $move(\{0, 1, 2, 4, 6, 7\}, a)$ = $\{3, 8\}$

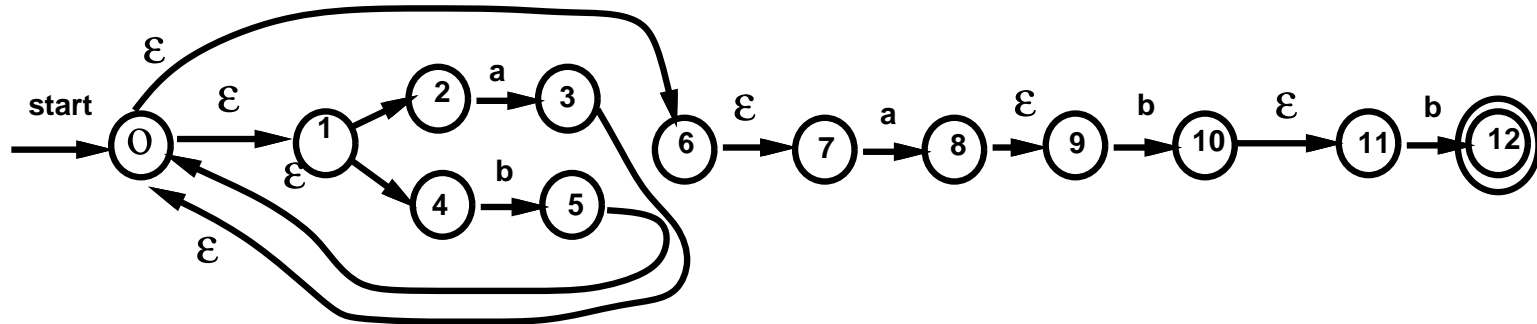
- ϵ -closure($\{3, 8\}$) = $\{1, 2, 3, 4, 6, 7, 8\}$

- $move(\{0, 1, 2, 4, 6, 7\}, b)$ = $\{5\}$

- ϵ -closure($\{5\}$) = $\{1, 2, 4, 5, 6, 7\}$



Example — cont.

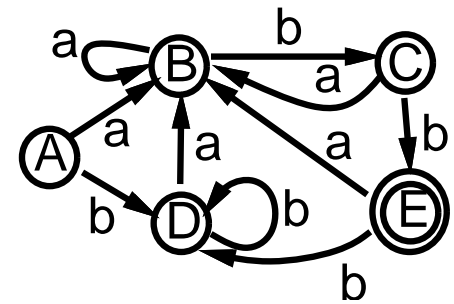


transition table:

states:

- $A = \{0, 1, 2, 4, 6, 7\}$
- $B = \{0, 1, 2, 3, 4, 6, 7, 8, 9\}$
- $C = \{0, 1, 2, 4, 5, 6, 7, 10, 11\}$
- $D = \{0, 1, 2, 4, 5, 6, 7\}$
- $E = \{0, 1, 2, 4, 5, 6, 7, 12\}$

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



Algorithm for executing an NFA

- **Algorithm:** s_0 is the starting state, F is the set of accepting states.

```
 $S \leftarrow \epsilon\text{-closure}(\{s_0\})$   
while next input  $a$  is not EOF do  
    ▷  $S \leftarrow \epsilon\text{-closure}(\text{move}(S, a))$   
end while  
if  $S \cap F \neq \emptyset$  then ACCEPT else REJECT
```

- **Execution time is $O(r^2 \cdot s)$, where**
 - r is the number of NFA states, and
 - s is the length of the input.
 - Need $O(r^2)$ time in running $\epsilon\text{-closure}(T)$ assuming linked list data structures are used for the ϵ transitions and we can use a linear-time hashing routine to remove duplicated states.

Trade-off in executing NFA's

- Can also convert an NFA to a DFA and then execute the equivalent DFA.
 - Running time: linear in the input size.
 - Space requirement: linear in the size of the DFA.
- Catch:
 - May get $O(2^r)$ DFA states by converting an r -state NFA.
 - The converting algorithm may also takes $O(2^r)$ time.

- Time-space tradeoff:

	space	time
NFA	$O(r^2)$	$O(r^2 \cdot s)$
DFA	$O(2^r)$	$O(s)$

 - If memory is cheap or it is going to run again, then use the DFA approach;
 - otherwise, use the NFA approach.

LEX

- An UNIX utility.
- An easy way to use regular expressions to do lexical analysis.
- Convery your LEX program into an equivalent C program.
- Depending on implementation, may use NFA or DFA algorithms.
- `file.l` \longrightarrow `lex file.l` \longrightarrow `lex.yy.c`
- `lex.yy.c` \longrightarrow `cc -ll lex.yy.c` \longrightarrow `a.out`
- `input` \longrightarrow `a.out` \longrightarrow output sequence of tokens

LEX formats

■ Source format:

- Declarations — a set of regular definitions, i.e., names and their regular expressions.
- %%
- Translation rules — actions to be taken when patterns are encountered.
- %%
- Auxiliary procedures

■ Global variables:

- *yyleng*: length of current string
- *yytext*: current string
- *yylex()*: the scanner routine

LEX formats – cont.

- **Declarations:**
 - variables: using C format
 - manifest constants: using C format; identifiers declared to represent constants
 - regular expressions.

- **Translation rules:**

P_1 {action₁}

if regular expression P_1 is encountered, then action₁ is performed.

- **LEX internals: regular expressions \longrightarrow NFA \longrightarrow DFA**

test.l — Declarations

```
%{  
    /* some initial C programs */  
#define BEGINSYM 1  
#define INTEGER 2  
#define IDNAME 3  
#define REAL 4  
#define STRING 5  
#define SEMICOLONSYM 6  
#define ASSIGNSYM 7  
%}  
Digit      [0-9]  
Letter     [a-zA-Z]  
IntLit     {Digit}+  
Id         {Letter}({Letter}|{Digit}|_)*
```

test.l — Rules

```
%%  
[ \t\n] { /* skip white spaces */}  
[Bb] [Ee] [Gg] [Ii] [Nn]          {return(BEGINSYM);}  
{IntLit}                          {return(INTEGER);}  
{Id}                               {  
    printf("var has %d characters, ",yyleng);  
    return(IDNAME);  
}  
({IntLit}[.]{IntLit})([Ee][+-]?{IntLit})? {return(REAL);}  
\" [^\n]*\"    {stripquotes(); return(STRING);}  
";"          {return(SEMICOLONSYM);}  
":="        {return(ASSIGNSYM);}  
.           {printf("error --- %s\n",yytext);}
```

test.l — Procedures

```
%%
/* some final C programs */
stripquotes()
{
    /* handling string within a quoted string */
    int frompos, topos=0, numquotes = 2;
    for(frompos=1; frompos<yyleng; frompos++){
        yytext[topos++] = yytext[frompos];
    }
    yytext[topos] = '\0';
}

void main(){
    int i;
    i = yylex();
    while(i>0 && i < 8){
        printf("<%s> is %d\n",yytext,i);
        i = yylex();    }    }
```

Sample run

```
austin% lex test.l
austin% cc lex.yy.c -ll
austin% cat data
Begin
123.3  321.4E21
x := 365;
"this is a string"
austin% a.out < data
<Begin> is 1
<123.3> is 4
<321.4E21> is 4
var has 1 characters, <x> is 3
<:=> is 7
<365> is 2
<;> is 6
<this is a string> is 5
%austin
```

More LEX formats

- Special format requirement:

P_1

```
{ action1
  ...
}
```

Note: { and } must indent.

- LEX special characters (operators):

“ \ [] ^ - ? . * + | () \$ { } % < >

- When there is any ambiguity in matching, prefer
 - longest possible match;
 - earlier expression if all matches are of equal length.

LEX internals

- **How to find a longest possible match if there are many legal matches?**
 - If an accepting state is encountered, do not immediately accept.
 - Push this accepting state and the current input position into a stack and keep on going until no more matches is possible.
 - Pop from the stack and execute the actions for the popped accepting state.
 - Resume the scanning from the popped current input position.
- **How to find the earliest match if all matches are of equal length?**
 - Number the accepting states according to the order in the expressions.
 - If you are in multiple accepting states, execute the action associated with the least indexed accepting state.

Practical considerations

■ key words v.s. Reserved word

- key word:

- ▷ *def: word has a well-defined meaning in a certain context.*
- ▷ *example: FORTRAN, PL/1, ...*
if if then else = then ;
id id id id
- ▷ *Makes compiler to work harder!*

- reserved word:

- ▷ *def: regardless of context, word cannot be used for other purposes.*
- ▷ *example: COBOL, ALGOL, PASCAL, C, ADA, ...*
- ▷ *task of compiler is simpler*
- ▷ *reserved words cannot be used as identifiers*
- ▷ *listing of reserved words is tedious for the scanner, also makes scanner large*
- ▷ *solutions: treat them as identifiers, and use a table to check whether it is a reserved word.*

Practical considerations – cont.

- **Multi-character lookahead:** how many more characters ahead do you have to look in order to decide which pattern to match?
- **FORTRAN:** lookahead until difference is saw without counting blanks.
 - `DO 10 I = 1, 15` \equiv a loop statement.
 - `DO 10 I = 1.15` \equiv an assignment statement for the variable `DO10I`.
- **PASCAL:** lookahead 2 characters with 2 or more blanks treating as one blank.
 - `10..100:` needs to look 2 characters ahead to decide this is not part of a real number.
- **LEX lookahead operator “/”:** r_1/r_2 : match r_1 only if it is followed by r_2 ; note that r_2 is not part of the match.
 - This operator can be used to cope with multi-character lookahead.
 - How is this implemented in LEX?