Lecture 3 : Lexical Analysis - Continued
Compiler Design (CS 3007)

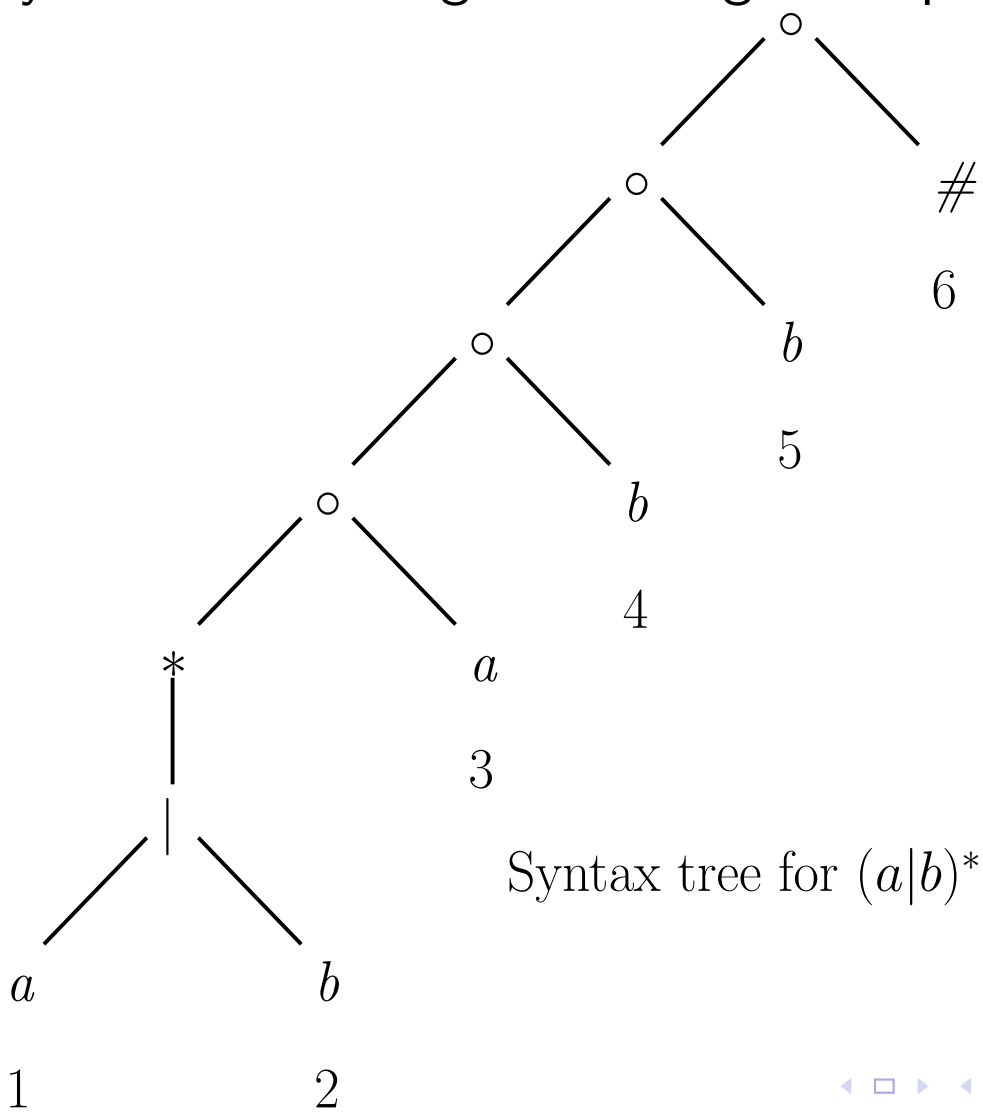
S. Pyne[1] & T. K. Mishra[2]

Assistant Professor[1,2]
Computer Science and Engineering Department
National Institute of Technology Rourkela
{pynes,mishrat}@nitrkl.ac.in

August 7, 2020

Constructing DFA from Regular expression

- Considering regular expression $(a|b)^*abb$
- Construct syntax tree for augmented regular expression $(a|b)^*abb\#$



Syntax tree for $(a|b)^*abb\#$

Functions computed from syntax tree for RE $(r)\#$

- $L(n)$ - language of subexpression rooted at syntax-tree node n
- $nullable(n)$ is true for node n iff $\epsilon \in L(n)$

$$nullable(n) = \begin{cases} \textbf{true}, \textbf{ if } n \text{ is a leaf labeled } \epsilon \\ \textbf{false}, \textbf{ if } n \text{ is a leaf with position } i \\ nullable(c_1) \textbf{ or } nullable(c_2), \textbf{ if or-node } n = c_1|c_2 \\ nullable(c_1) \textbf{ and } nullable(c_2), \textbf{ if cat-node } n = c_1c_2 \\ \textbf{true}, \textbf{ if star-node } n = c_1^* \end{cases}$$

- $firstpos(n)$ is set of positions for the first symbol of string $w \in L(n)$

$$firstpos(n) = \begin{cases} \emptyset, \textbf{ if } n \text{ is a leaf labeled } \epsilon \\ \{i\}, \textbf{ if } n \text{ is a leaf with position } i \\ firstpos(c_1) \cup firstpos(c_2), \textbf{ if or-node } n = c_1|c_2 \\ firstpos(c_1) \cup firstpos(c_2), \textbf{ if cat-node } n = c_1c_2 \textbf{ and } nullable(c_1) = \textbf{true} \\ firstpos(c_1), \textbf{ if cat-node } n = c_1c_2 \textbf{ and } nullable(c_1) = \textbf{false} \\ firstpos(c_1), \textbf{ if star-node } n = c_1^* \end{cases}$$

Functions computed from syntax tree for RE $(r)\#$

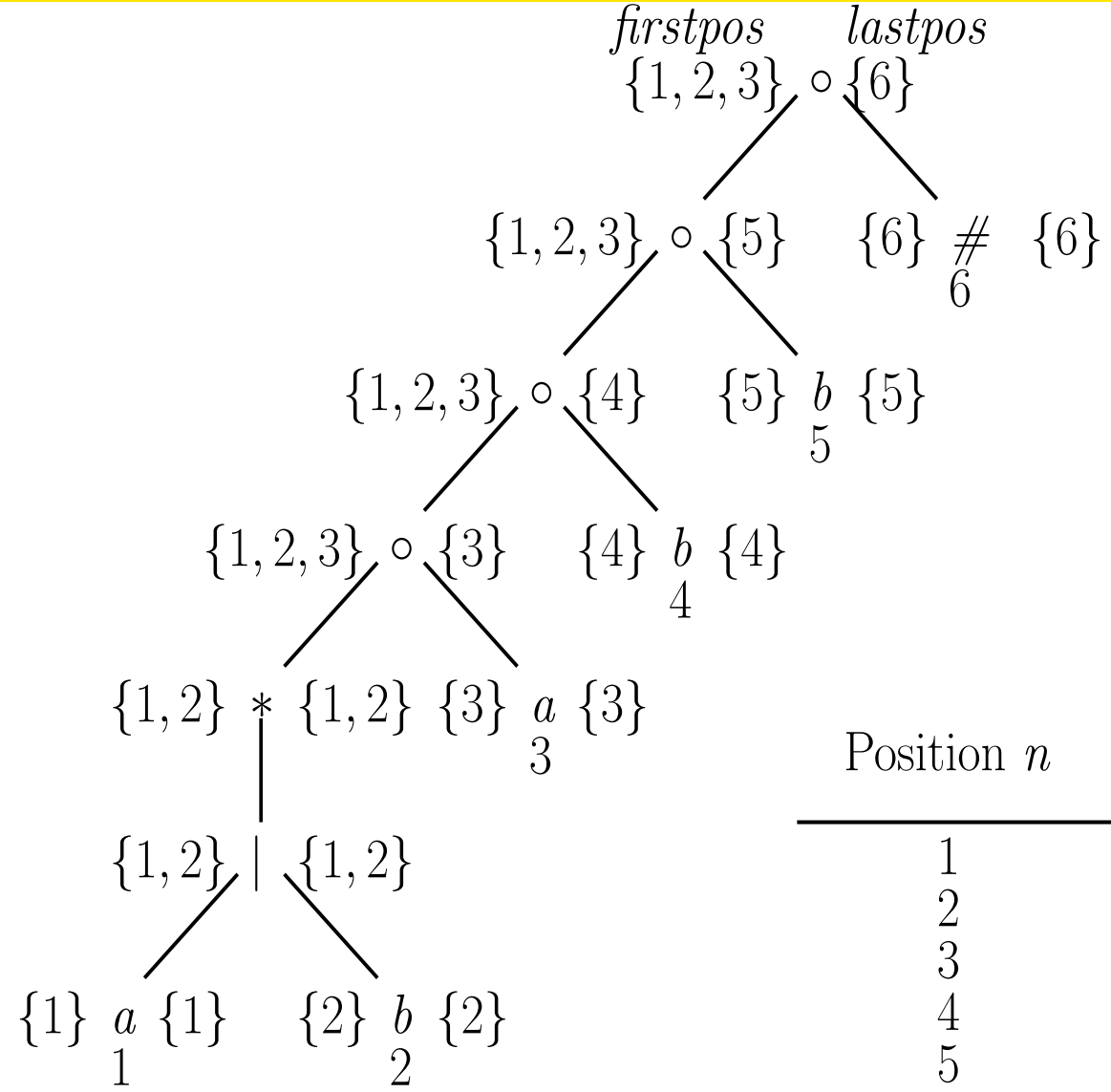
- $lastpos(n)$ is set of positions for the last symbol of string $w \in L(n)$

$$lastpos(n) = \begin{cases} \emptyset, \textbf{ if } n \text{ is a leaf labeled } \epsilon \\ \{i\}, \textbf{ if } n \text{ is a leaf with position } i \\ lastpos(c_1) \cup lastpos(c_2), \textbf{ if or-node } n = c_1|c_2 \\ lastpos(c_1) \cup lastpos(c_2), \textbf{ if cat-node } n = c_1c_2 \textbf{ and } nullable(c_2) = \textbf{true} \\ lastpos(c_2), \textbf{ if cat-node } n = c_1c_2 \textbf{ and } nullable(c_2) = \textbf{false} \\ lasttpos(c_1), \textbf{ if star-node } n = c_1^* \end{cases}$$

- p and q are positions of symbols a_p and a_q in string $x \in L((r)\#)$
- $followpos(p) = \{q | \exists x = a_1 a_2 \cdots a_p a_q \cdots a_{|x|} \in L((r)\#)\}$

$$followpos(i) = \begin{cases} firstpos(c_2) \forall \text{ position } i \in lastpos(c_1), \textbf{ if cat-node } n = c_1c_2 \\ firstpos(n), \textbf{ if star-node } n = c_1^* \textbf{ and } i \in lastpos(n) \end{cases}$$

Firstpos and lastpos

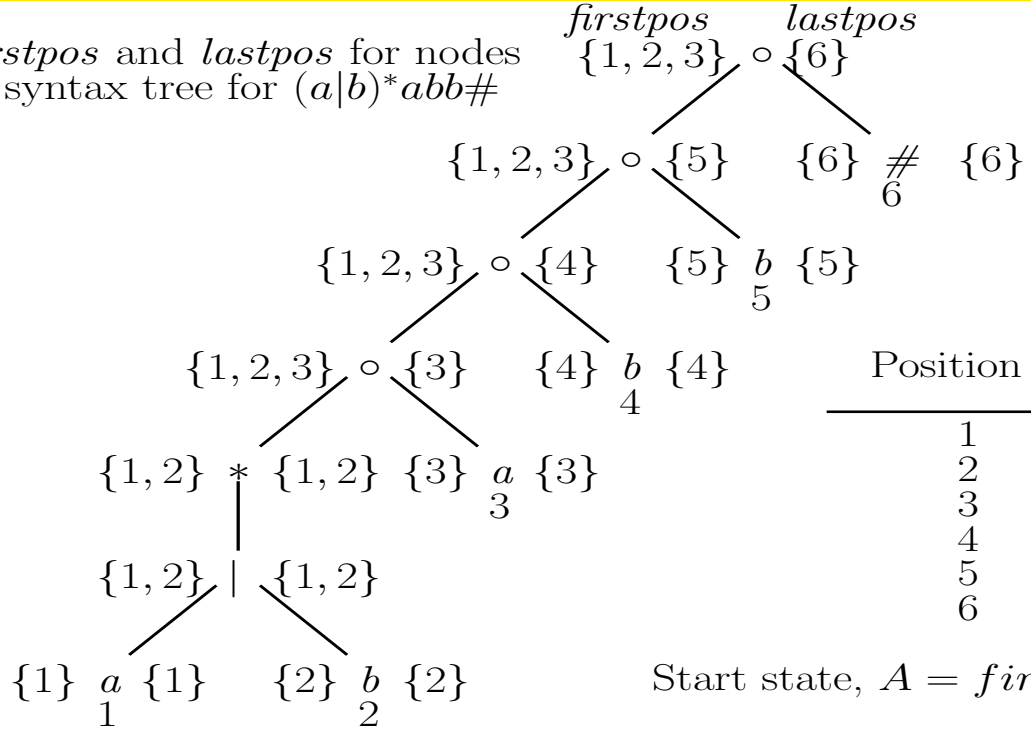


firstpos and *lastpos* for nodes in syntax tree for $(a|b)^*abb\#$

Position n	$followpos(n)$
1	$\{1, 2, 3\}$
2	$\{1, 2, 3\}$
3	$\{4\}$
4	$\{5\}$
5	$\{6\}$
6	$\emptyset$

Construction of DFA

firstpos and *lastpos* for nodes in syntax tree for $(a|b)^*abb\#$



Position n	$followpos(n)$
1	$\{1, 2, 3\}$
2	$\{1, 2, 3\}$
3	$\{4\}$
4	$\{5\}$
5	$\{6\}$
6	$\emptyset$

Start state, $A = firstpos(root) = \{1, 2, 3\}$

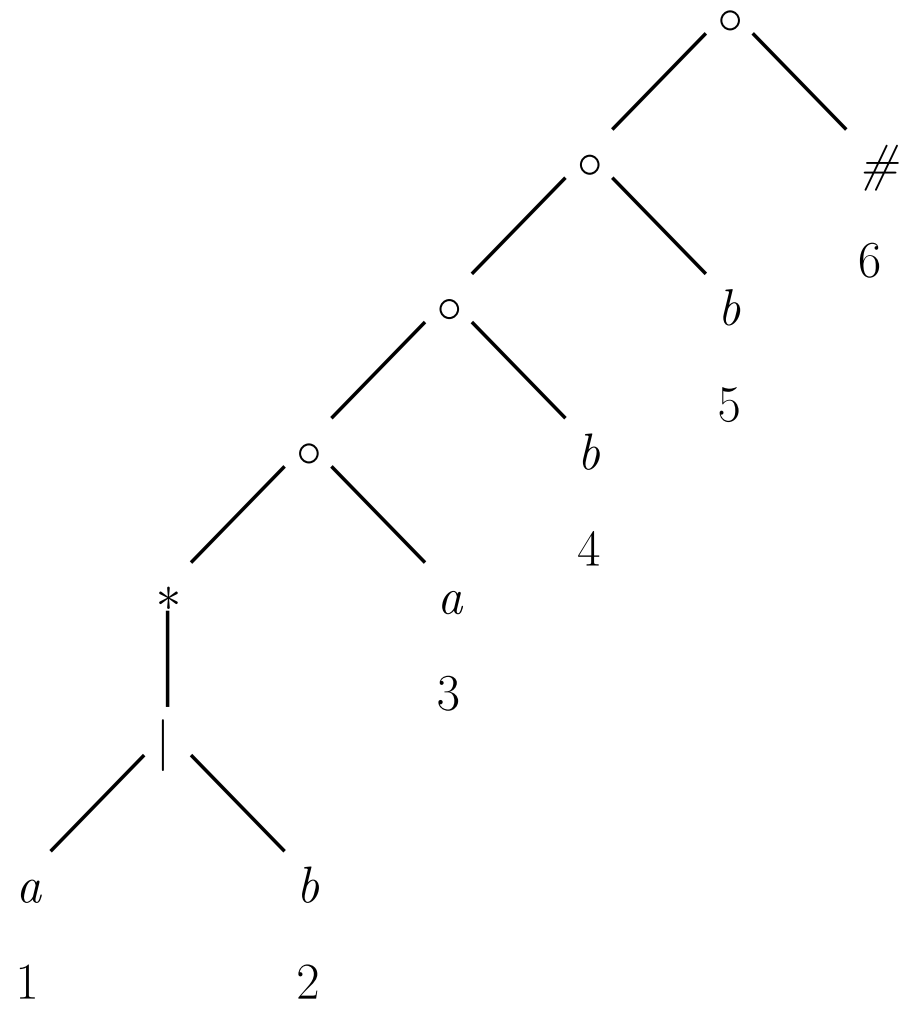
In A, positions 1 and 3 correspond to a, 2 corresponds to b
$\delta(A, a) = followpos(1) \cup followpos(3) = \{1, 2, 3, 4\} = B$
$\delta(A, b) = followpos(2) = \{1, 2, 3\} = A$

In B, positions 1 and 3 correspond to a, 2 and 4 correspond to b
$\delta(B, a) = followpos(1) \cup followpos(3) = \{1, 2, 3, 4\} = B$
$\delta(B, b) = followpos(2) \cup followpos(4) = \{1, 2, 3, 5\} = C$

In C, positions 1 and 3 correspond to a, 2 and 5 correspond to b
$\delta(C, a) = followpos(1) \cup followpos(3) = \{1, 2, 3, 4\} = B$
$\delta(C, b) = followpos(2) \cup followpos(5) = \{1, 2, 3, 6\} = D$, accept state

In D, positions 1 and 3 correspond to a, 2 correponds to b and 5 corresponds to $\#$
$\delta(D, a) = followpos(1) \cup followpos(3) = \{1, 2, 3, 4\} = B$
$\delta(D, b) = followpos(2) \cup followpos(6) = \{1, 2, 3\} = A$

Final DFA of $(a|b)^*abb\#$

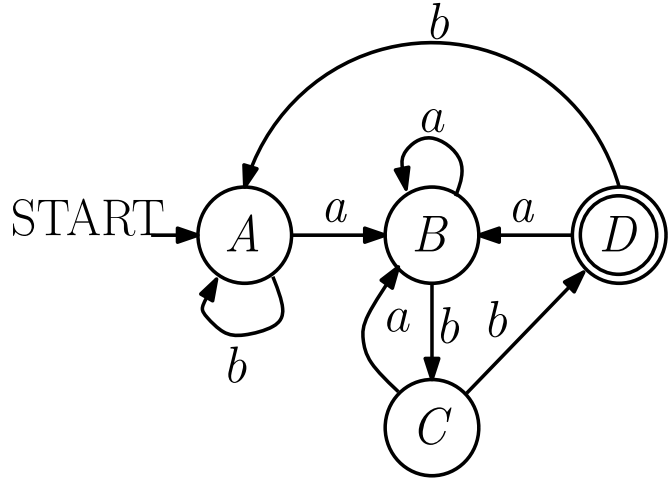


Syntax tree for $(a|b)^*abb\#$

DFA

	DFA State	Input a	Input b
Start	$A = \{1, 2, 3\}$	B	A
	$B = \{1, 2, 3, 4\}$	B	C
	$C = \{1, 2, 3, 5\}$	B	D
Accept	$D = \{1, 2, 3, 6\}$	B	A

Transition table

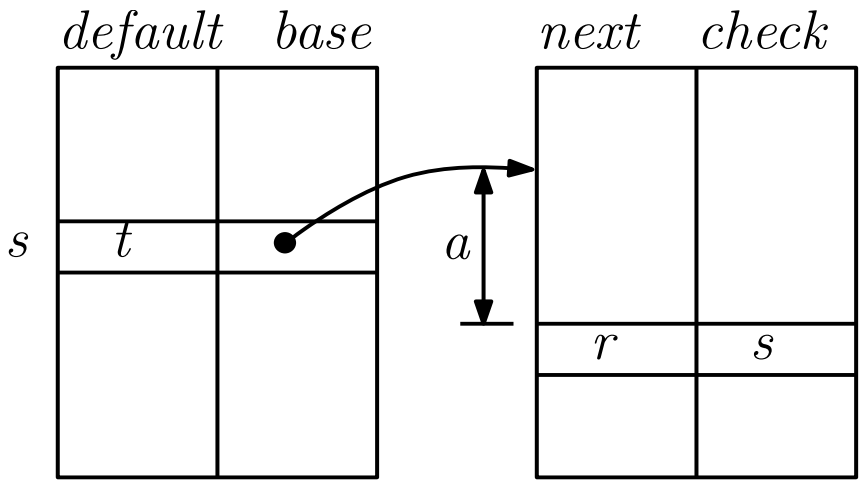


Transition diagram

Data Structures for DFA

- For a DFA with set of states Q, alphabet Σ and transition function δ
- A $|Q| \times |\Sigma|$ 2-D array with elements $\delta[s][a]$, $s \in Q$ and $a \in \Sigma$
- $\delta(s, a) = r \implies \delta[s][a] = r$
- Easy implementation
- State transition time is $O(1)$
- Space consumed $O(|Q| \times |\Sigma|)$
- Unused entries - wastage of memory
- Compression - a compact and slower structure using four arrays
- *base* array - determines base location of entries for state s ($base[s]$)
- *next* array - entry for next state of s on input a ($next[base[s] + a]$)
- *check* array - stores $check[base[s] + a] = s$ for a valid $\delta(s, a)$
- *default* array - alternative base location if $check[base[s] + a] \neq s$

Data Structures for DFA



$nextState(s, a)$
if $check[base[s] + a] = s$ **then**
return $next[base[s] + a]$
else
return $nextState(default[s], a])$
end if

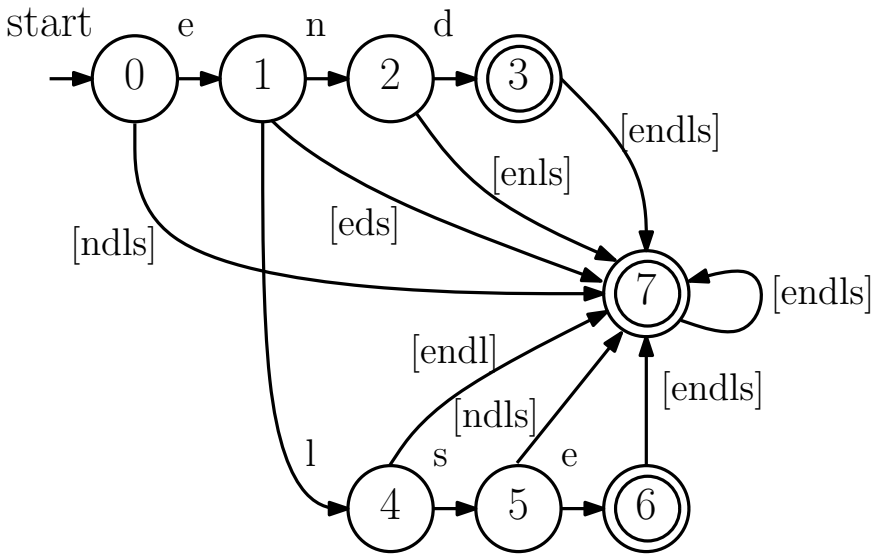
- For a $\delta(s, a) = r$ $nextState(s, a)$ is computed
- Location $l = base[s] + a$
- if $check[l] = s$ then entry is valid and next state is $r = next[l]$
- if $check[l] \neq s$ then next state is $t = default[s]$
- Repeat $nextState(t, a)$ with t as current state

Example

- Let us consider a DFA to accept the following regular expressions and return tokens

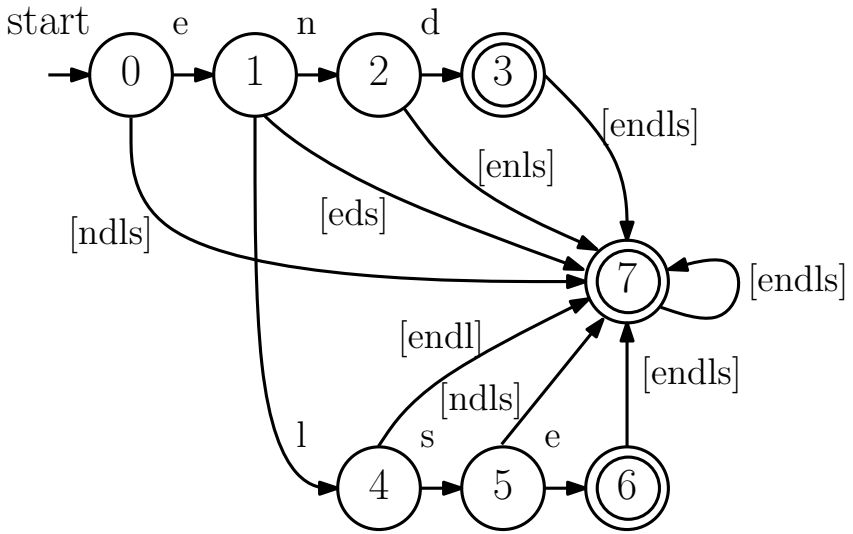
  end {return END}
  else {return ELSE}
  [endls]+ {return IDENTIFIER}

- 2-D array representation of DFA



State	Input				
	e	n	d	l	s
0	1	7	7	7	7
1	1	2	7	4	7
2	1	7	3	7	7
3	1	7	7	7	7
4	1	7	7	7	5
5	6	7	7	7	7
6	1	7	7	7	7
7	1	7	7	7	7

Compressed representation of DFA



State	Input				
	e	n	d	l	s
0	1	7	7	7	7
1	1	2	7	4	7
2	1	7	3	7	7
3	1	7	7	7	7
4	1	7	7	7	5
5	6	7	7	7	7
6	1	7	7	7	7
7	1	7	7	7	7

Symbol	e	n	d	l	s
Offset	0	1	2	3	4

index= *base*[state]+Offset(Symbol)

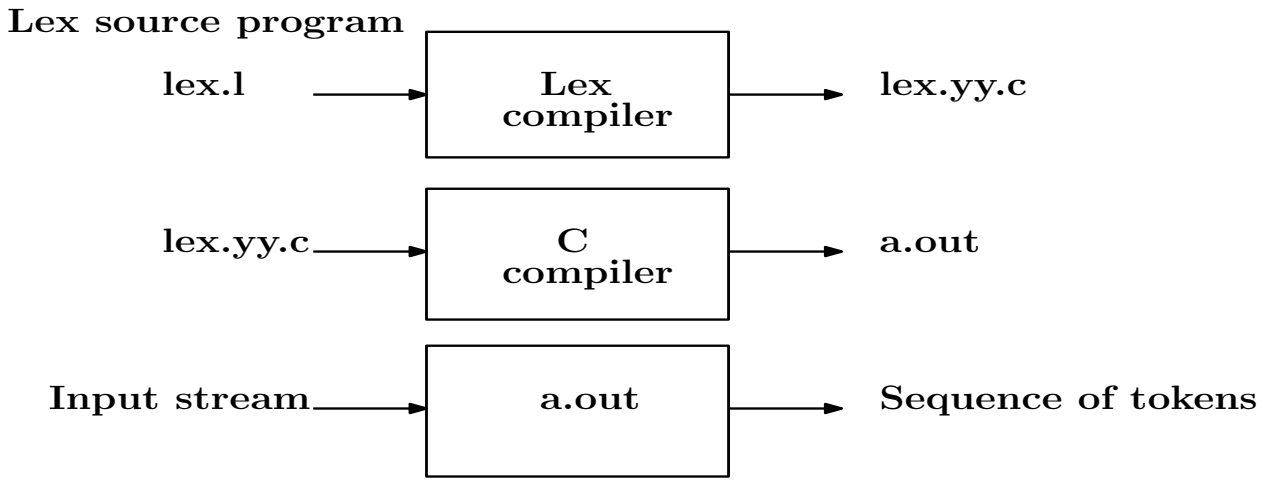
state	*default*	*base*
0	7	0
1	7	0
2	7	0
3	7	NULL
4	7	0
5	7	5
6	7	NULL
7	7	NULL

index	*next*	*check*
0	1	0
1	2	1
2	3	2
3	4	1
4	5	4
5	6	5

- $8 \times 5 = 40$ elements in naive representation
- $8 \times 2 + 6 \times 2 = 28$ elements in compressed representation
- 30% reduction in space

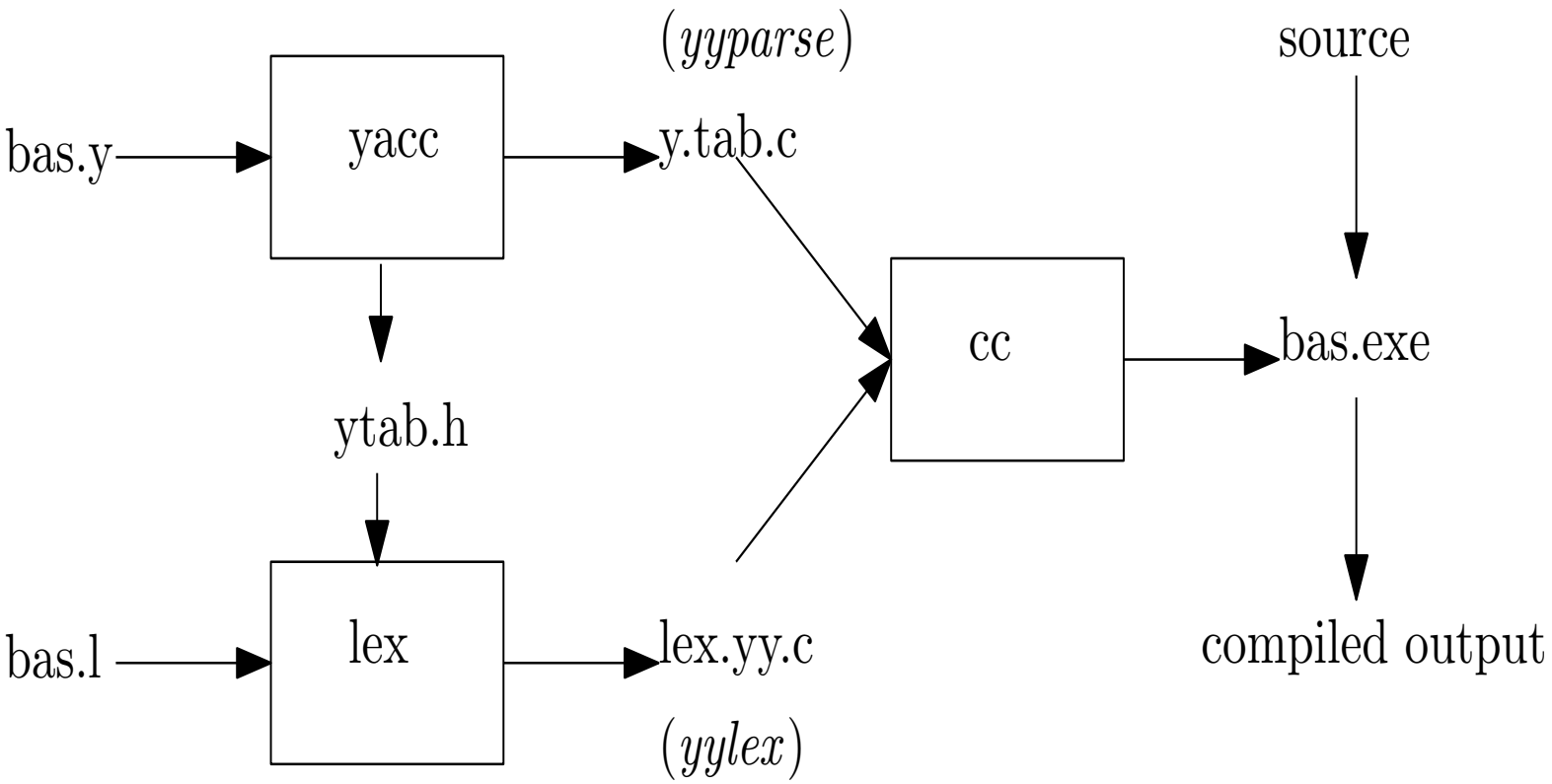
Lexical-Analyzer Generator Lex

- *Lex* tool = *Lex* language + *Lex* compiler
- More recent implementation is Flex
- Alternatives - JFlex uses Java, PyPI uses Python



- lex.l contains the regular expression
- lex.yy.c contains the lexical analyzer code in C language
- lex.yy.c is compiled to produce the lexical analyzer
- yylval - a global variable in lex.yy.c stores token attributes
- Token attribute - a numeric code, a pointer to symbol table or nothing
- yylval is shared between lexical analyer and parser

Building a Complier with Lex and Yacc



Structure of Lex Programs

- Format of a Lex program

declarations
%%
translation rules
%%
auxiliary functions

- declarations - variables, manifest contants and regular definitions
- translation rules - patterns with actions
- auxiliary functions - additional functions are used in the actions

- Format of each translation rule

Pattern { Action }

- Each pattern is a regular expression (may use regular definitions)
- Actions are fragment of code written in C

A Lex program to reconize following tokens

Token	Code	Value
begin	1	–
end	2	–
if	3	–
then	4	–
else	5	–
identifier	6	Pointer to symbol table
constant	7	Pointer to symbol table
<	8	1
<=	8	2
=	8	3
<>	8	4
>	8	5
>=	8	6

Tokens recognized

A Lex program - token_recognizer.l

```
%{
    #include<stdio.h>
    /* definitions of manifest contants */
    enum yytokentype {LT = 1, LE, EQ, NE, GT, GE,
        BEGIN, END, IF, THEN, ELSE, ID, CONSTANT};
%}
    /* regular definitions */
delim           [ \t\n]
ws              {delim}+
letter          [A-Z,a-z]
digit           [0-9]
id              {letter}({letter}|{digit})*
constant        {digit}+
%%
{ws}            {/* no action and no return */}
begin           {printf("BEGIN\n"); return BEGIN;}
end             {printf("END\n"); return END;}
if              {printf("IF\n"); return IF;}
then            {printf("THEN\n"); return THEN;}
else            {printf("ELSE\n"); return ELSE;}
{id}            {printf("ID\n"); yylval = (int) installID(); return ID;}
{constant}      {printf("CONSTANT\n"); yylval = (int) installConst();
                  return CONSTANT;}
"<"             {printf("LT\n"); yylval = LT; return RELOP;}
"<="            {printf("LE\n");yylval = LE; return RELOP;}
"="             {printf("LT\n"); yylval = EQ; return RELOP;}
"<>"            {printf("NE\n"); yylval = NE; return RELOP;}
">"             {printf("LT\n"); yylval = GT; return RELOP;}
">="            {printf("GE\n"); yylval = GE; return RELOP;}
%%
int installID(){ /* function to install the lexeme, whose
                    first character is pointed to yytext,
                    and whose length is yyleng, into the
                    symbol table and return a pointer thereto*/
    }
int installConst(){   /* similar to installID, but puts numerical
                         constants into a separate table */
    }
int main(){
        yylex(); /* scanner routine */
        return 0;
    }
```

Running token_recognizer.l

```
$ flex token_recognizer.l
$ gcc lex.yy.c -lfl
$ ./a.out
if total > 50 then
IF
ID
GT
CONSTANT
THEN
begin 22 end
BEGIN
CONSTANT
END
^D
$
```

- lex.yy.c is linked with flex library, -lfl
- Each time the program needs a token, it calls yylex()
- yylex() reads an input, matches pattern and returns token
- yylex() called again for next token

Flex Library -lfl

- default main routine with "while(yylex()!=0);"
- input() - lexer calls input() to fetch each of the matched characters
- unput() - unput(c) returns character c to input stream
- yyinput() and yyunput() - input() and unput() in C++ scanner
- yytext() - matched token is stored in null-terminated string yytext
- yyleng(yytext) - length of yytext
- yyless(n) - "push back" all but the first n characters of the token.
- yylex() and YY_DECL - yylex has no arguments, interacts through global variables. YY_DECL declares calling sequence and adds whatever arguments wanted for yytext
- yymore() - tell lex to append the next token to current one
- yyrestart() - yyrestart(f) makes the scanner read from open stdio file f
- yywrap() - on reaching EOF lexer calls yywrap() to find next job, returns 0 for continuing scanning, returns 1 for EOF

Thank you