Lexical Analysis

2CS701 Compiler Construction

Prof Monika Shah

Nirma University

Phase 1 : Lexical Analysis

Prof Monika Shah

Nirma University

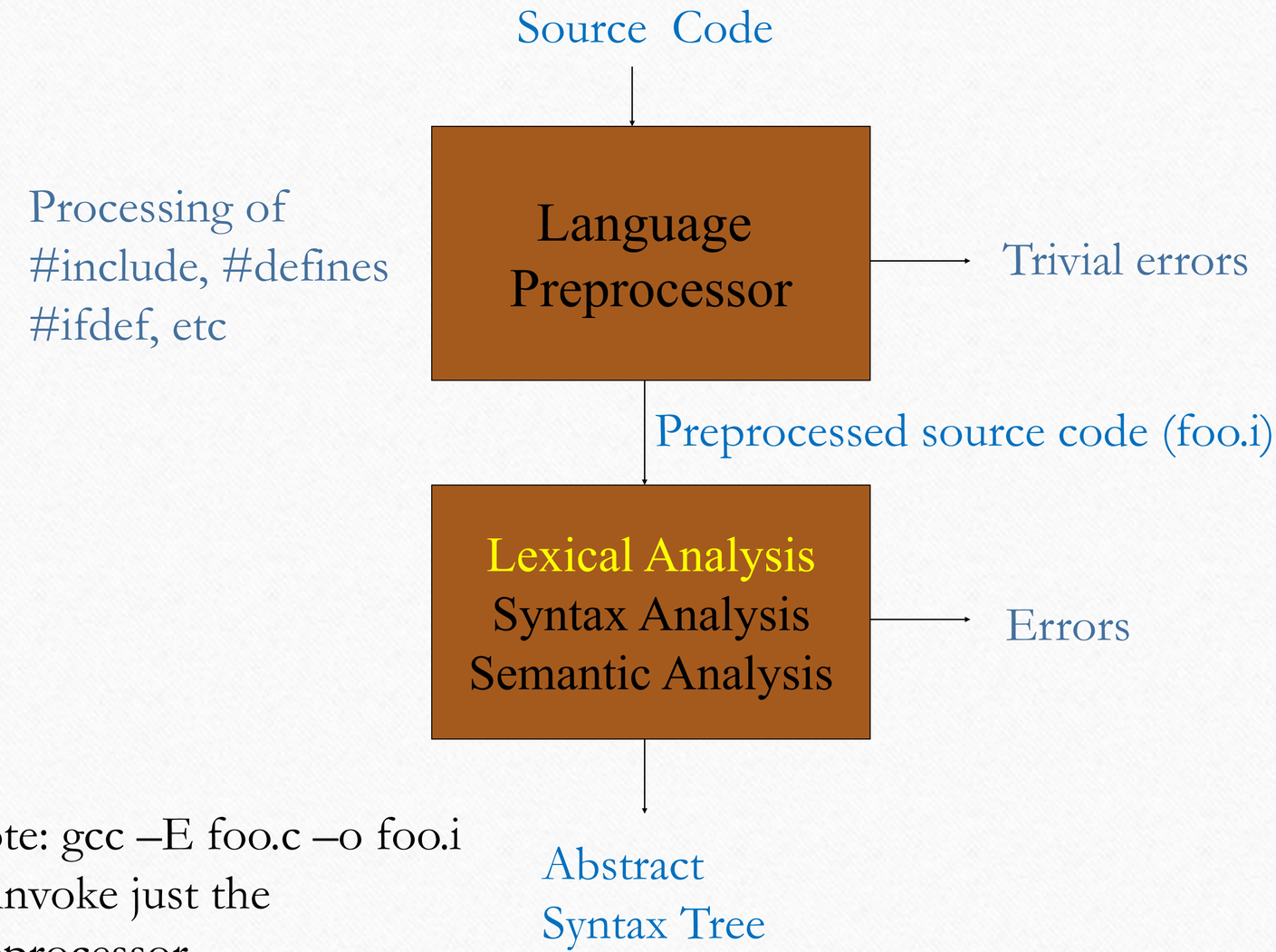
Ref : Ch.3 Compilers Principles, Techniques, and Tools by Alfred Aho, Ravi Sethi, and Jeffrey Ullman

Glimpse

- Role of the Lexical Analyzer
- Token, Lexeme, and Pattern
- How to describe patterns?
- Lexical Analyzer Generators
- Design of Lexical Analyzer
  - RE to DFA
  - Input Buffering
  - Sentinel

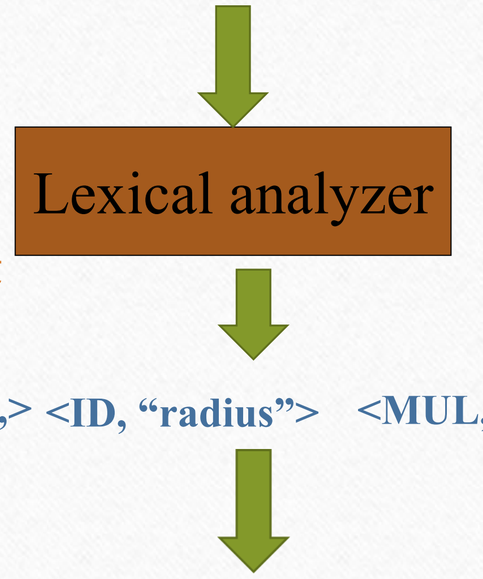
Role of Lexical Analyzer

- Scan source program
- Translate character stream to token stream
- Eliminate unnecessary tokens from token stream
- Enter Symbols along with location, type into Symbol Table
- Lexical error identification



Note: gcc –E foo.c –o foo.i to invoke just the preprocessor

area = 3.14 * radius * radius ; @ /*Equation*/ if . . .



Role - Transform multi-character input stream to token stream

Role – Identify Lexical error

Error: Invalid lexeme

<ID, “area”><ASSIGN,> <NUM, 3.14> <MUL,> <ID, “radius”> <MUL,> <ID,”radius”> <:,> @ <Comment, > <IF,>

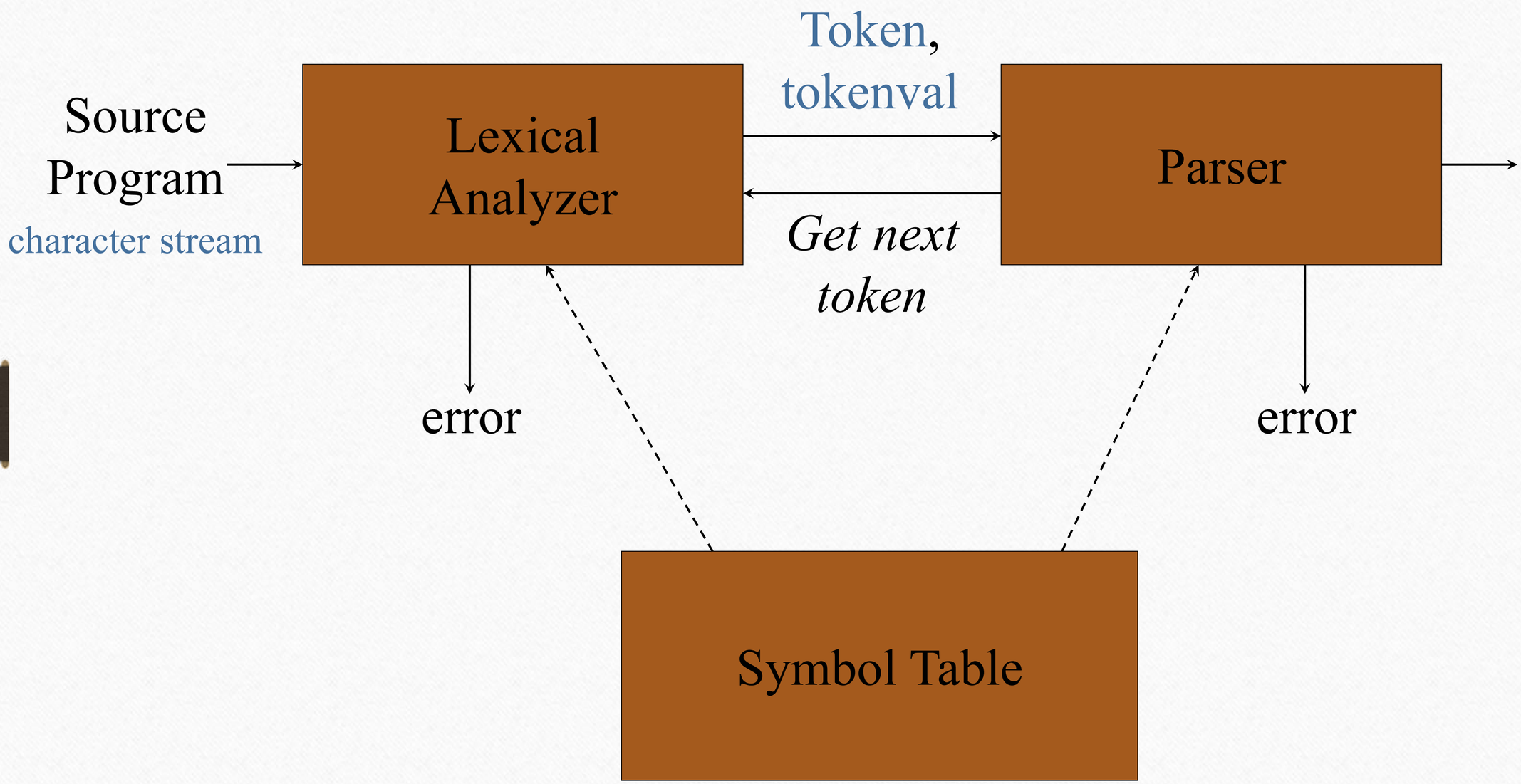
token `Tokenval`
(token attribute like **lexeme**)

<ID> <ASSIGN> <NUM,> <MUL> <ID> <MUL> <ID> <;> ~~<Comment, >~~ <IF,>

Symbol	Token	Data Type	Offset
area	ID		
radius	ID		

Syntax analyzer

Role – Reduce length by removing unnecessary tokens like comments, spaces etc.


Source Program
character stream
Lexical Analyzer
Token, tokenval
Get next token
Parser
error
error
Symbol Table

Tokens and Lexeme

- A ***token*** is a classification of lexical units
- ***Lexemes*** are the specific character strings that make up a token
- Identifier: x y11 elsex
- Keyword: if else while for break
- IntegerConstant: 2 1000 -20
- FloatConstant: 2.0 -0.0010 .02 1e5
- Symbol: + * { } ++ << < <= [ ]
- StringConstant: “x” “He said, \”I love EECS 483\””
- Comment: /* bla bla bla */

How to Describe Tokens

- ***Patterns*** are rules describing the set of lexemes belonging to a token
  - E.g. pattern for identifier : "letter followed by letters and digits"
- Use regular expressions to describe patterns of tokens of language of the source program

RE Notational Shorthand

- R* zero or more occurrence of R
- R+ one or more occurrence of R: R(R*)
- R? optional R: (R|ε)
- R{2,5} 2 to 5 occurances of R
- [abcd] one of **listed** characters: (a|b|c|d)
- [a-z] one character from this **range**: (a|b|c|d...|z)
- [^ab] anything **but** one of the listed chars
- [^a-z] one character not from this range
- ^R line starts with R
- R$ line ends with R
- . Any printable character except '\n'
- / Lookahead Operator

Self Evaluation

- A. Whats the difference?
  - [abcz] abcz
  - [abcz] [a|b|c |z]
  - [abcz] [a-cz]
  - [abcz] [^abcz]
- Extend the description of real on the previous slide to include numbers in scientific notation
  - -2.3E+17, -12.3e-17, -0.3E17, 2.31E+17

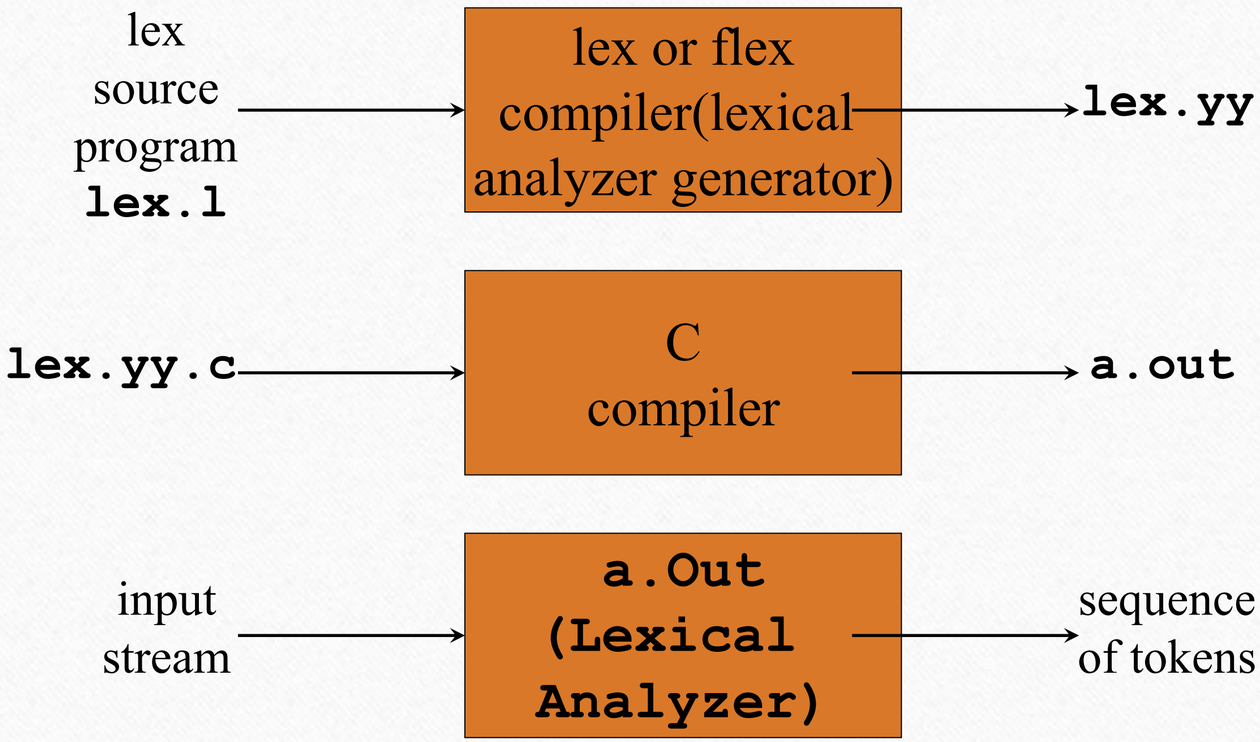
How to Break up Text

elsex = 0;

1	else	x	=	0	;
2	elsex	=	0	;	

- REs alone not enough, need rule for choosing when get multiple matches
  - Longest matching token wins
  - Ties in length resolved by priorities
    - Token specification order often defines priority
  - RE's + priorities + longest matching token rule = definition of a lexer

Automatic Generation of Lexer (Lexical Analyzer) using lexical analyzer generators i.e. lex/flex



Lex Specification

- A *lex specification* consists of three parts:

```
%{
        section 1 : regular definitions, C declarations in
%}
%%
section 2: translation rules
%%
section 3 : user-defined auxiliary procedures
```

- The *translation rules* are of the form:

p_1 { $action_1$ }
p_2 { $action_2$ }
…
p_n { $action_n$ }

Regular Expressions in Lex

x match the character **x**
\. match the character **.**
"*string*" match contents of string of characters
. match any character except newline
^ match beginning of a line
$ match the end of a line
[xyz] match one character **x**, **y**, or **z** (use **** to escape **-**)
[^xyz] match any character except **x**, **y**, and **z**
[a-z] match one of **a** to **z**
*r****** closure (match zero or more occurrences)
*r***+** positive closure (match one or more occurrences)
*r***?** optional (match zero or one occurrence)
$r_1 r_2$ match r_1 then r_2 (concatenation)
r_1**|**r_2 match r_1 or r_2 (union)
(*r* **)** grouping
r_1****r_2 match r_1 when followed by r_2
{*d***}** match the regular expression defined by *d*

Example Lex Specification 1

Translation rules

```
%{
#include <stdio.h>
%}
%%
[0-9]+   { printf("NUM\n"); }
\n       { }
.+/" "   {printf("%s invalid token\n", yytext);}
%%
main()
{ yylex();
}
```

Contains the matching lexeme

Invokes the lexical analyzer

```
lex spec.l
gcc lex.yy.c -ll
./a.out < test.c
```

Example Lex Specification 2

Translation rules

Regular definition

```
%{
#include <stdio.h>
int ch = 0, wd = 0, nl = 0;
%}
delim     [ \t]+
%%
\n        { ch++; wd++; nl++; }
^{delim}  { ch+=yyleng; }
{delim}   { ch+=yyleng; wd++; }
.         { ch++; }
%%
main()
{ yylex();
  printf("%8d%8d%8d\n", nl, wd, ch);
}
```

Example Lex Specification 3

Translation rules

Regular definitions

```
%{
#include <stdio.h>
%}
digit       [0-9]
letter      [A-Za-z]
id          {letter}({letter}|{digit})*
%%
{digit}+    { printf("number: %s\n", yytext); }
{id}        { printf("ident: %s\n", yytext); }
.           { printf("other: %s\n", yytext); }
%%
main()
{ yylex();
}
```

Example Lex Specification 4

```
%{ /* definitions of manifest constants */
#define LT (256)
…
%}
delim      [ \t\n]
ws         {delim}+
letter     [A-Za-z]
digit      [0-9]
id         {letter}({letter}|{digit})*
number     {digit}+(\.{digit}+)?(E[+\-]?{digit}+)?
%%
{ws}       { }
if         {return IF;}
then       {return THEN;}
else       {return ELSE;}
{id}       {yylval = install_id(); return ID;}
{number}   {yylval = install_num(); return NUMBER;}
"<"        {yylval = LT; return RELOP;}
"<="       {yylval = LE; return RELOP;}
"="        {yylval = EQ; return RELOP;}
">"        {yylval = GT; return RELOP;}
%%
int install_id()
…
```

Return token to parser

Token attribute

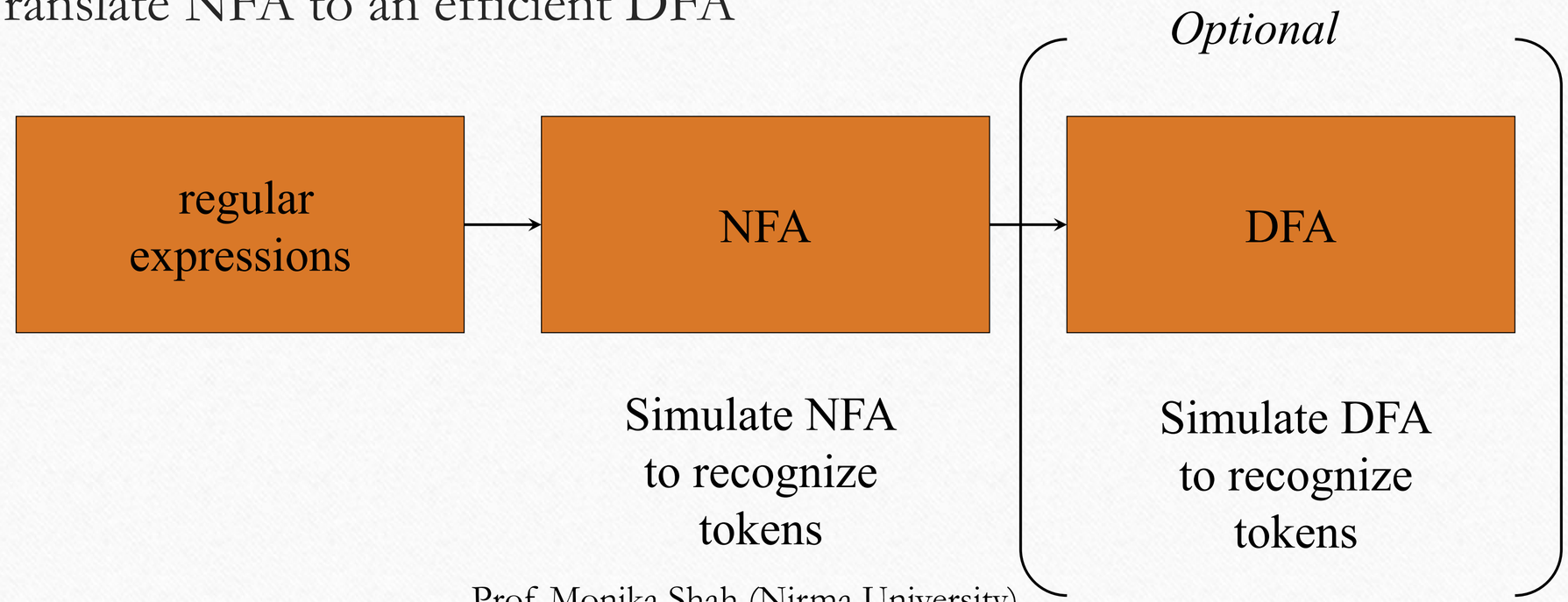
Install **yytext** as identifier in symbol table

Design of a Lexical Analyzer Generator

1. Approach to separate and recognize token using DFA
2. Input Buffering
3. Optimization using sentinel

Design of a Lexical Analyzer Generator

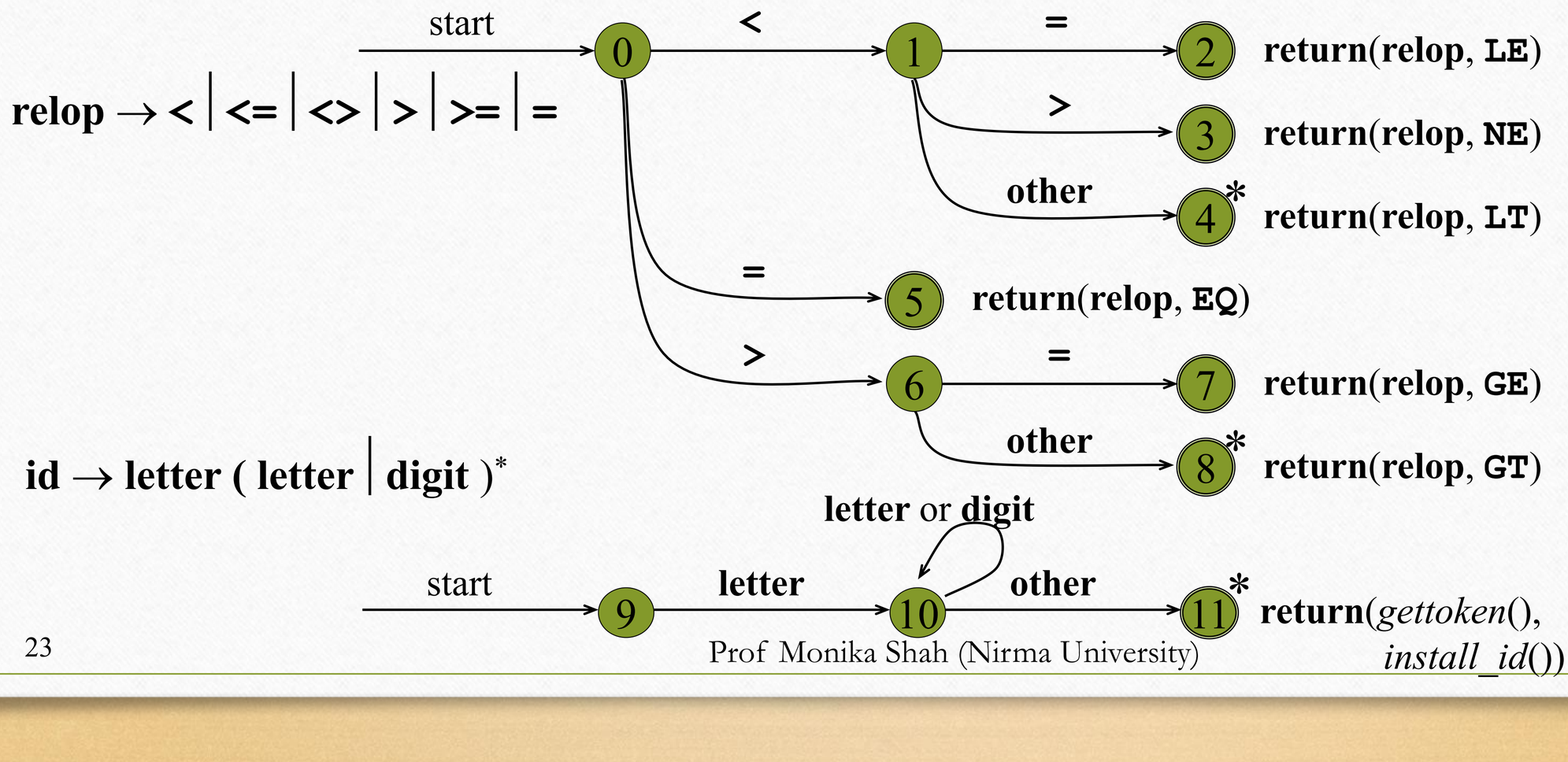
- Translate regular expressions to NFA
- Translate NFA to an efficient DFA



Time-Space Tradeoffs

Automaton	*Space (worst case)*	*Time (worst case)*
NFA	$O(\|r\|)$	$O(\|r\| \times \|x\|)$
DFA	$O(2^{\|r\|})$	$O(\|x\|)$

Coding Regular Definitions in *Transition Diagrams*



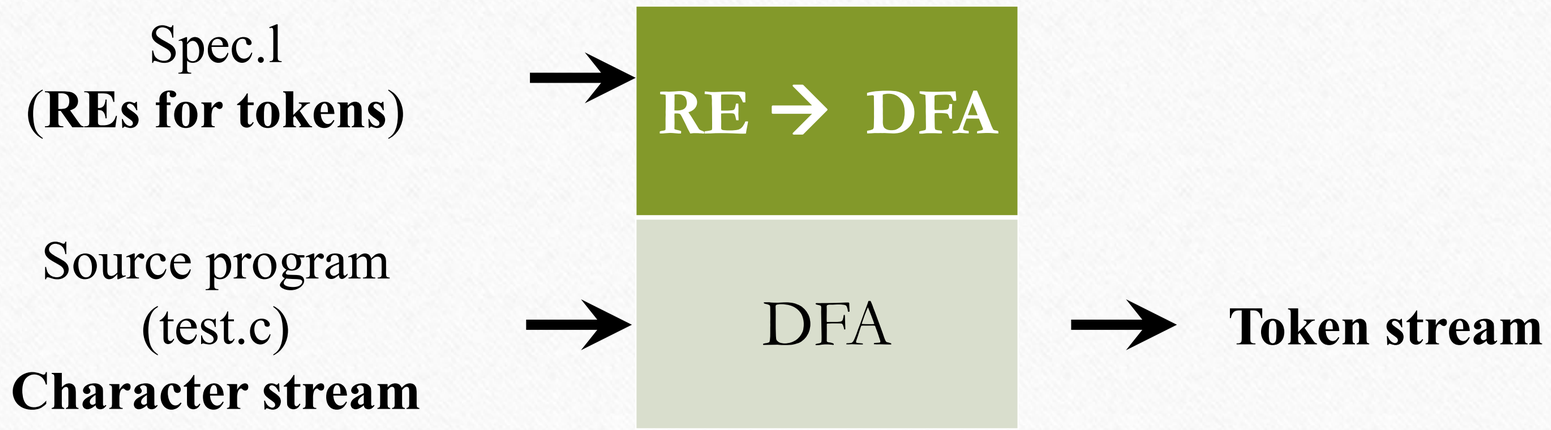
Coding Regular Definitions in Transition Diagrams: Code

```
token nexttoken()
{ while (1) {
    switch (state) {
    case 0: c = nextchar();
       if (c==blank || c==tab || c==newline) {
         state = 0;
         lexeme_beginning++;
       }
       else if (c=='<') state = 1;
       else if (c=='=') state = 5;
       else if (c=='>') state = 6;
       else state = fail();
       break;
    case 1:
       …
    case 9: c = nextchar();
       if (isletter(c)) state = 10;
       else state = fail();
       break;
    case 10: c = nextchar();
       if (isletter(c)) state = 10;
       else if (isdigit(c)) state = 10;
       else state = 11;
       break;
    …
```

Decides the next start state to check

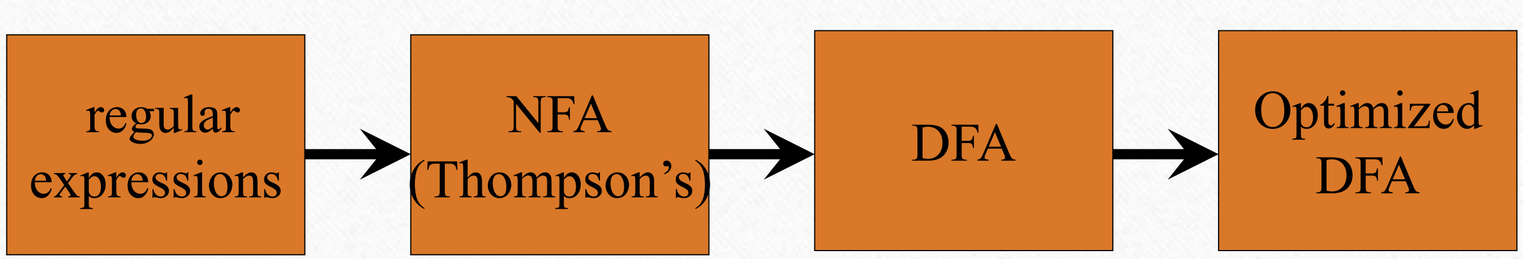
```
int fail()
{ forward = token_beginning;
  swith (start) {
  case  0: start =  9; break;
  case  9: start = 12; break;
  case 12: start = 20; break;
  case 20: start = 25; break;
  case 25: recover(); break;
  default: /* error */
  }
  return start;
}
```

How does Lexical Analyzer generator(i.e.Lex/Flex) work ?



1. Regular Expression to DFA

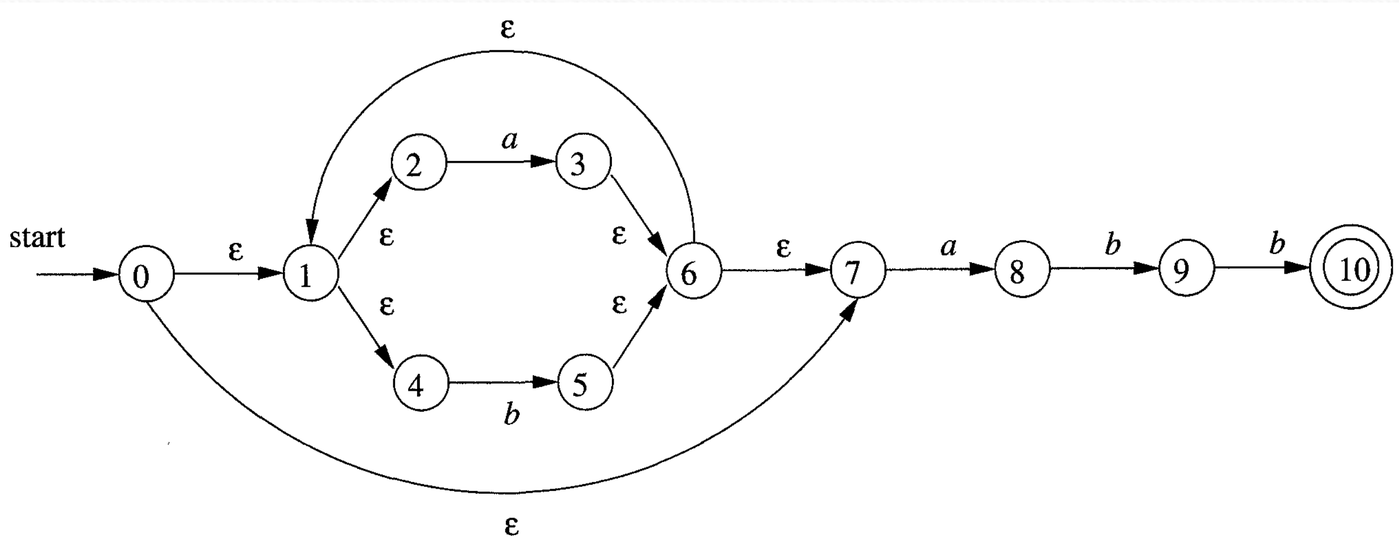
Approach 1



Approach 2

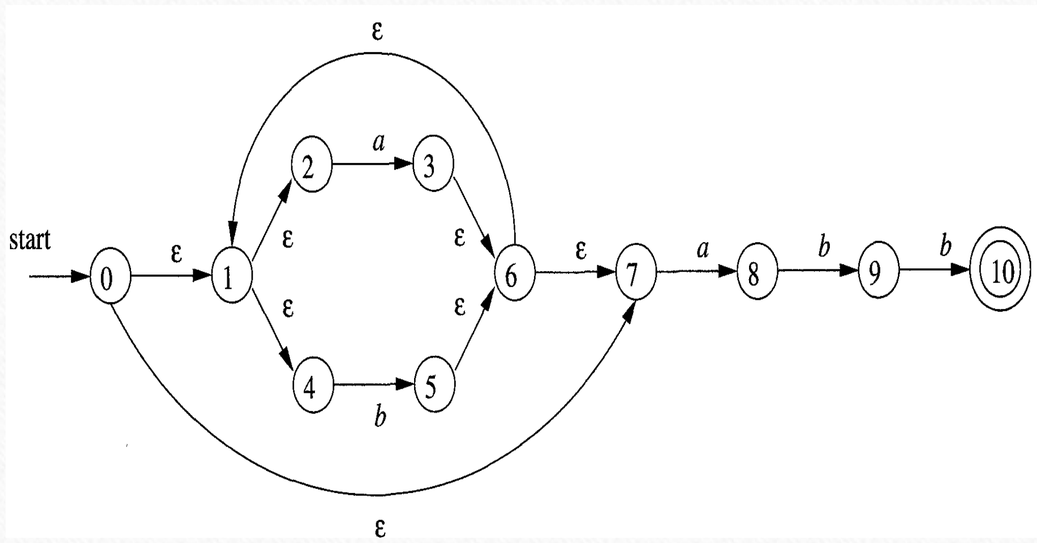
Approach 1: constructing optimized DFA from RE (RE->NFA->DFA->optimized DFA)

- Null NFA (using Thompson's construction) for NFA for (a/b)*abb



Approach 1: constructing optimized DFA from RE (RE->NFA->DFA->optimized DFA) contd…

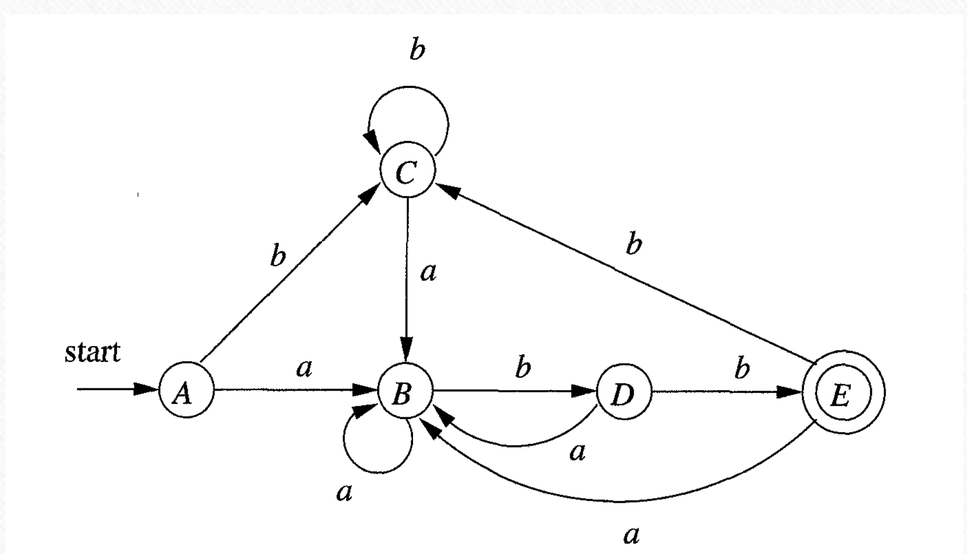
- NFA to DFA using Subset Construction Algorithm
- Start state = null-closure (0)={0,1,2,4,7}



State	a null-closure (State,a)	b null-closure (State,b)
A={0,1,2,4,7}	B	C
B={1,2,3,4,6,7,8}	B	D
C={1,2,4,5,6,7}	B	C
D={1,2,4,5,6,7,9}	B	E
E={1,2,4,5,6,7,10}	B	C

Approach 1: constructing optimized DFA from RE (RE->NFA->DFA->optimized DFA) contd…

- NFA to DFA using Subset Construction Algorithm
- Start state = null-closure (0)={0,1,2,4,7}



State	a null-closure (State,a)	b null-closure (State,b)
A={0,1,2,4,7}	B	C
B={1,2,3,4,6,7,8}	B	D
C={1,2,4,5,6,7}	B	C
D={1,2,4,5,6,7,9}	B	E
E={1,2,4,5,6,7,10}	B	C

Approach 1: constructing optimized DFA from RE (RE->NFA->DFA->optimized DFA) contd…

- Minimizing DFA

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

Given DFA

Non-final States

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

Final States

State	a	b
E	B	C

Remove redundant transitions

State	a	b
AC	B	AC
B	B	D
D	B	E

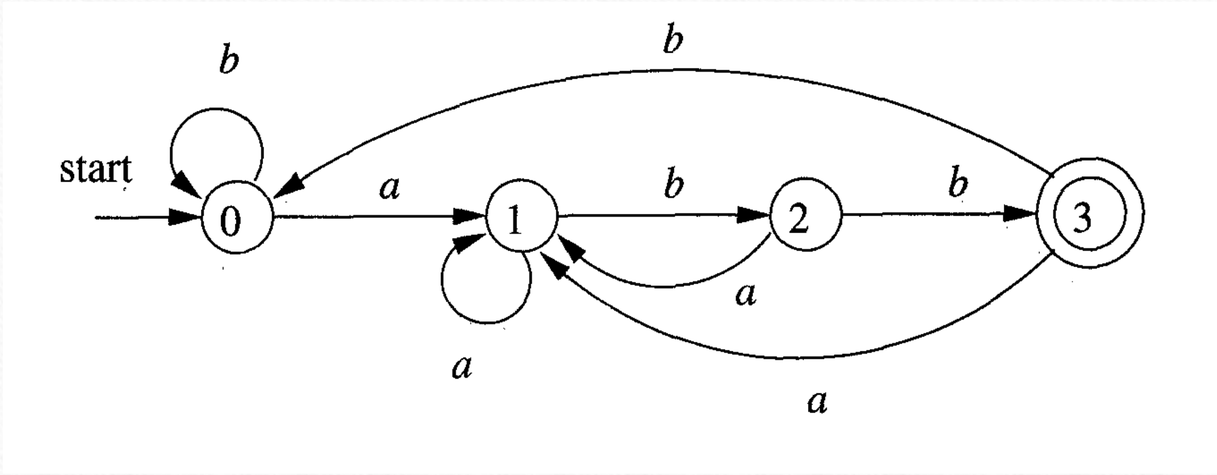
Final States

State	a	b
E	B	C

Optimized DFA

State	a	b
AC	B	AC
B	B	D
D	B	E
E	B	AC

Approach 1: constructing optimized DFA from RE (RE->NFA->DFA->optimized DFA) contd…



Optimized DFA

State	a	b
AC (0)	B	AC
B (1)	B	D
D (2)	B	E
E (3)	B	AC

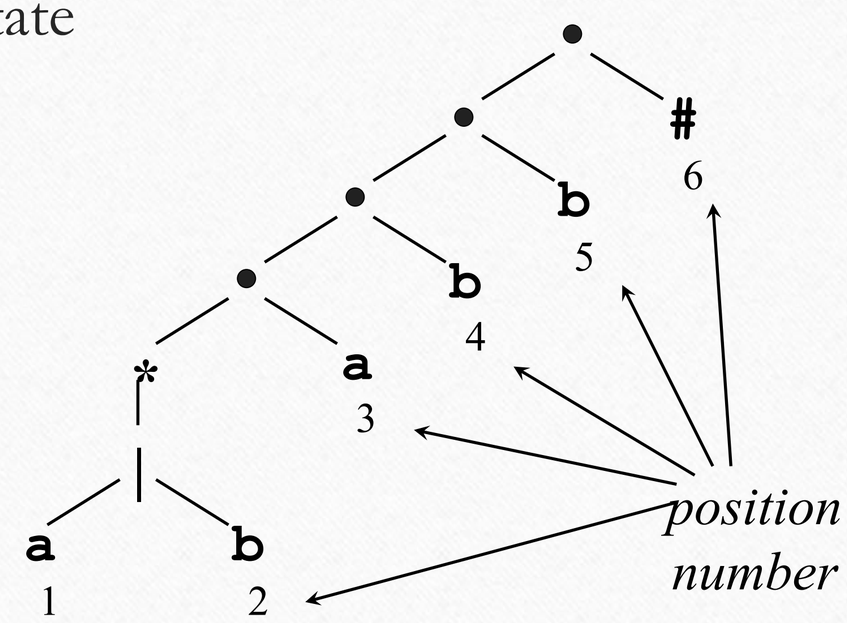
RE to DFA direct approach

1. Augment RE r with # to make accepting state

e.g. r = (a/b)*abb ➔ (a/b)*abb**#**

2. Make Syntax tree

3. Number each input symbol including #



From Regular Expression to DFA Directly: Annotating the Tree

- *nullable*(*n*): the subtree at node *n* generates languages including the empty string
- *firstpos*(*n*): set of positions that can match the first symbol of a string generated by the subtree at node *n*
- *lastpos*(*n*): the set of positions that can match the last symbol of a string generated be the subtree at node *n*
- *followpos*(*i*): the set of positions that can follow position *i* in the tree

From Regular Expression to DFA Directly: Annotating the Tree

Node n	*nullable*(n)	*firstpos*(n)	*lastpos*(n)
Leaf ε	true	∅	∅
Leaf i	false	$\{i\}$	$\{i\}$
	*nullable*(c_1) or *nullable*(c_2)	*firstpos*(c_1) ∪ *firstpos*(c_2)	*lastpos*(c_1) ∪ *lastpos*(c_2)
	*nullable*(c_1) and *nullable*(c_2)	**if** *nullable*(c_1) **then** *firstpos*(c_1) ∪ *firstpos*(c_2) **else** *firstpos*(c_1)	**if** *nullable*(c_2) **then** *lastpos*(c_1) ∪ *lastpos*(c_2) **else** *lastpos*(c_2)
	true	*firstpos*(c_1)	*lastpos*(c_1)

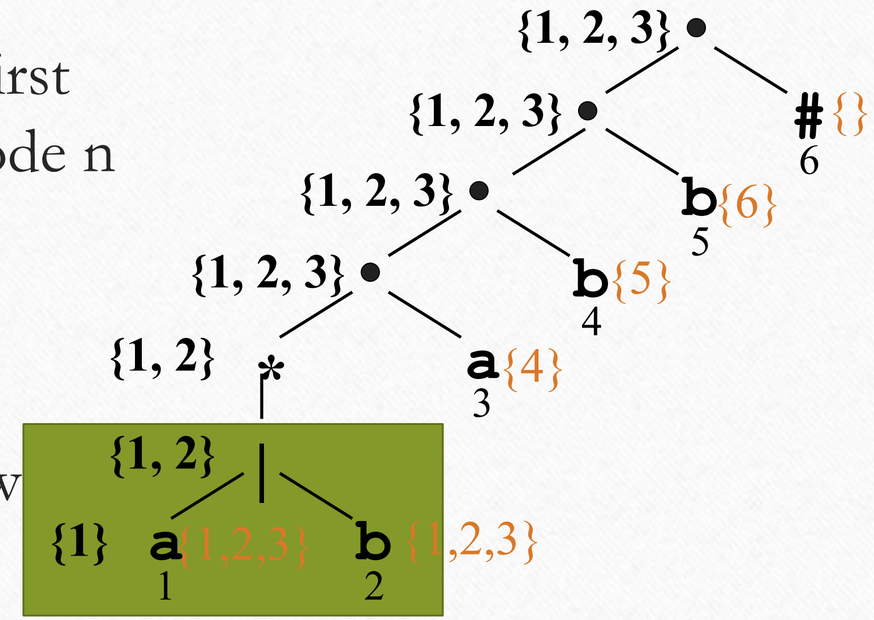
RE to DFA direct approach contd…

4. Find first position at each node

firstpos(n) : set of position that match the first symbol of a string generated by subtree at node n

5. Find follow position at each leaf node

followpos(n) : set of position that can follow node n at tree

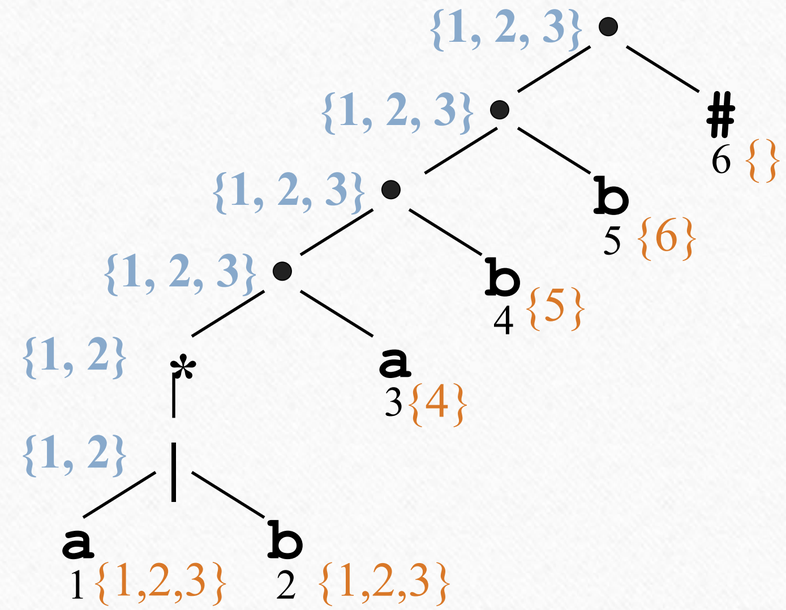


firstpos *followpos*

RE to DFA direct approach contd…

6. Make firstpos of last node join as start state➔ Start state = A = {1,2,3}
7. Make Transition table (state x input symbols)

State	a	b
A = {1, 2, 3}	= {1, 2, 3} U {4} = {1, 2, 3, 4} =B	= {1, 2, 3} = A
B = {1, 2, 3, 4}	= {1, 2, 3, 4} =B	= {1, 2, 3} U {5} = {1, 2, 3, 5} =C
C = {1, 2, 3, 5}	= {1, 2, 3, 4} =B	= {1, 2, 3} U {6} = {1, 2, 3, 6} =D
D = {1, 2, 3, 6}	= {1, 2, 3, 4} =B	= {1, 2, 3} =A

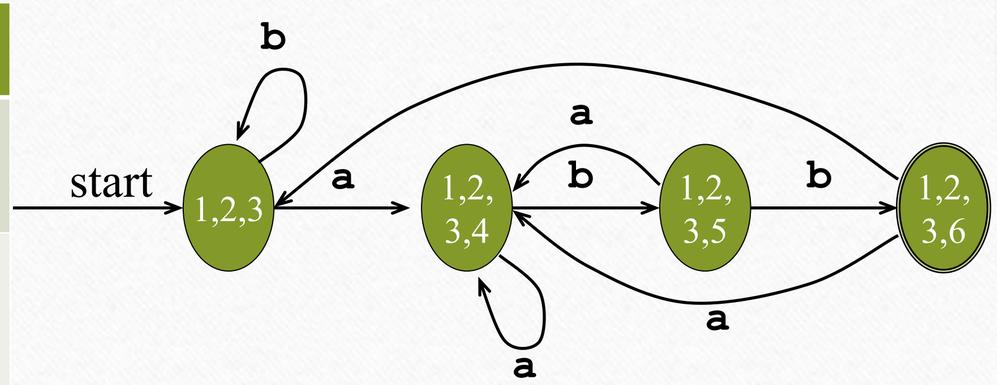


firstpos *followpos*

RE to DFA direct approach contd…

4. Make firstpos of last node join as start state➔ Start state = A = {1,2,3}
5. Make Transition table (state x input symbols)

State	a	b
A = {1, 2, 3}	= {1, 2, 3} U {4} = {1, 2, 3, 4} =B	= {1, 2, 3} = A
B = {1, 2, 3, 4}	= {1, 2, 3, 4} =B	= {1, 2, 3} U {5} = {1, 2, 3, 5} =C
C = {1, 2, 3, 5}	= {1, 2, 3, 4} =B	= {1, 2, 3} U {6} = {1, 2, 3, 6} =D
D = {1, 2, 3, 6}	= {1, 2, 3, 4} =B	= {1, 2, 3} =A



2. Input Buffering
single input buffer

- Significance of Input Buffer
  - A Scanner in Lexical Analyzer reads the input character by character
  - Request to OS for every character read is inefficient due to context switch

➔ Scanner use its own Input Buffer

forward

2. Input Buffering
single input buffer

Source file

f	l	o	a	t		p	i	=	3	.	1	4	;	\n	f	l	o	a	t			a	r	e	a	,	r	a	d	i	u...
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30		

1	2	3	4	5	6	7	8	9	10	11	12
4	;	\n	f	l	o	a	t		a	r	e

char buf[12]

Lexeme beginning

forward

<Data Type, float >

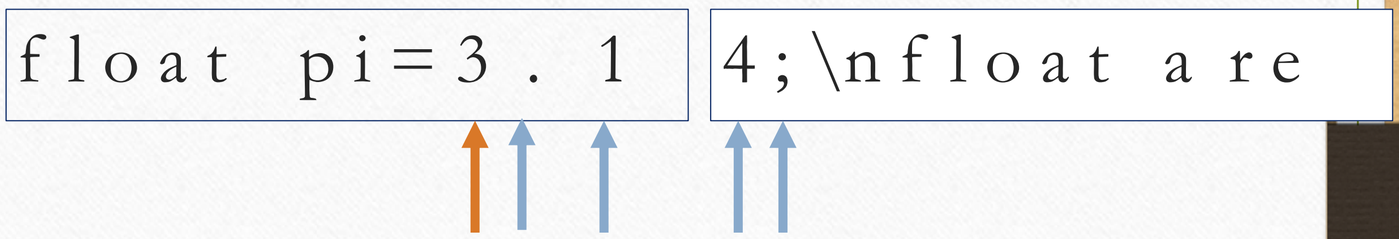
<ID, pi >

<ASSIGN, >

<FNUM, are4 > X

2. Input Buffering
double input buffer

- Concept
  - forwardP = forwardP +1
  - if (forwardP = End of 1st Buffer)
    - Load to 2nd Buffer
    - forwardP = forwardP +1 // beginning of second buffer
  - Else if (forwardP = end of 2nd Buffer)
    - Load to 1st Buffer
    - forward = 1 // beginning of first half
  - Else
    - Forward + +



<FNUM, 3.14 > √

3. Algorithm optimization using sentinel

- **Basic Concept**
  - ForwardP ++
  - if (forwardP = End of 1st Buffer)
    - Load to 2nd Buffer, forwadP = forwardP + 1
  - Else if (forwardP = end of 2nd Buffer)
    - Load to 1st Buffer, forwardP = 1
  - Else if (forwardP = end of file)
    - Terminate scan, Match pattern
- Time complexity = T(4n)

- **Concept using sentinel**
  - forwardP ++
  - if (forwardP = sentinel)
    - If (forwardP = End of 1st Buffer)
      - Load to 2nd Buffer, forwadP = forwardP +1
    - Else if (forwardP = End of 2nd Buffer)
      - Load to 1st Buffer, forwardP = 1
    - Else //if (forwardP = End of File)
      - Terminate scan, Match pattern
- **Time complexity = T(2n + 3*n/BufSize)**

Self Evaluation

- Lexical analyzer vs lexical analyzer generator
- Why DFA is preferred for implementation ?
- What is advantage of direct RE to DFA ?
- Why two buffers required to load input program text in source language ?
- Time complexity of code simulation with and without sentinel

Self Evaluation

- Covert optimized DFA for following RE using direct approach
  - (a/b)* a (a/b) (a/b)
  - 0 (0/1)* 0
  - (0/1)* 1 0 * 1 0 *
- Write regular expression to recognize "**if**" as keyword only if it is followed by "**(**"

  Hint : lookahead operator
- What is disadvantage of single input buffer in scanning ?
- What is advantage of sentinel?