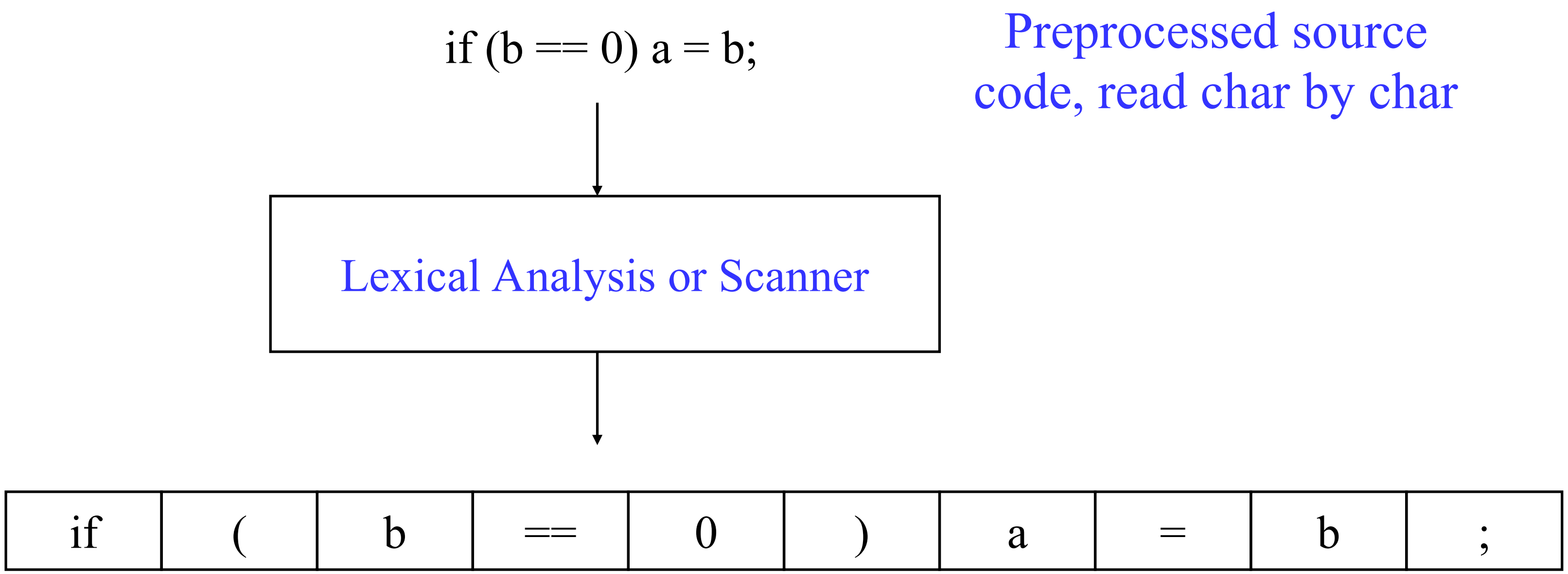
The Lexical Analysis

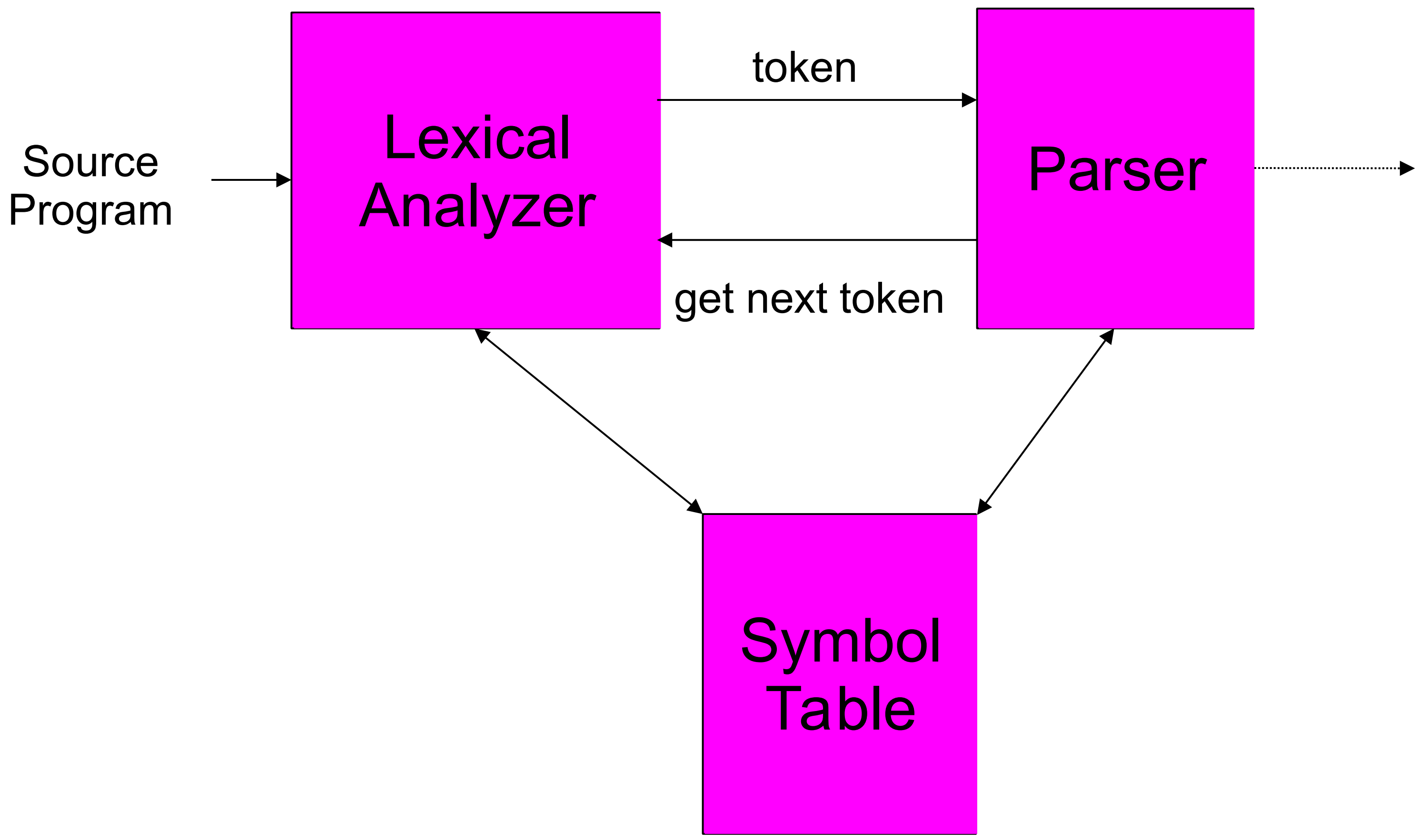
Lexical Analysis Process



Lexical analysis

- Transform multi-character input stream to token stream
- Reduce length of program representation (remove spaces)

Lexical Analyzer and Its Role in A Compiler



Tokens

- Identifiers: x y11 elsex
- Keywords: if else while for break
- Integers: 2 1000 -20
- Floating-point: 2.0   -0.0010 .02 1e5
- Symbols: + * { } ++ << < <= [ ]
- Strings: “x” “He said, \”I luv CC\””

How to Describe Tokens

- Use regular expressions to describe programming language tokens!
- A regular expression (RE) is defined inductively
  - a ordinary character stands for itself
  - $\in$ empty symbol
  - R|S either R or S (alteration), where R,S = RE
  - RS R followed by S (concatenation)
  - R* concatenation of R, 0 or more times (Kleene closure)

Language

- A regular expression R describes a set of strings of characters denoted L(R)
- L(R) = the language defined by R
  - L(abc) = { abc }
  - L(hello|goodbye) = { hello, goodbye }
  - L(1(0|1)*) = all binary numbers that start with a 1
- Each token can be defined using a regular expression

RE Notational Shorthand

- R^+ one or more strings of R: $R(R^*)$
- R? optional R: $(R|\in)$
- [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

Lexical and Syntax Analysis

stmt → **if** *expr* **then** *stmt*
| **if** *expr* **then** *stmt* **else** *stmt*
| ∈
expr → *term* **relop** *term*
| *term*
term → **id**
| **num**

if → `if`
then → `then`
else → `else`
relop → **< | <= | = | <> | > | >=**
id → letter (letter | digit)*
num → digit$^+$(.digit$^+$)?(E(+|-)? digit$^+$)?

How to Break up Text

elsex = 0;

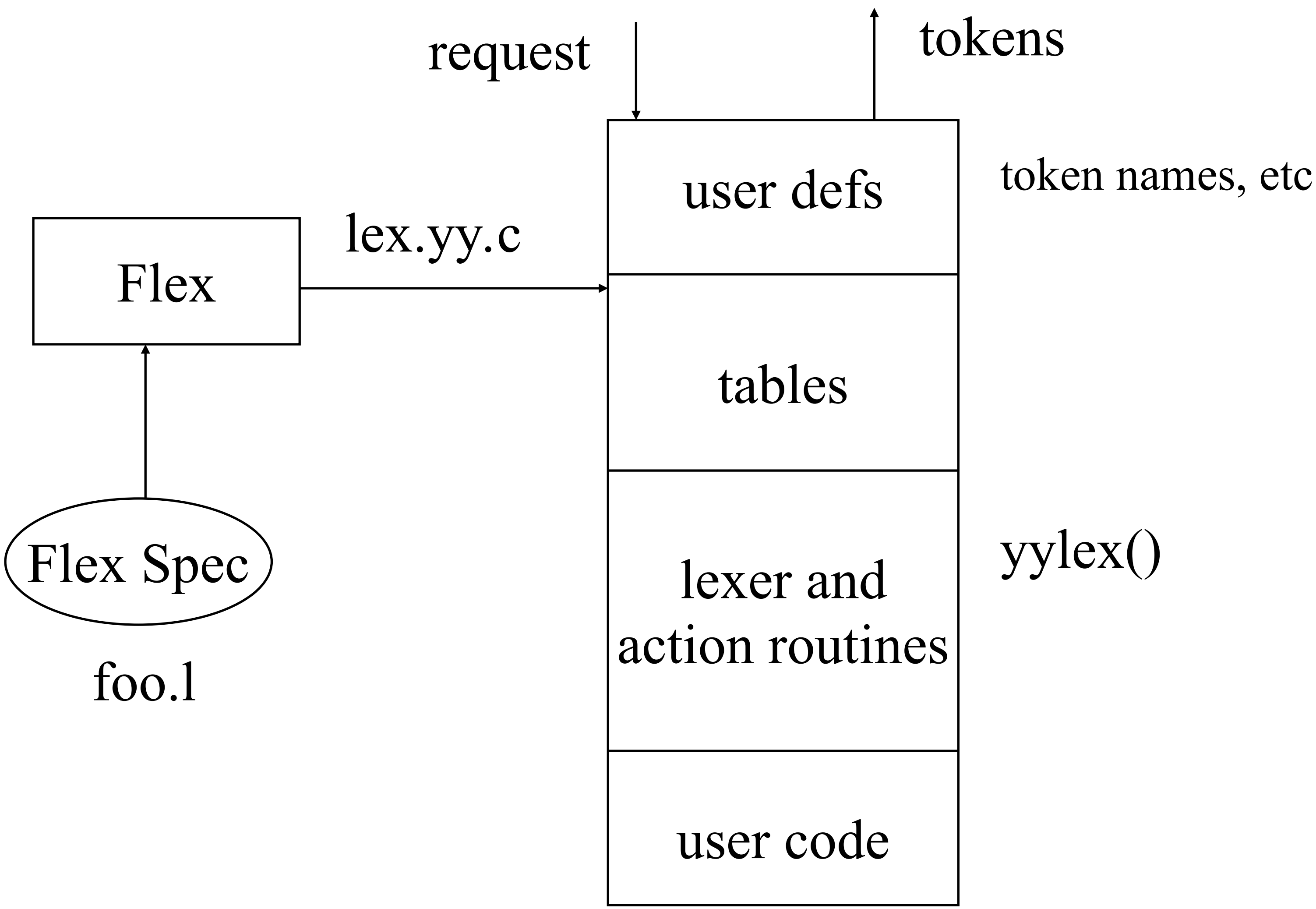
1	else	x	=	0	;
2	elsex	=	0	;	

- 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 Lexers

- 2 programs developed at Bell Labs in mid 70's for use with UNIX
  - Lex – transducer, transforms an input stream into the alphabet of the grammar processed by yacc
  - Written by Mike E. Lesk
    - Flex = fast lex, later developed by Free Software Foundation
  - Yacc/bison – yet another compiler/compiler (next lecture)
- Input to lexer generator
  - List of regular expressions in priority order
  - Associated action with each RE
- Output
  - Program that reads input stream and breaks it up into tokens according the the REs

Lex/Flex


request
tokens
user defs
token names, etc
lex.yy.c
Flex
tables
Flex Spec
lexer and
action routines
yylex()
foo.l
user code

Lex Specification

- Definition section
  - All code contained within "%{" and "%}" is copied to the resultant program. Usually has token defns established by the parser
  - User can provide names for complex patterns used in rules
  - Any additional lexing states (states prefaced by %s directive)
  - Pattern and state definitions must start in column 1 (All lines with a blank in column 1 are copied to resulting C file)

lex file always has 3 sections:

definition section

%%

rules section

%%

user functions section

Lex Specification

- Rules section
  - Contains lexical patterns and semantic actions to be performed upon a pattern match. Actions should be surrounded by {} (though not always necessary)
  - Again, all lines with a blank in column 1 are copied to the resulting C program
- User function section
  - All lines in this section are copied to the final .c file
  - Unless the functions are very immediate support routines, better to put these in a separate file

Partial Flex Program

```
D           [0-9]
%%
if          printf ("IF statement\n");
[a-z]+      printf ("ID, value %s\n", yytext);
{D}+        printf ("decimal number %s\n", yytext);
"++"        printf ("incrementation op\n");
"+"         printf ("addition op\n");
```

pattern

action

Note: **`yytext`** is a pointer to first char of the token
`yyleng` = length of token

Flex Program

```
%{
       #include <stdio.h>
       int num_lines = 0, num_chars = 0;
%}

%%
\n        ++num_lines; ++num_chars;
.         ++num_chars;

%%
int main()
{
 yylex();
 printf( "# of lines = %d, # of chars = %d \n",num_lines, num_chars );
}
```

- Running the above program:

```
neo$ flex count.l
neo$ gcc lex.yy.c -lfl
neo$ a.out < count.l
# of lines = 16, # of chars = 221
```

Lex Program for A Lexer in a Compiler

```
%{
          /* definitions of manifest constants
          LT, LE, EQ, NE, GT,
          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 = install_id(); return(ID);}
{number}  {yylval = install_num(); return(NUMBER);}
"<"       {yylval = LT; return(RELOP);}
"<="      {yylval = LE; return(RELOP);}
"="       {yylval = EQ; return(RELOP);}
"<>"      {yylval = NE; return(RELOP);}
">"       {yylval = GT; return(RELOP);}
">="      {yylval = GE; return(RELOP);}

%%

int install_id() {
          /* procedure to install the lexeme, whose first character is pointed by yytext
          and whose length is yyleng, into the symbol table and return an index thereof */
}

int install_num() {
          /* similar procedure to install a lexeme that is a number */
}
```

Lex Regular Expression Meta Chars

Meta Char	Meaning
.	match any single char (except \n)
*	Kleene closure (0 or more)
[]	Match any character within brackets
	- in first position matches -
	^ in first position inverts set
^	matches beginning of line
$	matches end of line
{a,b}	match count of preceding pattern
	from a to b times, b optional
\	escape for metacharacters
+	positive closure (1 or more)
?	matches 0 or 1 REs
|	alteration
/	provides lookahead
()	grouping of RE
<>	restricts pattern to matching only in that state

How Does Lex Work?

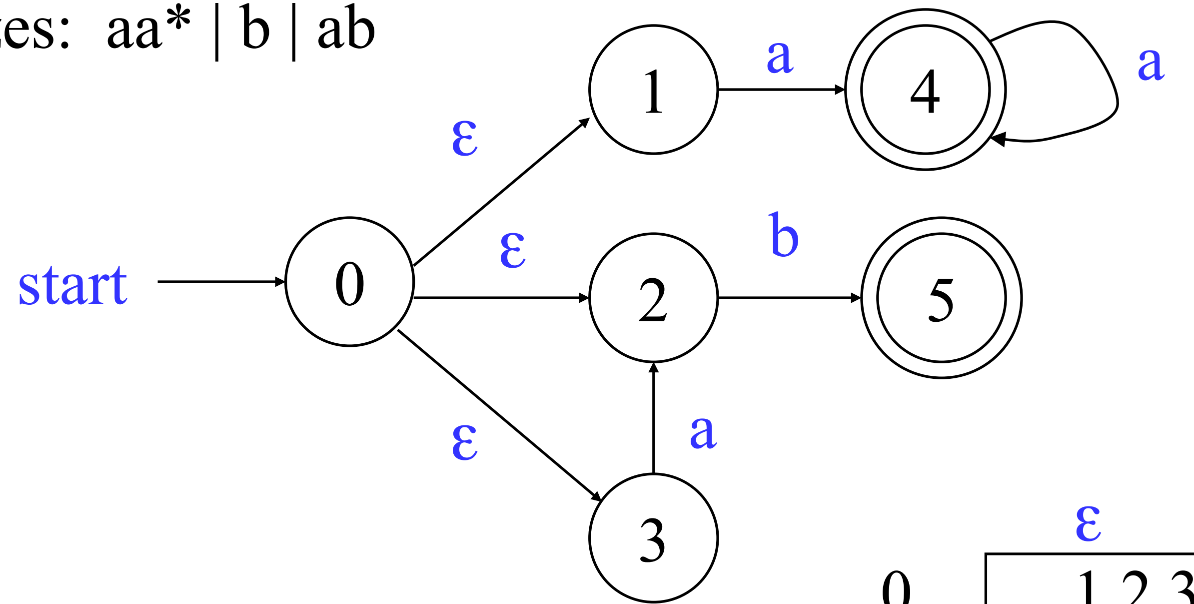
- Formal basis for lexical analysis is the finite state automaton (FSA)
  - REs generate regular sets
  - FSAs recognize regular sets
- FSA – informal defn:
  - A finite set of states
  - Transitions between states
  - An initial state (start)
  - A set of final states (accepting states)

Two Kinds of FSA

- Non-deterministic finite automata (NFA)
  - There may be multiple possible transitions or some transitions that do not require an input (ε)
- Deterministic finite automata (DFA)
  - The transition from each state is uniquely determined by the current input character
    - For each state, at most 1 edge labeled ‘a’ leaving state
  - No ε transitions

NFA Example

Recognizes: aa* | b | ab

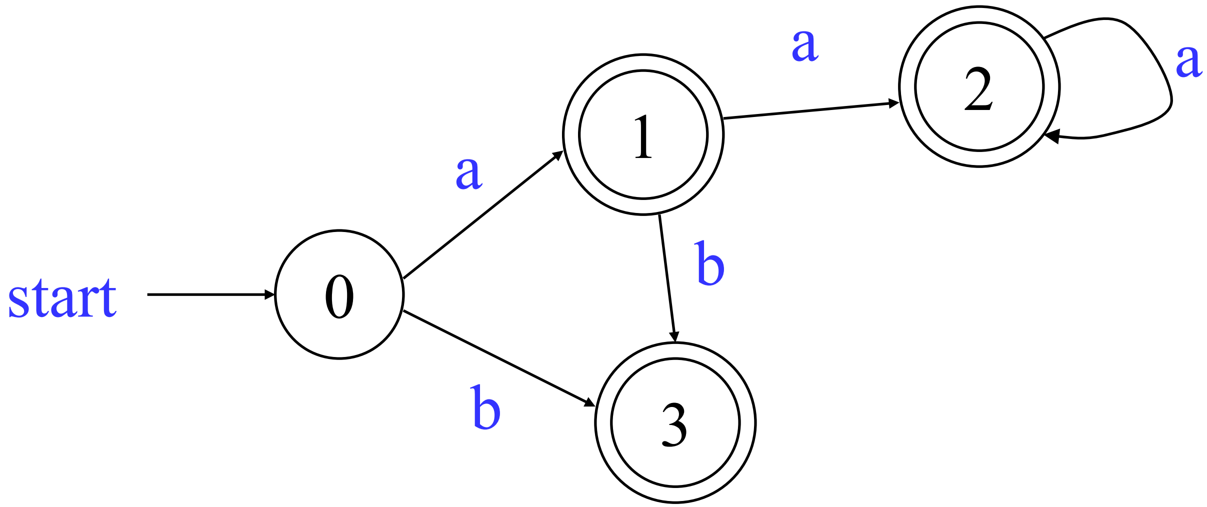


Can represent FA with either
graph or transition table

	ε	a	b
0	1,2,3	-	-
1	-	4	-
2	-	-	5
3	-	2	-
4	-	4	-
5	-	-	-

DFA Example

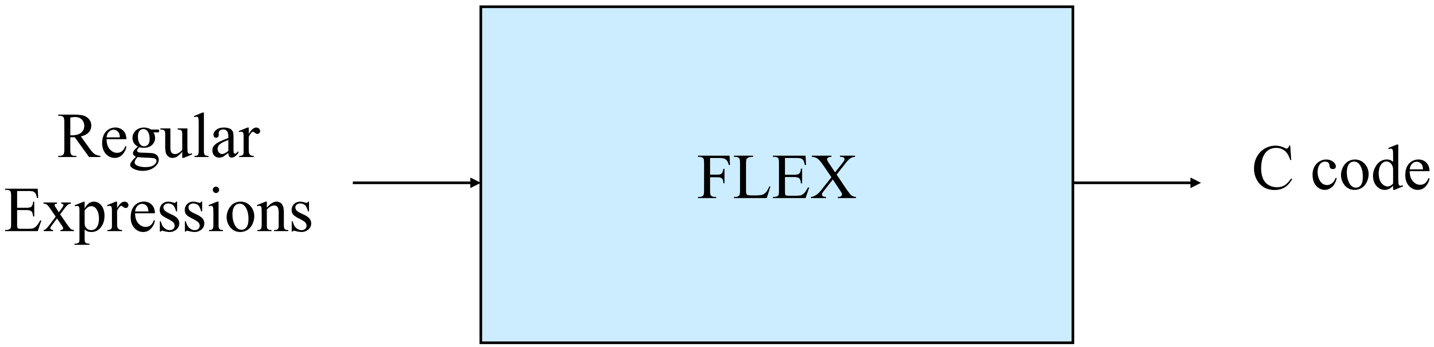
Recognizes: aa* | b | ab



NFA vs DFA

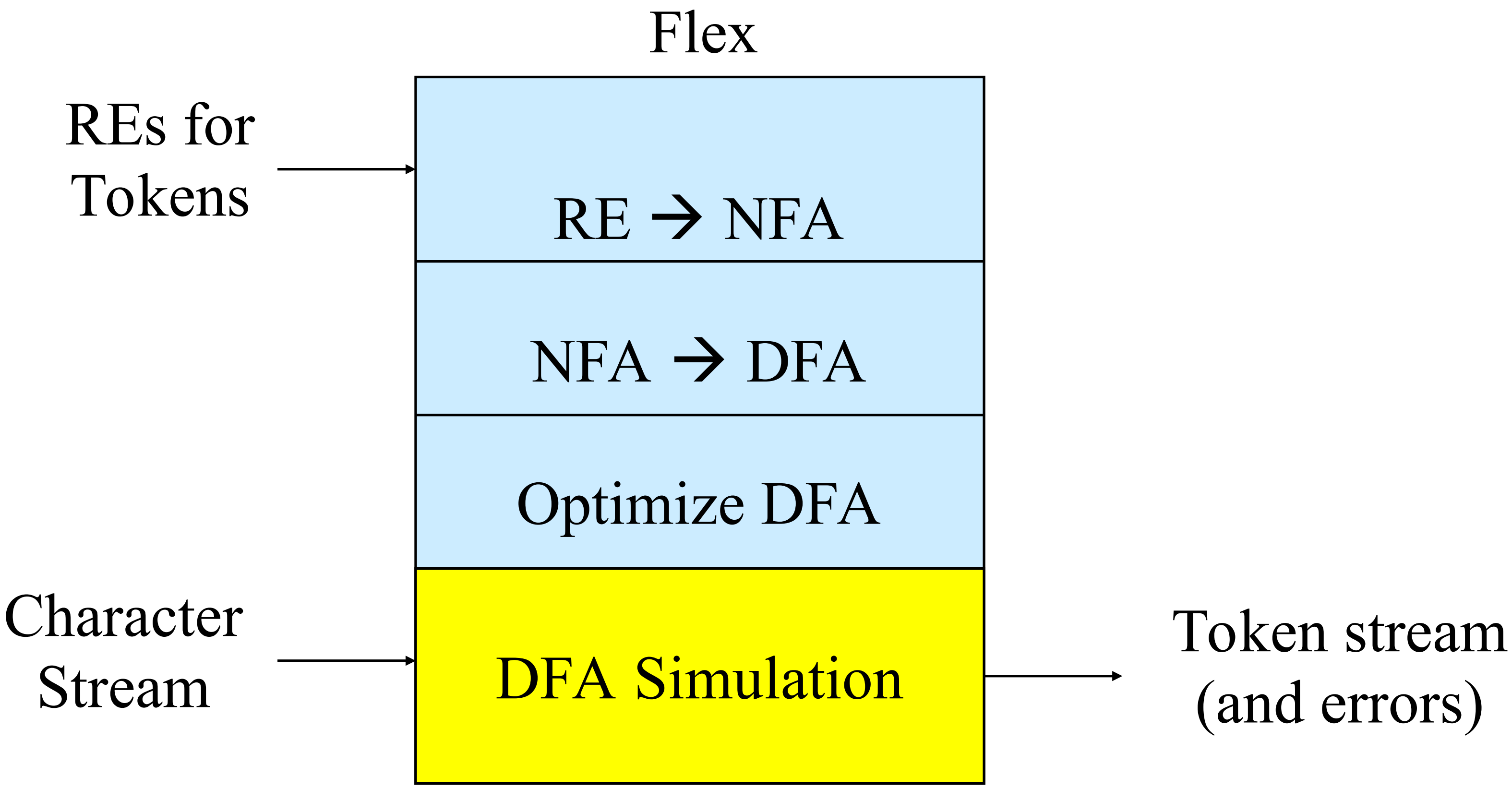
- DFA
  – Action on each input is fully determined
  – Implement using table-driven approach
  – More states generally required to implement RE
- NFA
  – May have choice at each step
  – Accepts string if there is ANY path to an accepting state
  – Not obvious how to implement this

How Does Lex Work?



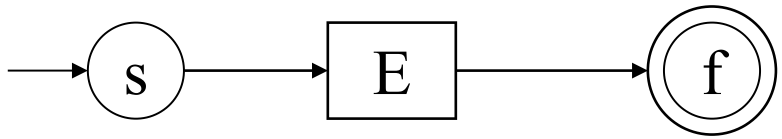
Some kind of DFAs and NFAs
stuff going on inside

How Does Lex Work?



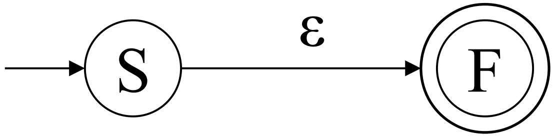
Regular Expression to NFA

- Its possible to construct an NFA from a regular expression
- Thompson's construction algorithm
  – Build the NFA inductively
  – Define rules for each base RE
  – Combine for more complex RE's

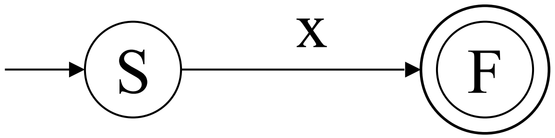


general machine

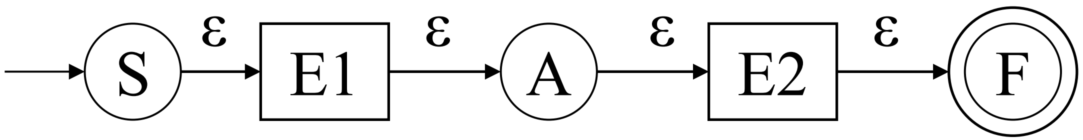
Thompson Construction



empty string transition



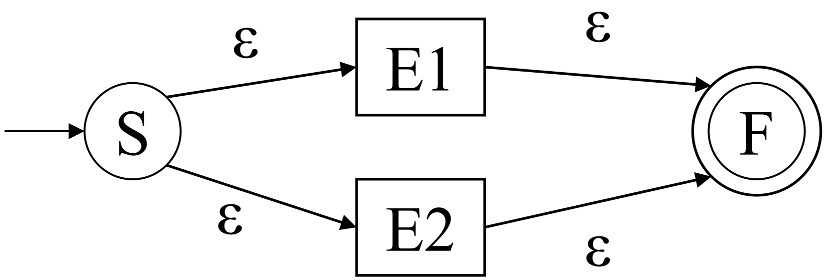
alphabet symbol transition



Concatenation:
(E1 E2)

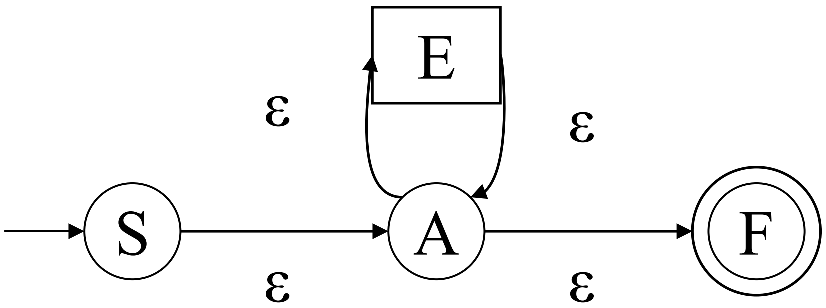
- New start state S ε-transition to the start state of E1
- ε-transition from final/accepting state of E1 to A, ε-transition from A to start state of E2
- ε-transitions from the final/accepting state E2 to the new final state F

Thompson Construction



Alteration: (E1 | E2)

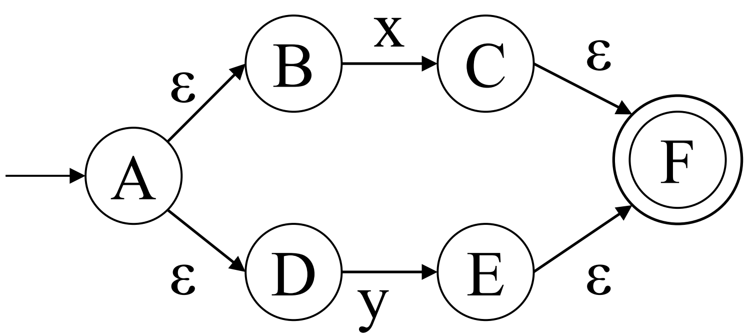
- New start state S ε-transitions to the start states of E1 and E2
- ε-transitions from the final/accepting states of E1 and E2 to the new final state F



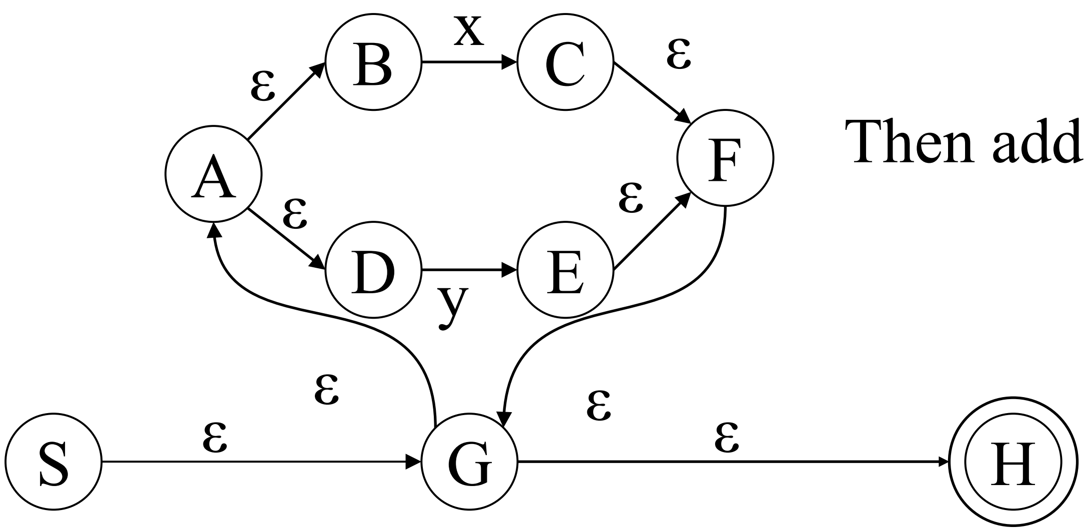
Closure: (E*)

Thompson Construction - Example

Develop an NFA for the RE: (x | y)*



First create NFA for (x | y)



Then add the closure operator

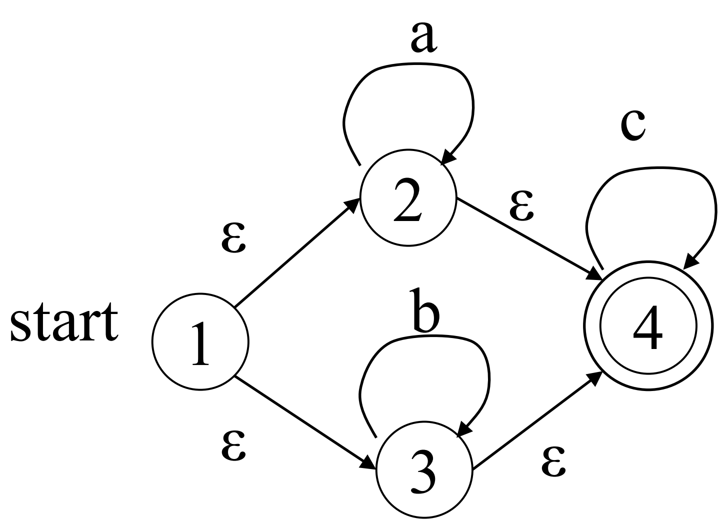
Class Problem

Develop an NFA for the RE: (\+? | -?) d+

NFA to DFA

- Remove the non-determinism
- 2 problems
  - States with multiple outgoing edges due to same input
  - ε transitions

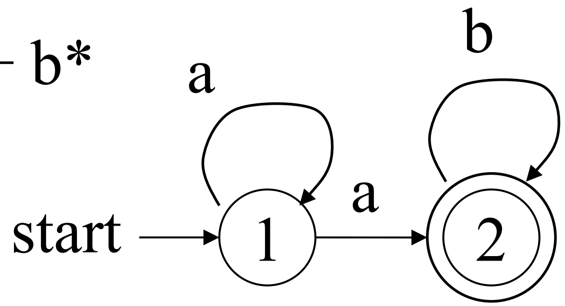
(a*| b*) c*

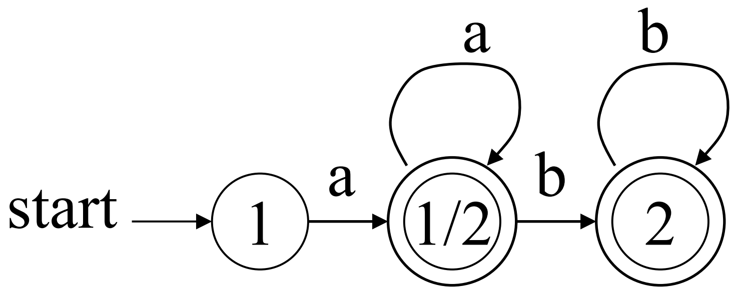


NFA to DFA

- Problem 1: Multiple transitions
  - Solve by subset construction
  - Build new DFA based upon the power set of states on the NFA
  - Move (S,a) is relabeled to target a new state whenever single input goes to multiple states

a+ b*





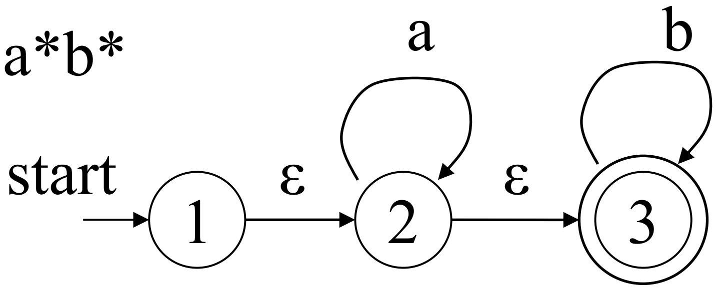
(1,a) → 1 or 2, create new state 1/2
(1/2,a) →1/2
(1/2,b) → 2

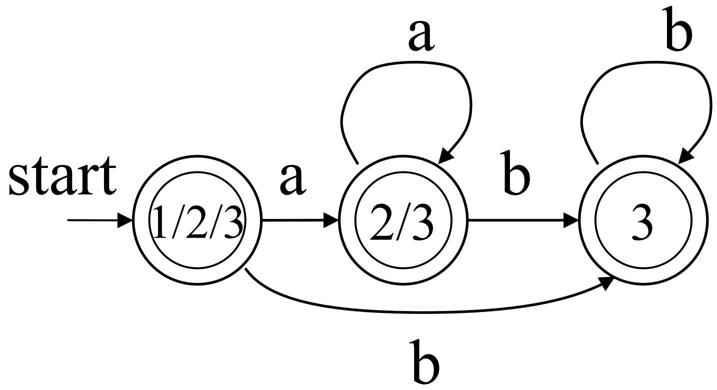
(2,a) → ERROR
(2,b) → 2
Any state with "2" in name is a final state

NFA to DFA

- Problem 2: ε transitions
  - Any state reachable by an ε transition is "part of the state"
  - ε-closure - Any state reachable from S by ε transitions is in the ε-closure; treat ε-closure as 1 big state, always include ε-closure as part of the state

a*b*

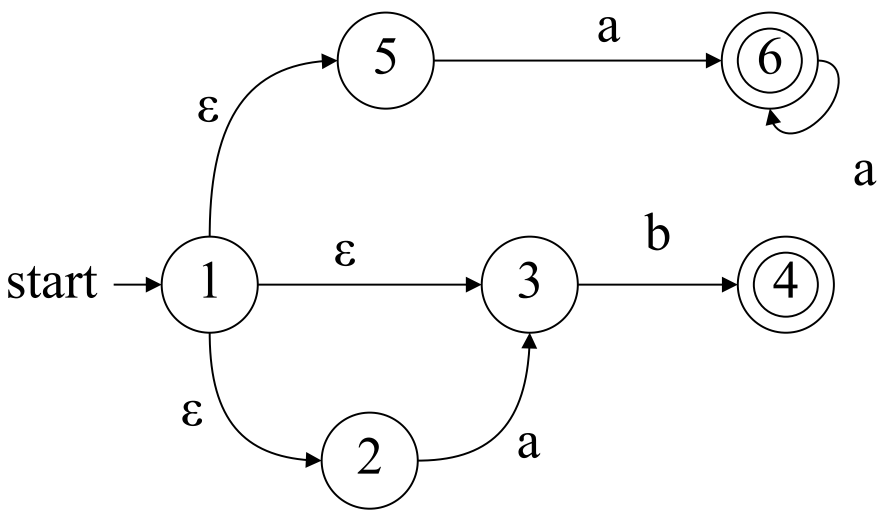


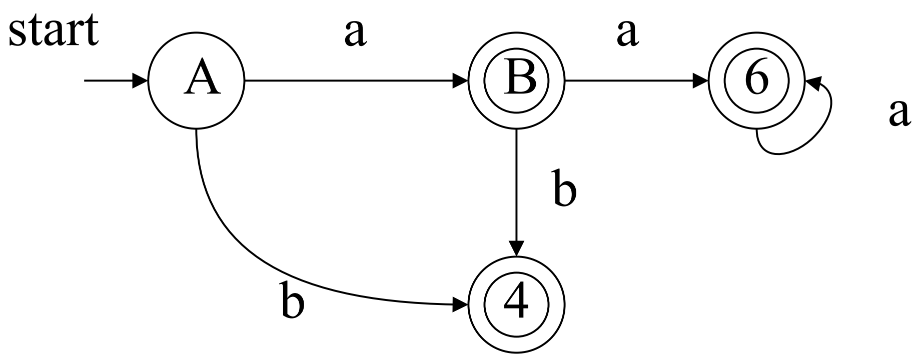


ε-closure(1) = {1,2,3}
ε-closure(2) = {2,3}
create new state 1/2/3
create new state 2/3

(1/2/3, a) → 2/3
(1/2/3, b) → 3
(2/3, a) → 2/3
(2/3, b) → 3

NFA to DFA - Example

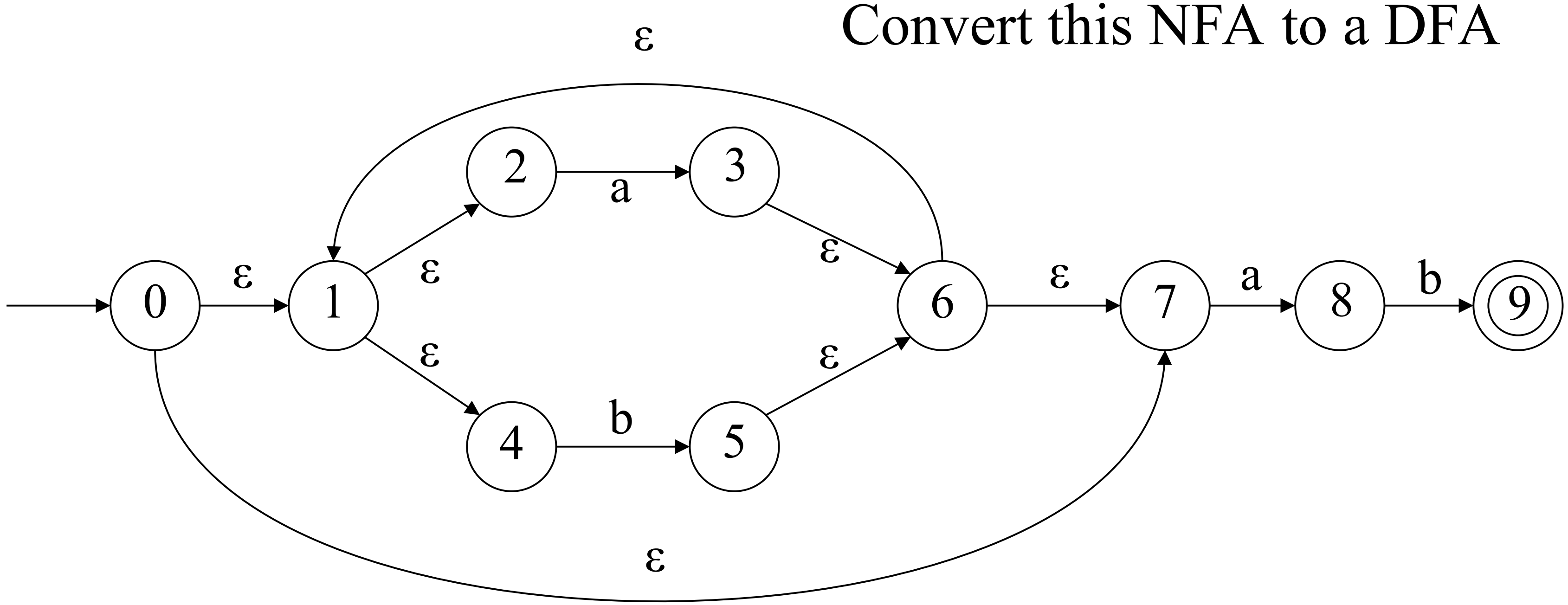




- ε-closure(1) = {1, 2, 3, 5}
- Create a new state A = {1, 2, 3, 5} and examine transitions out of it
- move(A, a) = {3, 6}
- Call this a new subset state = B = {3, 6}
- move(A, b) = {4}
- move(B, a) = {6}
- move(B, b) = {4}
- Complete by checking move(4, a); move(4, b); move(6, a); move(6, b)

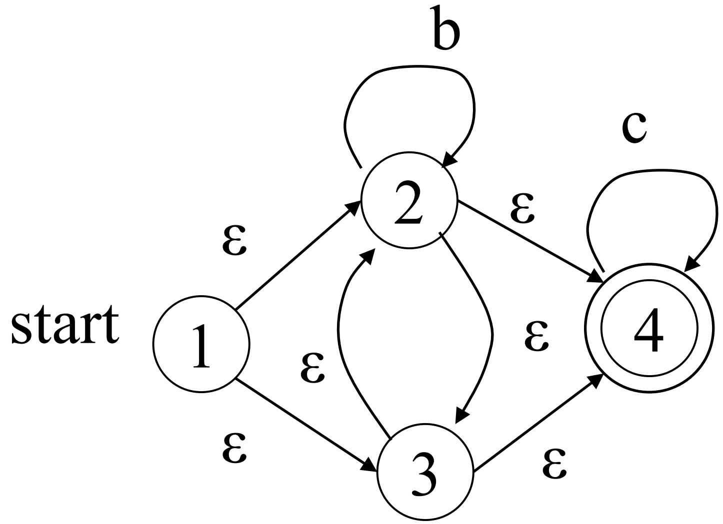
Class Problem

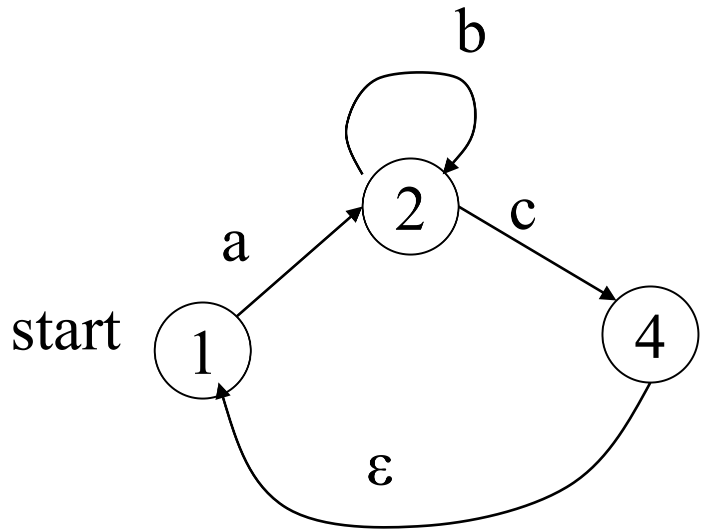
Convert this NFA to a DFA



NFA to DFA Optimizations

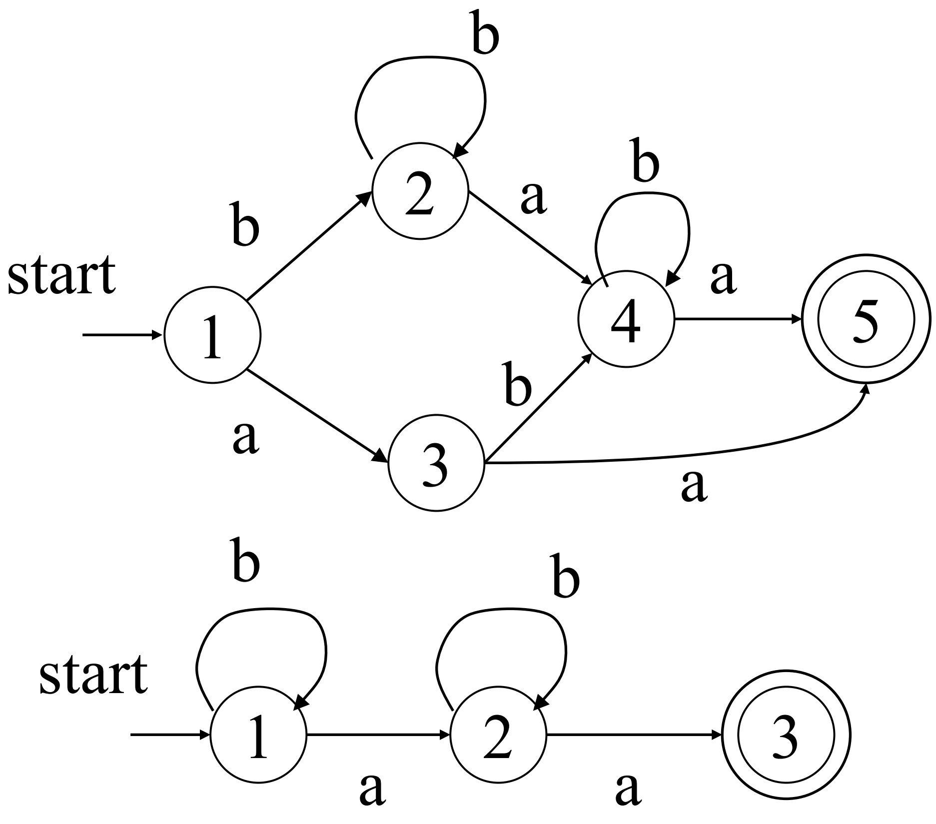
- Prior to NFA to DFA conversion:
- Empty cycle removal
  - Combine nodes that comprise cycle
  - Combine 2 and 3
- Empty transition removal
  - Remove state 4, change transition 2-4 to 2-1





State Minimization

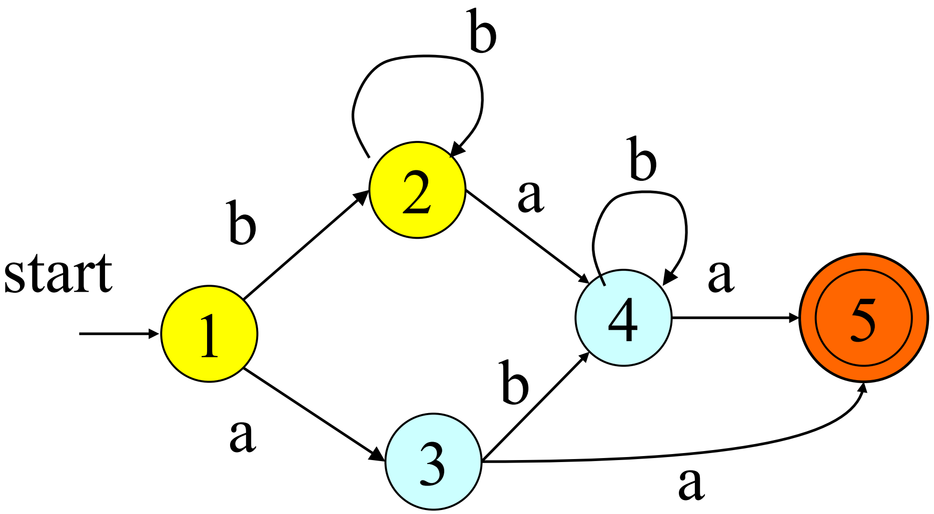
- Resulting DFA can be quite large
  - Contains redundant or equivalent states



Both DFAs accept
b*ab*a

State Minimization

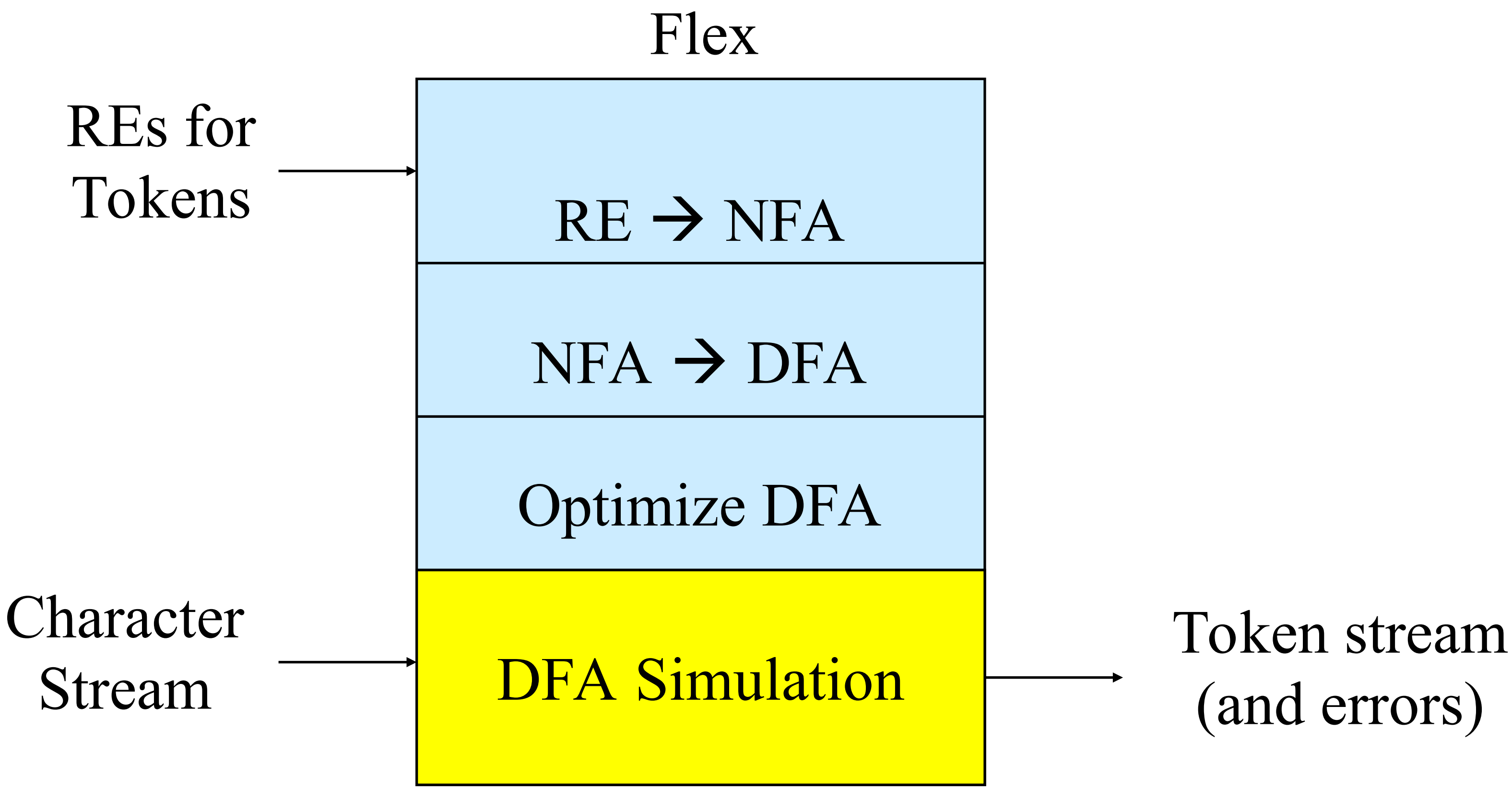
- Idea – find groups of equivalent states and merge them
  - All transitions from states in group G1 go to states in another group G2
  - Construct minimized DFA such that there is 1 state for each group of states



Basic strategy: identify distinguishing transitions

Putting It All Together

- Remaining issues: how to simulate, multiple REs, producing a token stream, longest match, rule priority



Simulating the DFA

- Straight-forward translation of DFA to C program
- Transitions from each state/input can be represented as table
  - Table lookup tells where to go based on current state/input

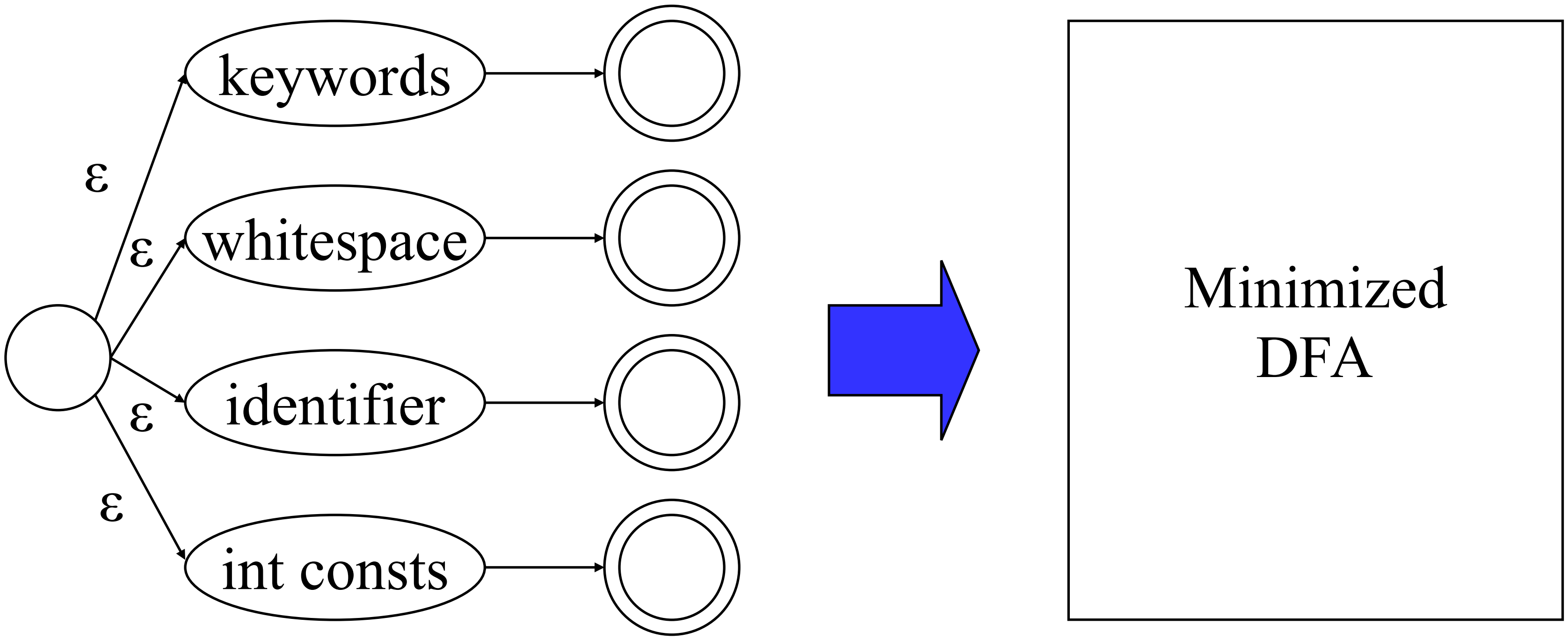
```
trans_table[NSTATES][NINPUTS];
accept_states[NSTATES];
state = INITIAL;

while (state != ERROR) {
      c = input.read();
      if (c == EOF) break;
      state = trans_table[state][c];
}
return accept_states[state];
```

Not quite
this simple
but close!

Handling Multiple REs

- Combine the NFAs of all the regular expressions into a single NFA
- Accepting states are not equivalent – they recognize different REs



Remaining Issues

- Token stream at output
  - Associate tokens with final states
  - Output corresponding token when reach final state
- Longest match
  - When in a final state, look if there is a further transition. If no, return the token for the current final state
- Rule priority
  - Same longest matching token when there is a final state corresponding to multiple tokens
  - Associate that final state to the token with highest priority