

Lexical Analysis – Part I

EECS 483 – Lecture 2

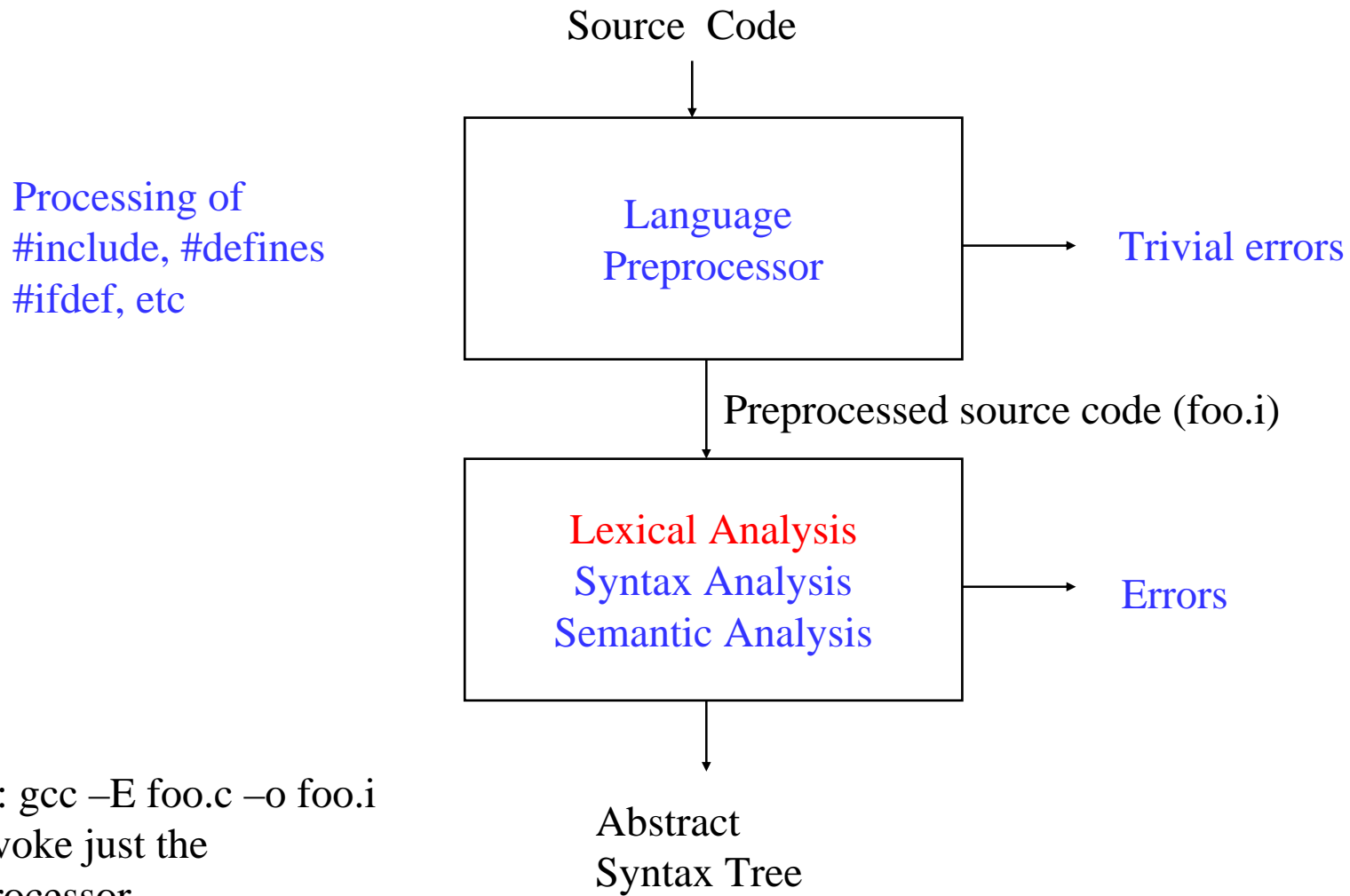
University of Michigan

Monday, September 11, 2006

Announcements

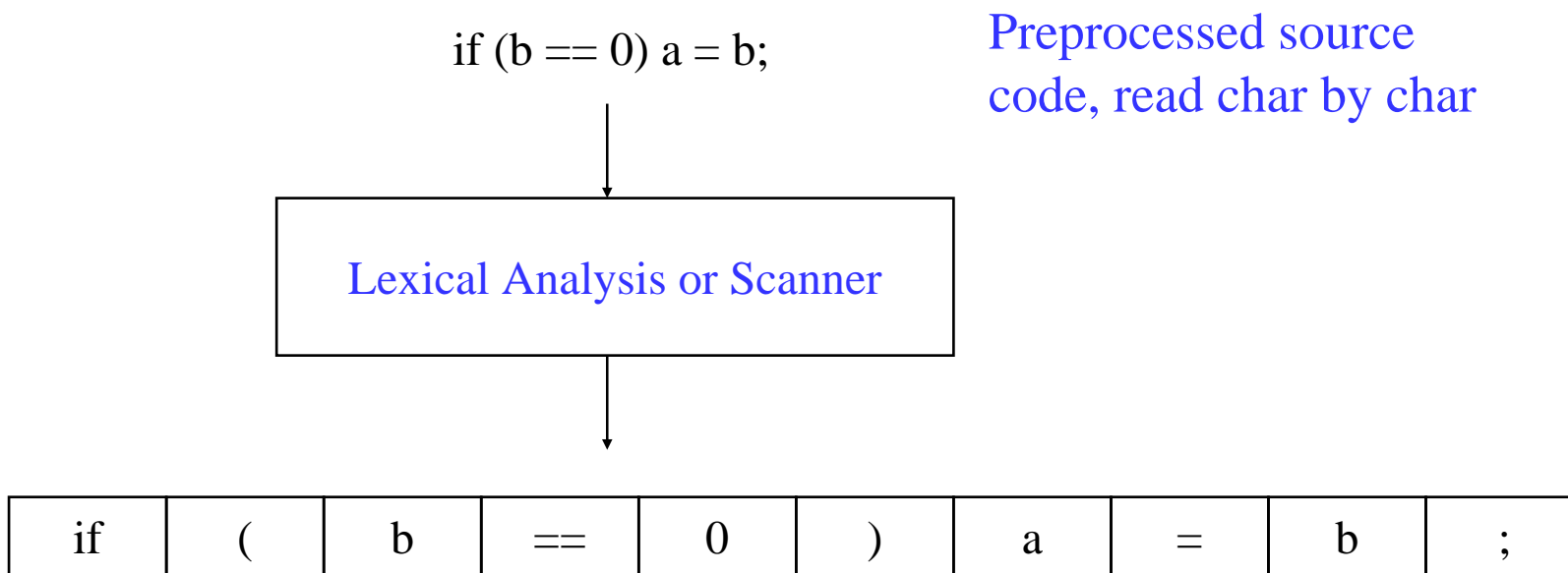
- ❖ Course webpage is up
 - » <http://www.eecs.umich.edu/~mahlke/483f06>
 - » Also link from EECS webpage, under courses
- ❖ Project 1
 - » Available by Wednes
- ❖ Simon's office hours
 - » 1:30 – 3:30, Tues/Thurs
 - » Study room, Table 2 on the first floor of CSE
 - » Note, Tuesdays 3-3:30, the study rooms are not available, so he will just be at one of the tables by Foobar for the last half hour

Frontend Structure



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

Lexical Analysis Process



Lexical analysis

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

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 EECS
483\””

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
 - » ϵ empty string
 - » $R|S$ either R or S (alteration), where $R, S = \text{RE}$
 - » 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
 - » $L(abc) = \{ abc \}$
 - » $L(\text{hello|goodbye}) = \{ \text{hello, goodbye} \}$
 - » $L(1(0|1)^*) =$ 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 : $(R|\epsilon)$
- ❖ $[abcd]$ one of listed characters: $(a|b|c|d)$
- ❖ $[a-z]$ one character from this range:
 $(a|b|c|d\dots|z)$
- ❖ $[^ab]$ anything but one of the listed chars
- ❖ $[^a-z]$ one character not from this range

Example Regular Expressions

❖ Regular Expression, R

- » a
- » ab
- » a|b
- » (ab)*
- » (a| ϵ)b
- » digit = [0-9]
- » posint = digit+
- » int = -? posint
- » real = int (ϵ | (. posint))
= -?[0-9]+ (ϵ |(. [0-9]+))

❖ Strings in L(R)

- » “a”
- » “ab”
- » “a”, “b”
- » “”, “ab”, “abab”, ...
- » “ab”, “b”
- » “0”, “1”, “2”, ...
- » “8”, “412”, ...
- » “-23”, “34”, ...
- » “-1.56”, “12”, “1.056”, ...
- » Note, “.45” is not allowed in this definition of real

Class Problem

- ❖ A. Whats the difference?
 - » [abc] abc
- ❖ Extend the description of real on the previous slide to include numbers in scientific notation
 - -2.3E+17, -2.3e-17, -2.3E17

How to Break up Text

elsex = 0;

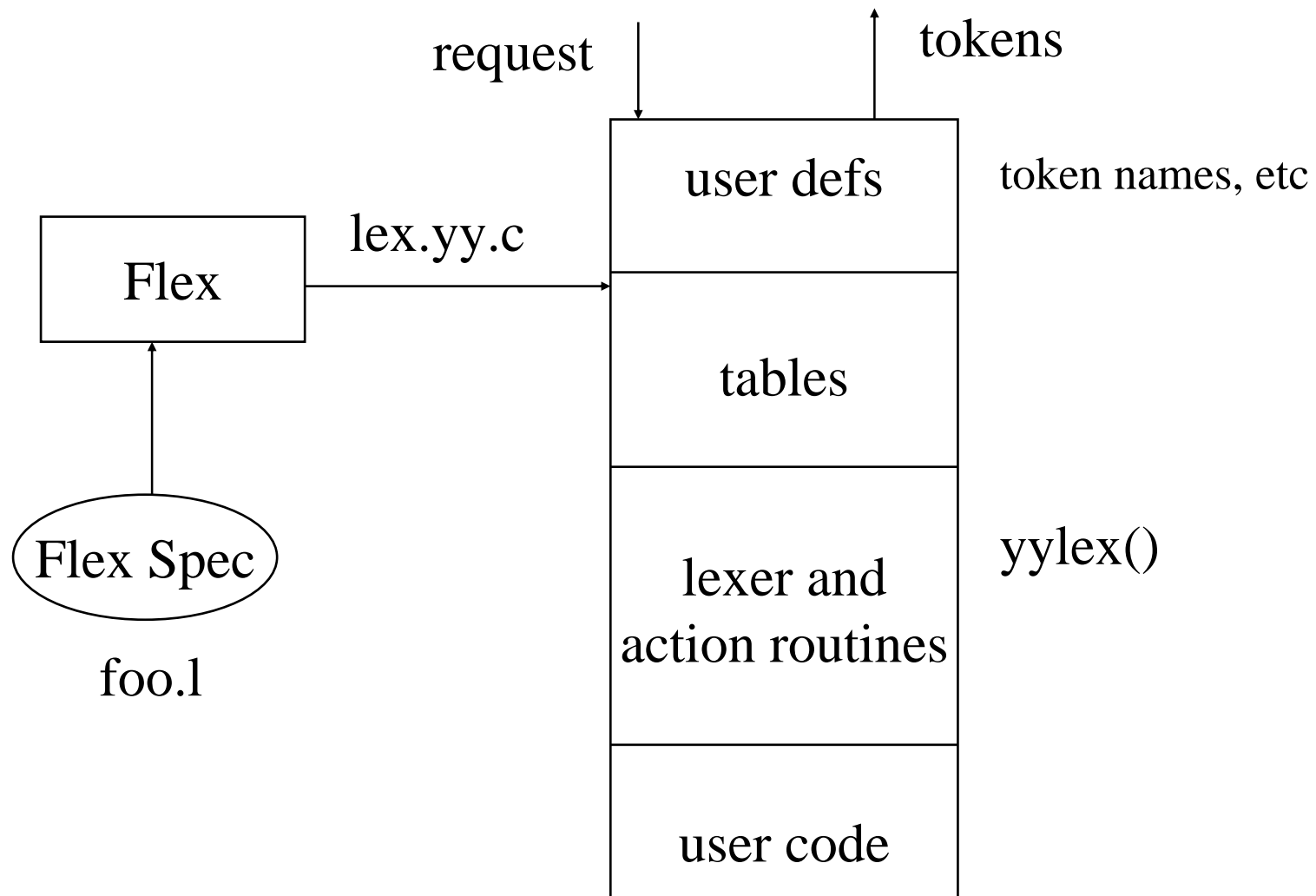
1	else	x	=	0	;
2	elsex	=	0	;	

- ❖ 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
 - Flex = fast lex, later developed by Free Software Foundation
 - » Yacc/bison – yet another compiler/compiler (next week)
- ❖ 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



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 (continued)

❖ Rules section

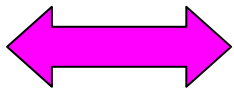
- » Contains lexical patterns and semantic actions to be performed upon a pattern match. Actions should be surrounded by { } (though not always necessary)
- » Again, all lines with a blank in column 1 are copied to the resulting C program

❖ User function section

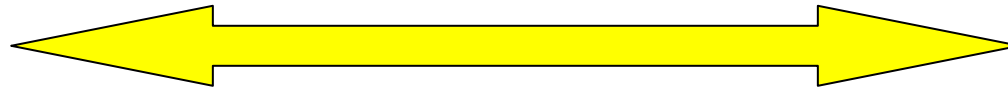
- » All lines in this section are copied to the final .c file
- » Unless the functions are very immediate support routines, better to put these in a separate file

Partial Flex Program

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



pattern



action

Flex Program

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

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

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

Running the above program:

```
[ 17 ] sandbox -: flex count.l
[ 18 ] sandbox -: gcc lex.yy.c -lfl
[ 19 ] sandbox -: a.out < count.l
# of lines = 16, # of chars = 245
```

Another Flex Program

```
% {
/* recognize articles a, an, the */
#include <stdio.h>
% }

%%
[ \t]+      /* skip white space - action: do nothing */ ;
a |        /* | indicates do same action as next pattern */
an |
the        {printf("%s: is an article\n", yytext);}
[a-zA-Z]+  {printf("%s: ???\n", yytext);}

%%
main()
{
    yylex();
}
```

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

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

Class Problem

Write the flex rules to strip out all comments of the form
/*, */ from an input program

Hints: Action ECHO copies input token to output

Think of using 2 states

Keyword BEGIN “state” takes you to that state

How Does Lex Work?

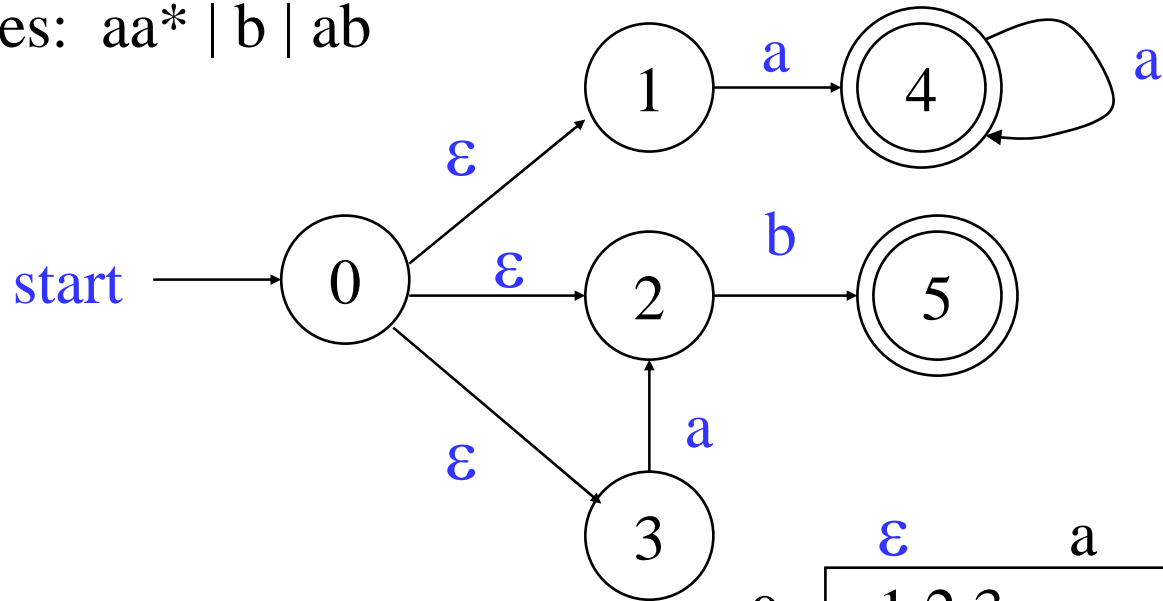
- ❖ Formal basis for lexical analysis is the finite state automaton (FSA)
 - » REs generate regular sets
 - » FSAs recognize regular sets
- ❖ FSA – informal defn:
 - » A finite set of states
 - » Transitions between states
 - » An initial state (start)
 - » A set of final states (accepting states)

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: $aa^* | b | ab$

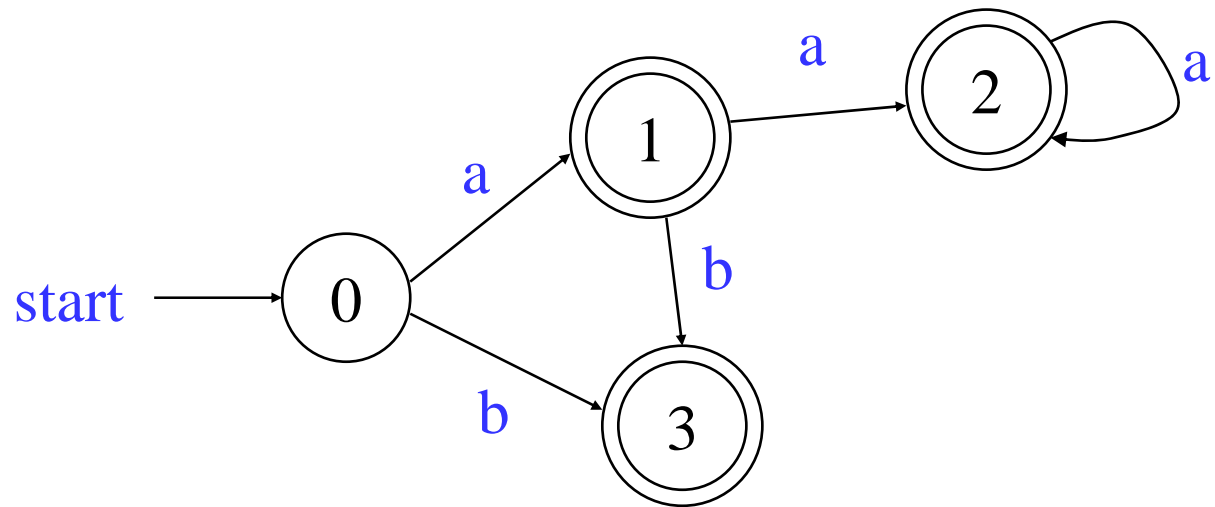


Can represent FA with either graph or transition table

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

DFA Example

Recognizes: $aa^* \mid b \mid ab$



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

Class Problem

Is this a DFA or NFA?

What strings does it recognize?

