Concepts Introduced in Chapter 3

- Lexical Analysis
- Regular Expressions (RE)
- Lex
- Nondeterministic Finite Automata (NFA)
- Converting an RE to an NFA
- Deterministic Finite Automata (DFA)
- Converting an NFA to a DFA
- Minimizing a DFA

Lexical Analysis

- Why separate the analysis phase of compiling into lexical analysis and parsing?
  - Simpler design of both phases
  - Compiler efficiency is improved

Lexical Analysis Terms

- A ***token*** is a group of characters having a collective meaning (e.g. id).
- A ***lexeme*** is an actual character sequence forming a specific instance of a token (e.g. num).
- A ***pattern*** is the rule describing how a particular token can be formed (e.g. [A-Za-z_][A-Za-z_0-9]*).
- Characters between tokens are called ***whitespace*** (e.g.blanks, tabs, newlines, comments).
- A lexical analyzer reads input characters and produces a sequence of tokens as output.

followed by Fig. 3.1, 3.2

Attributes for Tokens

- Some tokens have attributes that can be passed back to the parser.
  - Constants
    - value of the constant
  - Identifiers
    - pointer to the corresponding symbol table entry

Lexical Errors

- Only possible lexical error is that a sequence of characters do not represent a valid token.
  - Use of @ character in C.
- The lexical analyzer can either report the error itself or report it back to the parser.
- A typical recovery strategy is to just skip characters until a legal lexeme can be found.
- Syntax errors are much more common when parsing.

General Approaches to Lexical Analyzers

- Use a lexical-analyzer generator, such as Lex.
- Write the lexical analyzer in a conventional programming language.
- Write the lexical analyzer in assembly language.

Languages

- An alphabet is a finite set of symbols.
- A string is a finite sequence of symbols drawn from an alphabet.
- The ε symbol indicates a string of length 0.
- A language is a set of strings over some fixed alphabet.

followed by Tale on pg 199, Fig. 3.6

Regular Expressions

Given an alphabet Σ

1. ε is a regular expression that denotes {ε}, the set containing the empty string.
2. For each $a \in \Sigma$, a is a regular expression denoting {a}, the set containing the string a.
3. r and s are regular expressions denoting the languages L(r) and L(s). Then
   a) (r) | (s) denotes L(r) ∪ L(s)
   b) (r)(s) denotes L(r) L(s)
   c) (r)* denotes (L(r))*

Regular Expressions (cont.)

- *
  - has highest precedence and is left associative.
- concatenation
  - has second highest precedence and is left associative.
- |
  - Has lowest precedence and is left associative.
- Example:

  a|(b(c*)) = a | bc*

Examples of Regular Expressions

Let Σ = {a, b}

a | b => {a, b}

(a | b) (a | b) => {aa, ab, ba, bb}

a* => {ε, a, aa, aaa, ... }

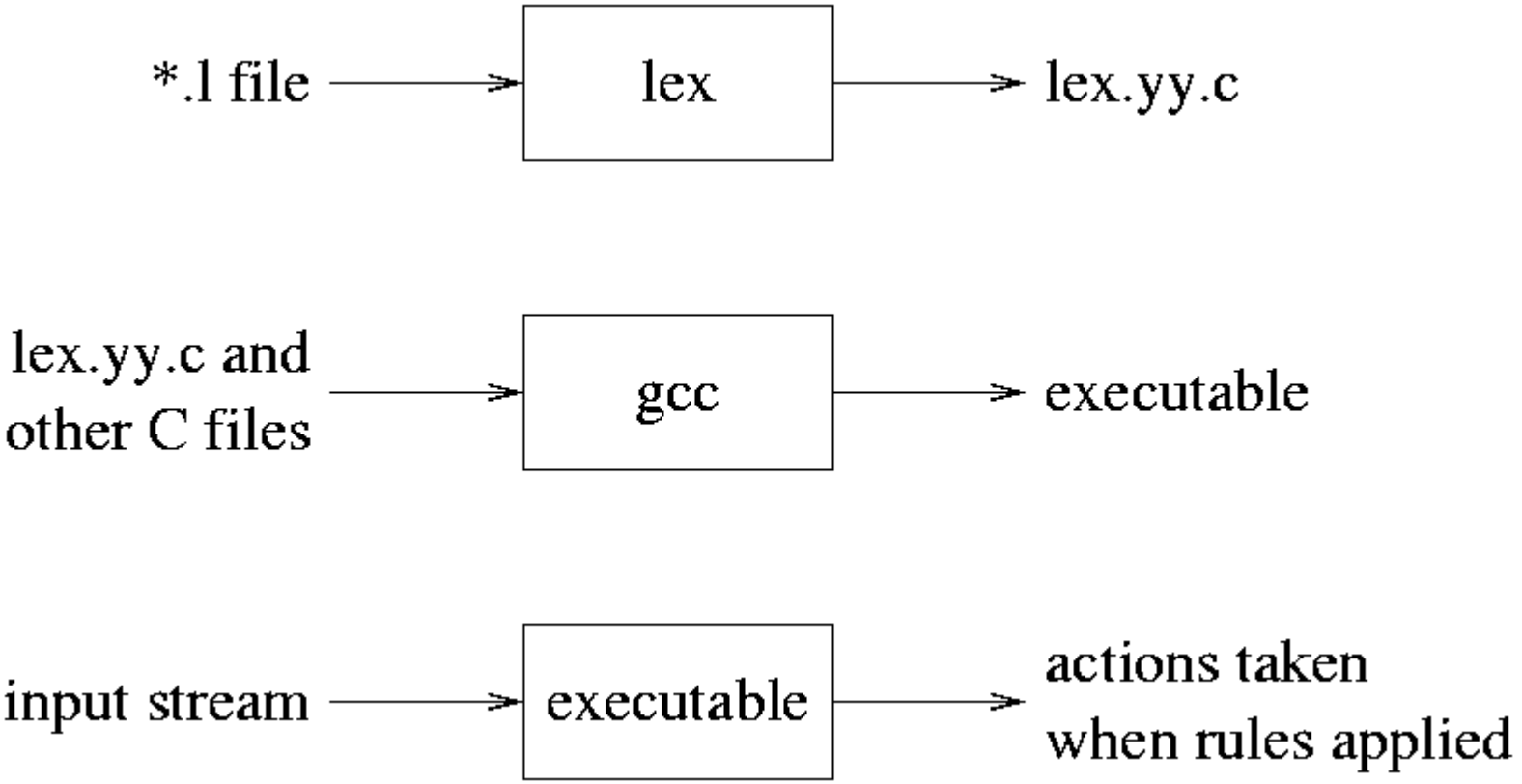
(a | b)* => all strings containing zero or more instances of a's and b's

a | a * b => { a, b, ab, aab, aaab, ... }

followed by Fig. 3.7

Lex - A Lexical Analyzer Generator

- Can link with a lex library to get a main routine.
- Can use as a function called yylex().
- Easy to interface with yacc.



Lex - A Lexical Analyzer Generator (cont)

Lex Source

{ definitions }
%%
{ rules }
%%
{ user subroutines }

Definitions

declarations of variables, constants, and regular definitions

Rules

regular expression action

Regular Expressions

operators "\ [ ] ^ -? . * + | ( ) $ / { }
actions C code

Lex Regular Expression Operators

- “s” string s literally
- \c character c literally (used when c would normally be used as a lex operator)
- [s] for defining s as a character class
- ^ to indicate the beginning of a line
- [^s] means to match characters not in the s character class
- [a-b] used for defining a range of characters (a to b) in a character class
- r? means that r is optional

Lex Regular Expression Operators (cont.)

- . means any character but a newline
- r* means zero or more occurrences of r
- r+ means one or more occurrences of r
- r1| r2 r1 or r2
- (r) r (used for grouping)
- $ means the end of the line
- r1/r2 means r1 when followed by r2
- r{m,n} means m to n occurrences of r

Example Regular Expressions in Lex

a*	zero or more a's
a+	one or more a's
[abc]	a, b, or c
[a-z]	lower case letter
[a-zA-Z]	any letter
[^a-zA-Z]	any character that is not a letter
a.b	a followed by any character followed by b
ab|cd	ab or cd
a(b|c)d	abd or acd
^B	B at the beginning of line
E$	E at the end of line

followed by Fig. 3.8

Lex (cont.)

Actions

Actions are C source fragments. If it is compound or takes more than one line, then it should be enclosed in braces.

Example Rules

```
[a-z]+                  printf("found word\n");
[A-Z][a-z]*     {       printf("found capitalized word\n");
                            printf{"  %s\n", yytext);
                        }
```

Definitions

```
name                    translation
```

Example Definition

```
digits                  [0-9]
```

Example Lex Program

```
digits      [0-9]
ltr         [a-zA-Z]
alpha       [a-zA-Z0-9]
%%

[-+]{digits}+          |
{digits}+                   printf("number: %s\n", yytext);
{ltr}(_|{alpha})*      printf("identifier: %s\n", yytext);
"'".."'"                    printf("character: %s\n", yytext);
.                                printf("?: %s\n", yytext);
```

Prefers longest match and earlier of equals.

followed by Fig. 3.12, 3.23

Nondeterministic Finite Automata

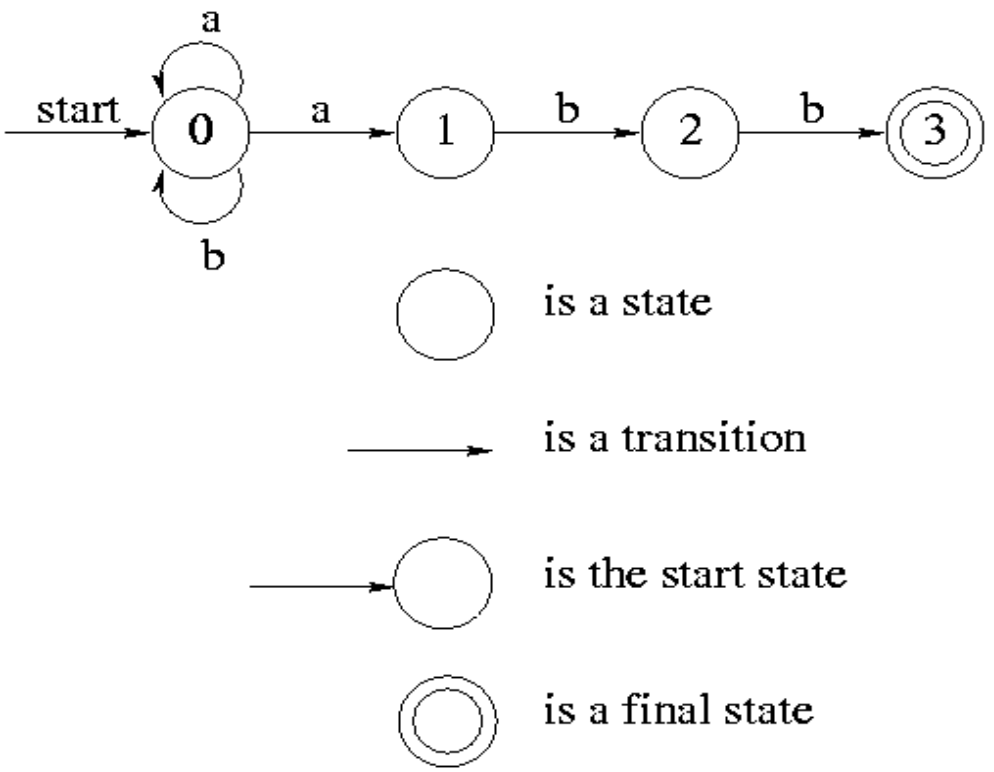
- A nondeterministic finite automaton (NFA) consists of
  - a set of states S
  - a set of input symbols Σ (the input symbol alphabet)
  - a transition function move that maps state-symbol pairs to sets of states
  - a state s0 that is distinguished as the start (or initial) state
  - a set of states F distinguished as accepting (or final) states

Operation of an Automata

- An automata operates by making a sequence of moves. A move is determined by a current state and the symbol under the read head. A move is a change of state and may advance the read head.

Representations of Automata

- Ex: (a|b)*abb
- Transition Diagram

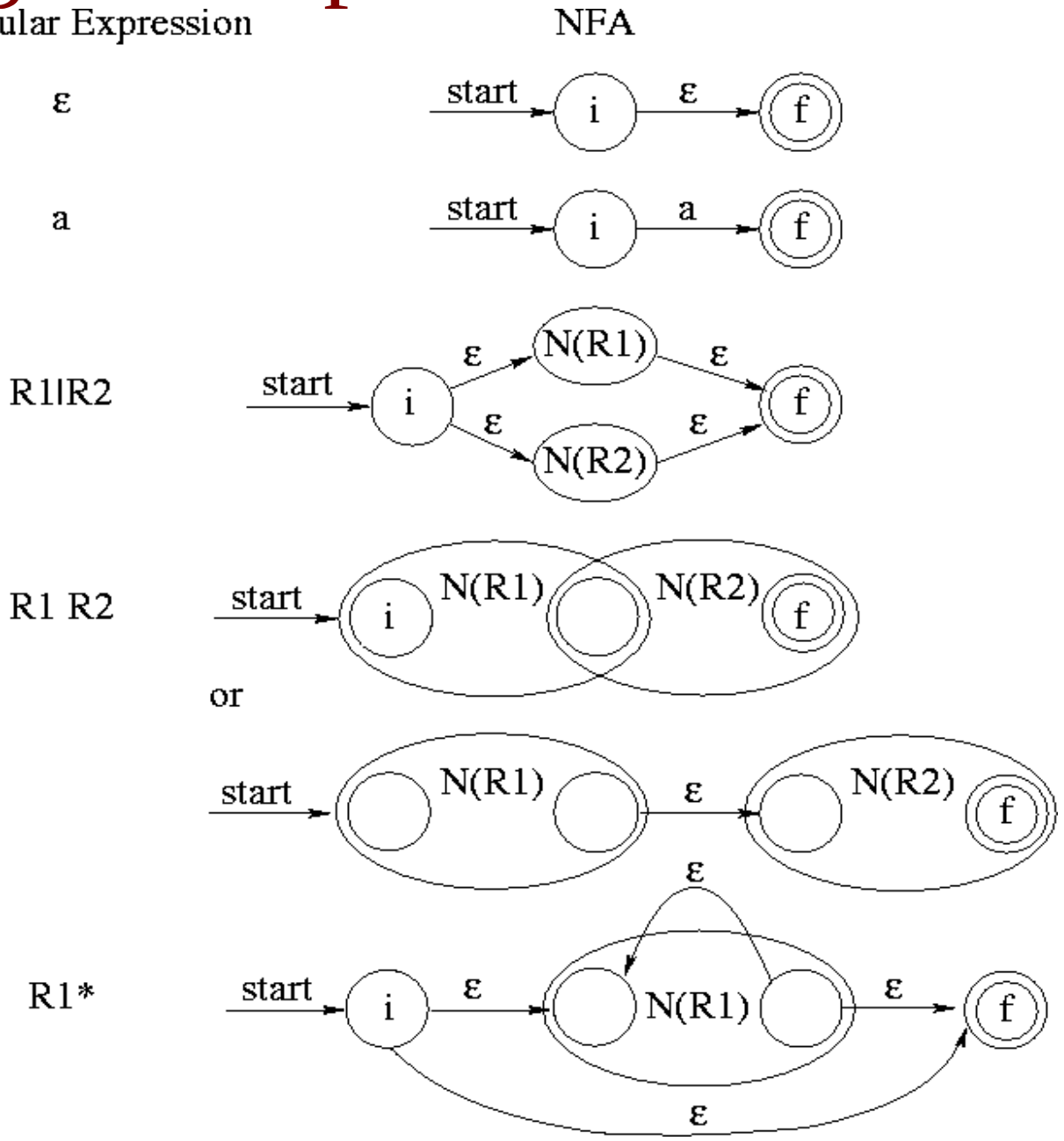


- Transition Table

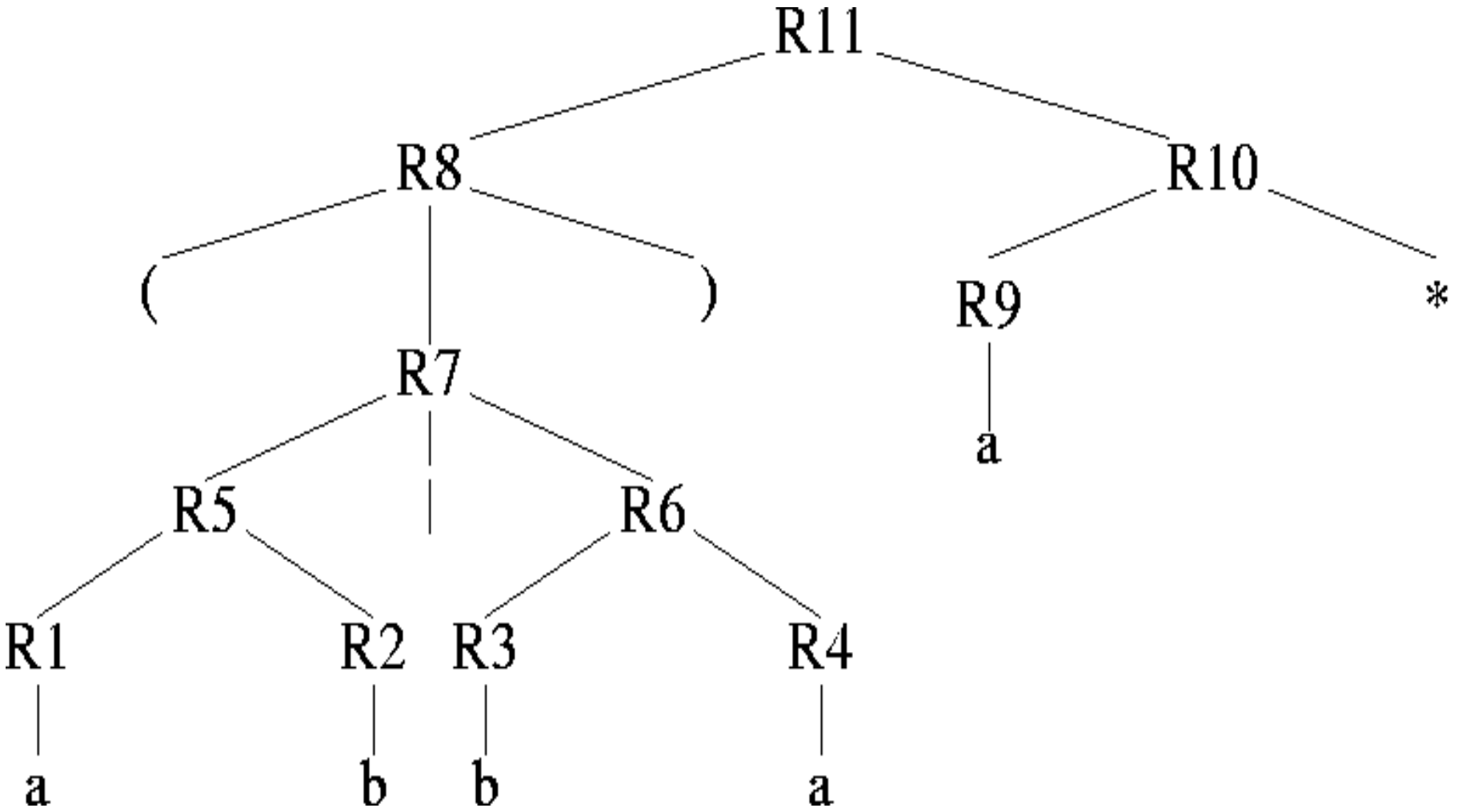
State	input symbol a	input symbol b
0	{0,1}	{0}
1	–	{2}
2	–	{3}
3	–	–

followed by Fig. 3.31

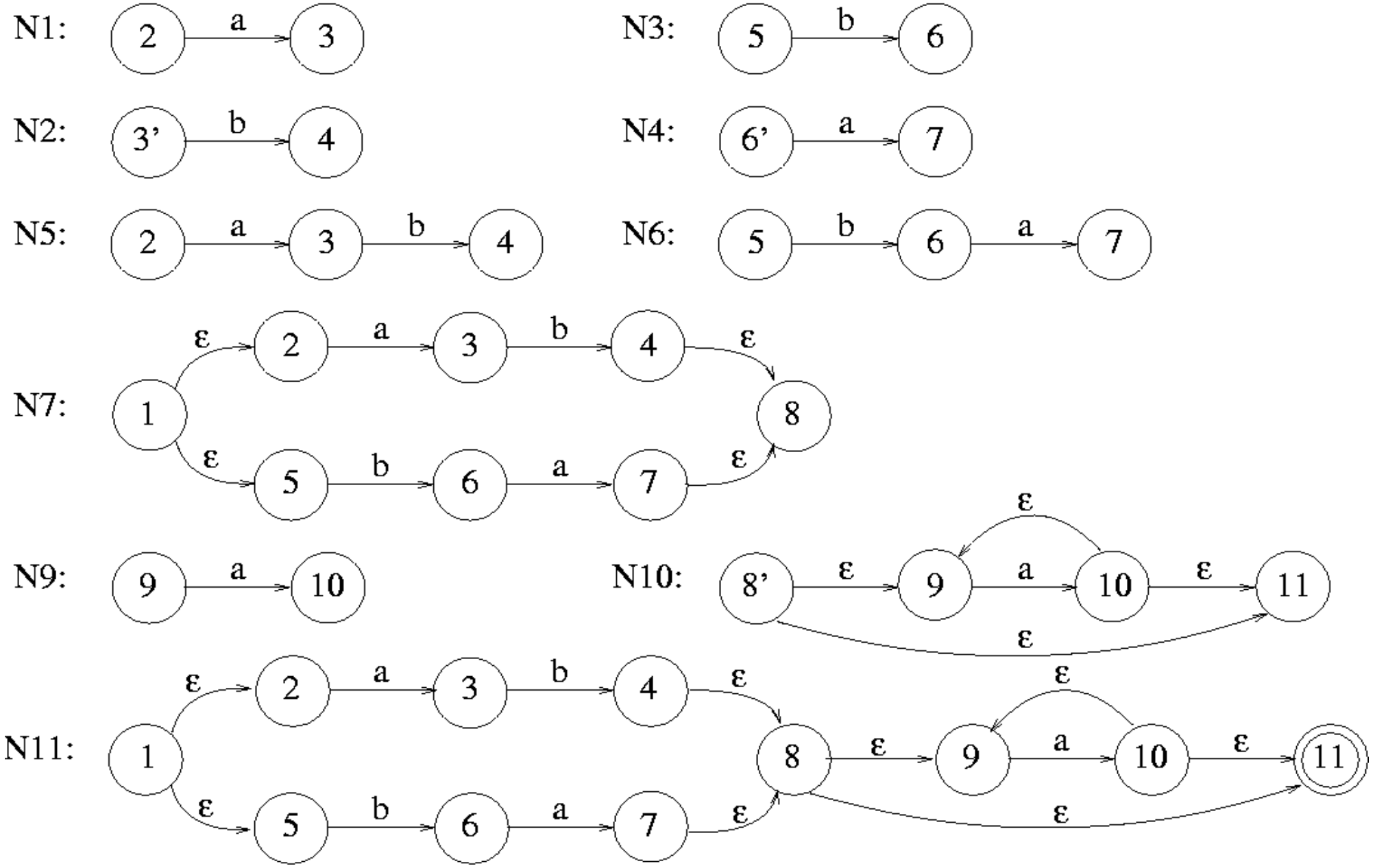
Regular Expression to an NFA



Decompostion of (ab|ba)a*



Decompostion of (ab|ba)a* (cont.)



Deterministic Finite Automata

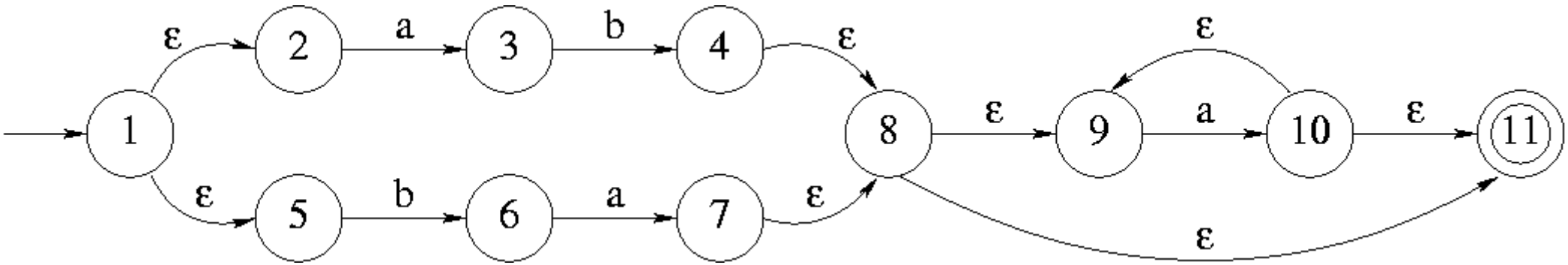
- An FSA is deterministic (a DFA) if

1. No transitions on input ε.

2. For each state s and input symbol *a*, there is at most one edge labeled *a* leaving s.

followed by Fig. 3.31, 3.32, 3.33

Example of Converting an NFA to a DFA



Example of Converting an NFA to a DFA (cont.)

$A = \text{e–closure}(\{1\}) = \{1, 2, 5\}$

mark A

$\{1, 2, 5\} \xrightarrow{a} \{3\}$

$B = \text{e–closure}(\{3\}) = \{3\}$

$\{1, 2, 5\} \xrightarrow{b} \{6\}$

$C = \text{e–closure}(\{6\}) = \{6\}$

mark B

$\{3\} \xrightarrow{b} \{4\}$

$D = \text{e–closure}(\{4\}) = \{4, 8, 9, 11\}$

mark C

$\{6\} \xrightarrow{a} \{7\}$

$E = \text{e–closure}(\{7\}) = \{7,8,9,11\}$

mark D

$\{4, 8, 9, 11\} \xrightarrow{a} \{10\}$

$F = \text{e–closure}(\{10\}) = \{9, 10, 11\}$

mark E

$\{7, 8, 9, 11\} \xrightarrow{a} \{10\}$

mark F

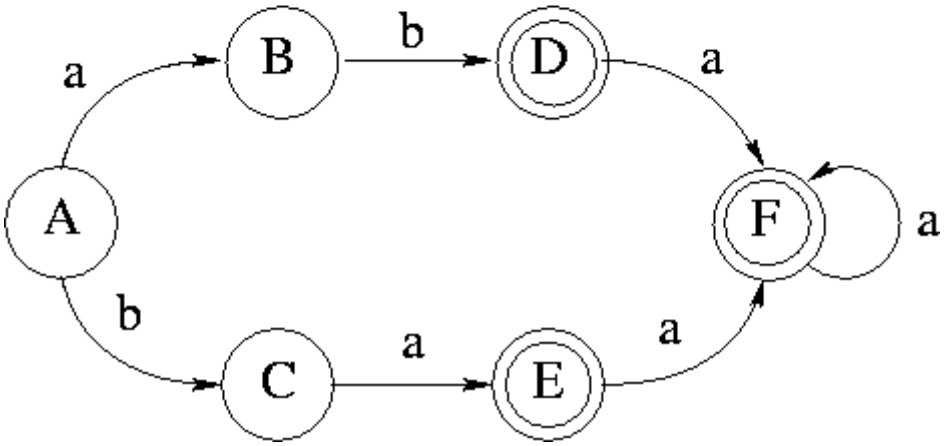
$\{9, 10, 11\} \xrightarrow{a} \{10\}$

Example of Converting an NFA to a DFA (cont.)

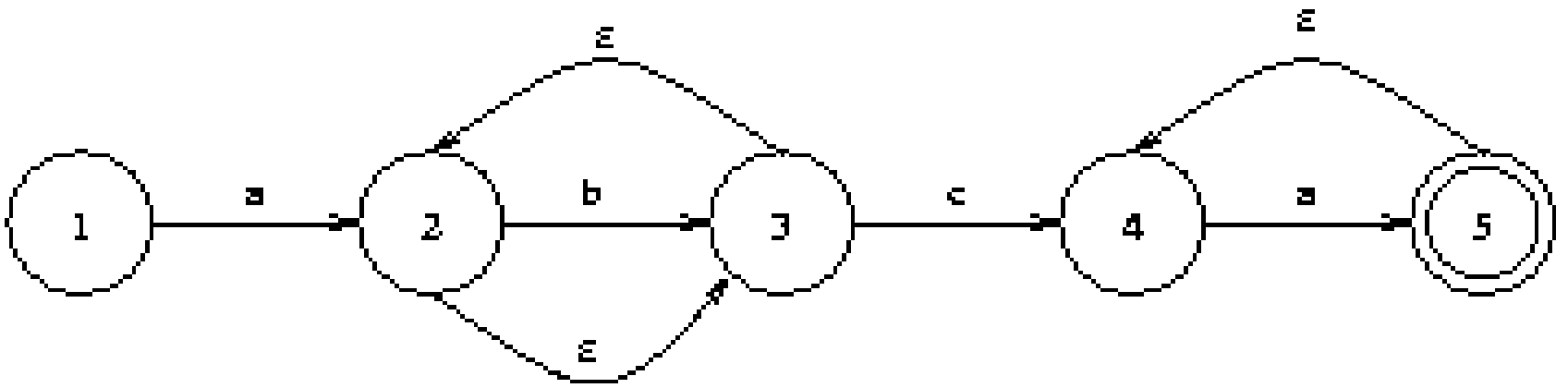
- Transition Table

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

- Transition Diagram



Another Example of Converting an NFA to a DFA



Lex Implementation Details

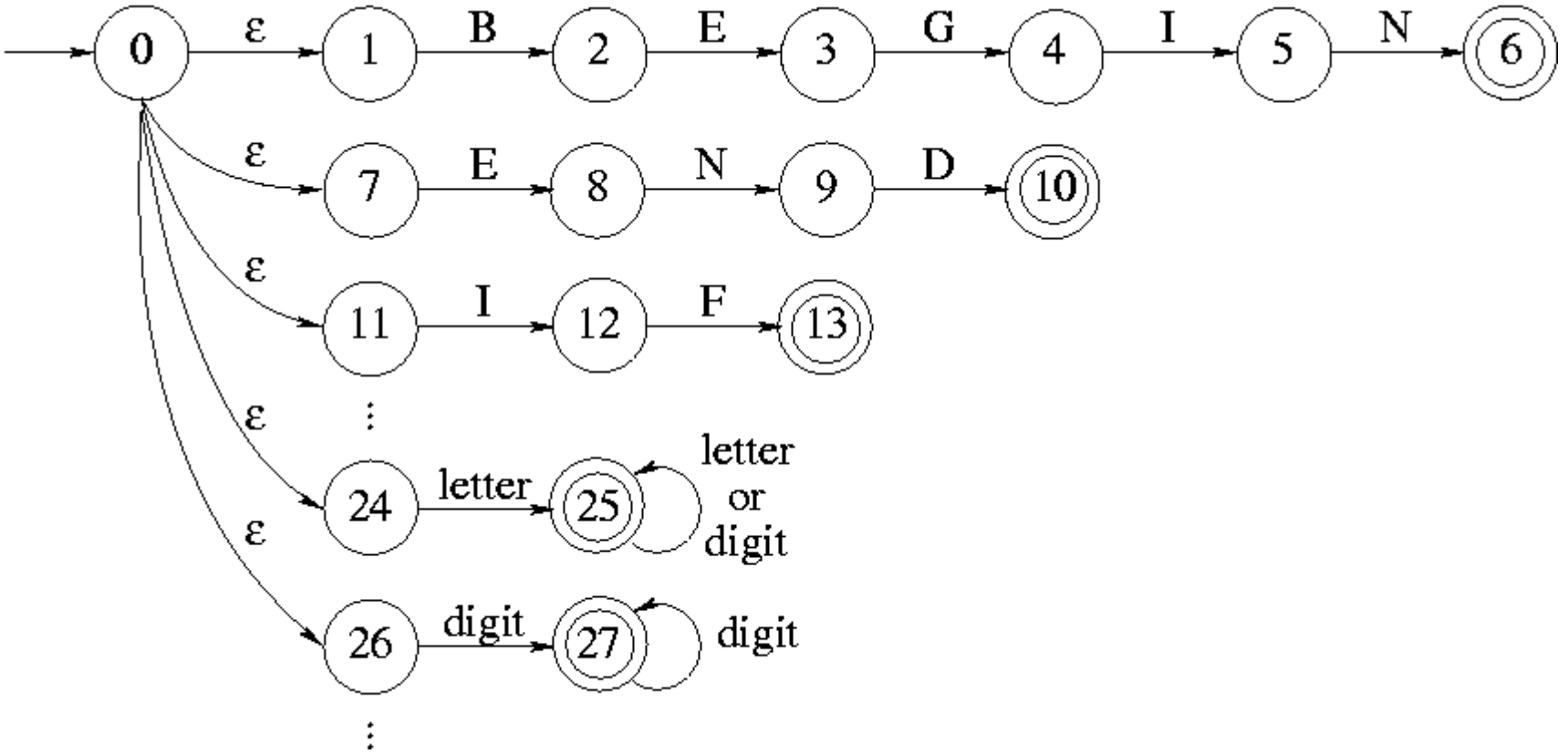
1. Construct an NFA to recognize the sum of the Lex patterns.
2. Convert the NFA to a DFA.
3. Minimize the DFA, but separate distinct tokens in the initial pattern.
4. Simulate the DFA to termination (i.e., no further transitions.)
5. Find the last DFA state entered that holds an accepting NFA state. (This picks the longest match.) If we can't find such a DFA state, then it is an invalid token.

Example Lex Program

```
%%
BEGIN                   { return (1); }
END                     { return (2); }
IF                      { return (3); }
THEN                    { return (4); }
ELSE                    { return (5); }
letter(letter|digit)*   { return (6); }
digit+                  { return (7); }
<                       { return (8); }
<=                      { return (9); }
=                       { return (10); }
<>                      { return (11); }
>                       { return (12); }
>=                      { return (13); }
```

Lex Implementation Details (cont.)

- NFA



Lex Implementation Details (cont.)

- DFA