Concepts Introduced in Chapter 3

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

Lexical Analysis

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

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.

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

- The 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.

Terms for Parts of Strings

- A *prefix* of string *s* is any string obtained by removing zero or more symbols from the end of *s*.
- A *suffix* of string *s* is any string obtained by removing zero or more symbols from the beginning of *s*.
- A *substring* of *s* is obtained by deleting any prefix and any suffix from *s*.
- The *proper* prefixes, suffixes, and substrings of a string *s* are those prefixes, suffixes, and substrings, respectively, of *s* that are not ε and not equal to *s* itself.
- A *subsequence* of *s* is any string formed by deleting zero or more not necessarily consecutive positions of *s*.

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, ... }

Nondeterministic Finite Automata

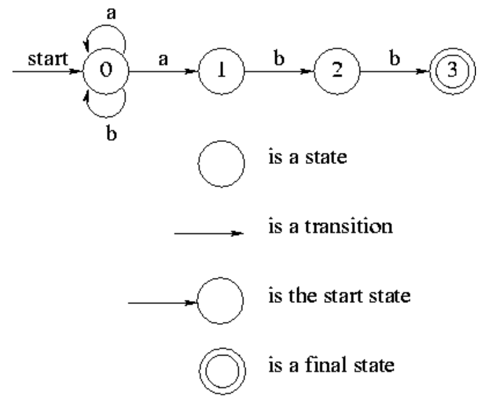
- A nondeterministic finite automaton (NFA) consists of
  - a set of states S
  - a set of input symbols Σ (the input symbol alphabet)
  - a transition function move that maps state-symbol pairs to sets of states
  - a state s0 that is distinguished as the start (or initial) state
  - a set of states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

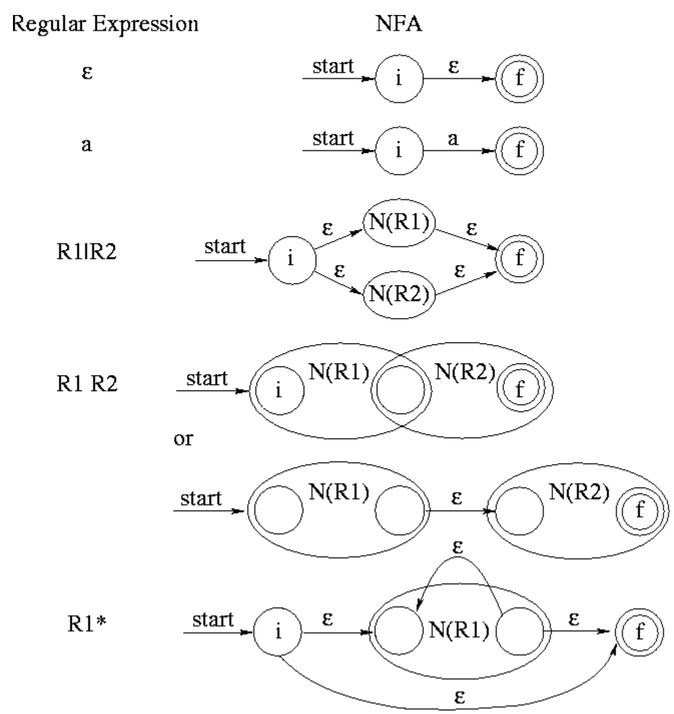
- Regular Expression (a|b)*abb
- Transition Diagram



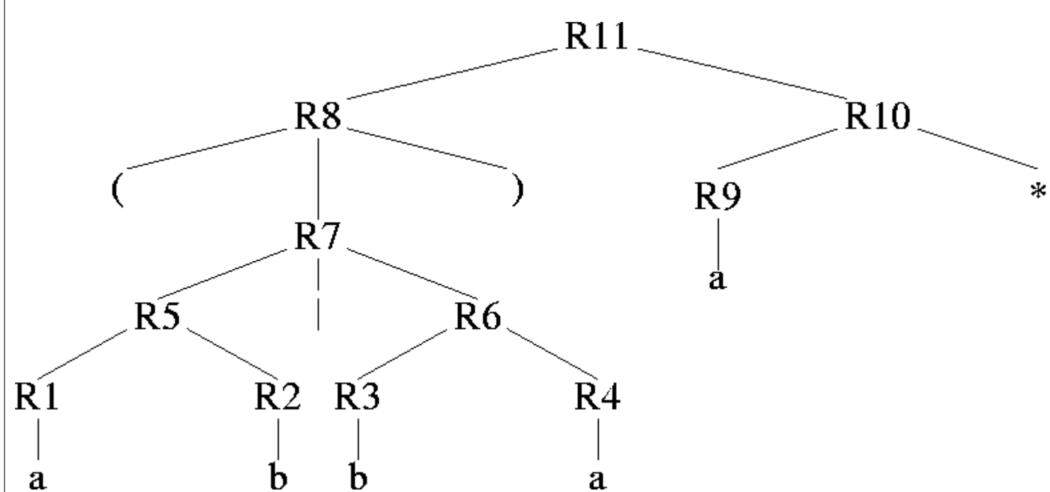
- Transition Table

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

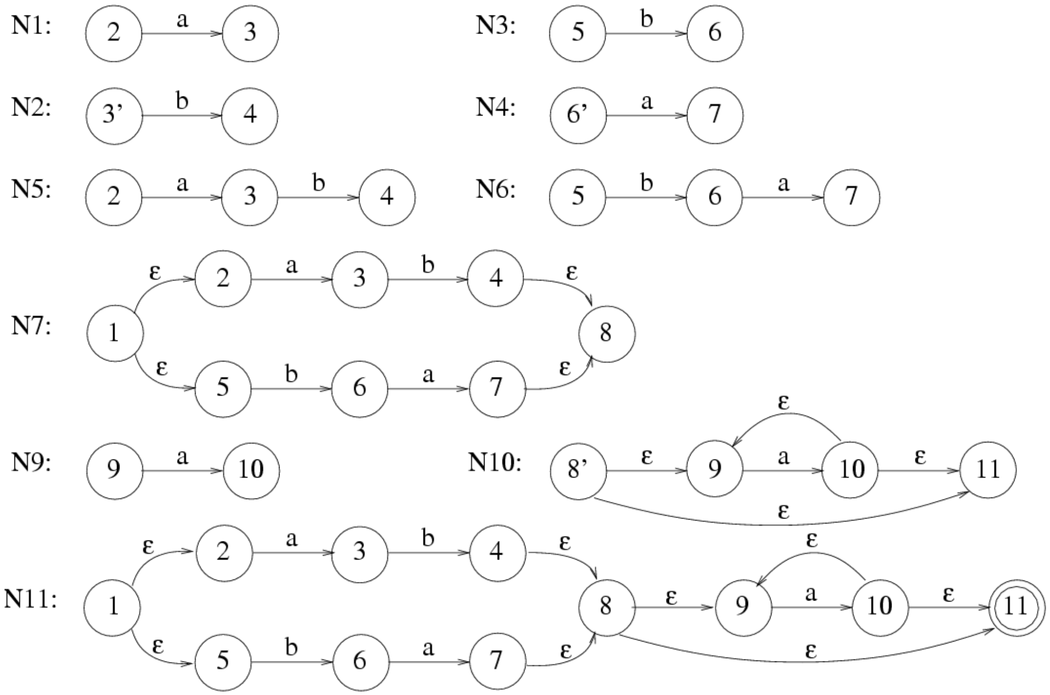
Converting a Regular Expression to an NFA



Decomposition of (ab|ba)a*



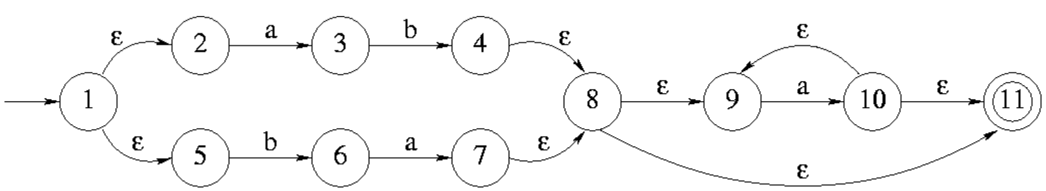
Decomposition of (ab|ba)a* (cont.)



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*.

Example of Converting an NFA to a DFA



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

A = e−closure({1}) = {1, 2, 5}
mark A
$\{1, 2, 5\} \xrightarrow{a} \{3\}$
B = e−closure({3}) = {3}
$\{1, 2, 5\} \xrightarrow{b} \{6\}$
C = e−closure({6}) = {6}
mark B
$\{3\} \xrightarrow{b} \{4\}$
D = e−closure({4}) = {4, 8, 9, 11}
mark C
$\{6\} \xrightarrow{a} \{7\}$

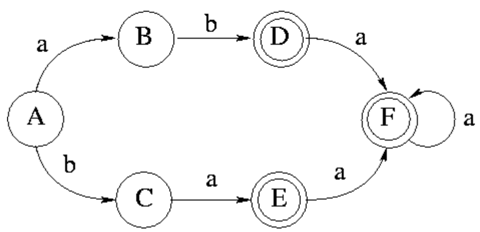
E = e−closure({7}) = {7,8,9,11}
mark D
$\{4, 8, 9, 11\} \xrightarrow{a} \{10\}$
F = e−closure({10}) = {9, 10, 11}
mark E
$\{7, 8, 9, 11\} \xrightarrow{a} \{10\}$
mark F
$\{9, 10, 11\} \xrightarrow{a} \{10\}$

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

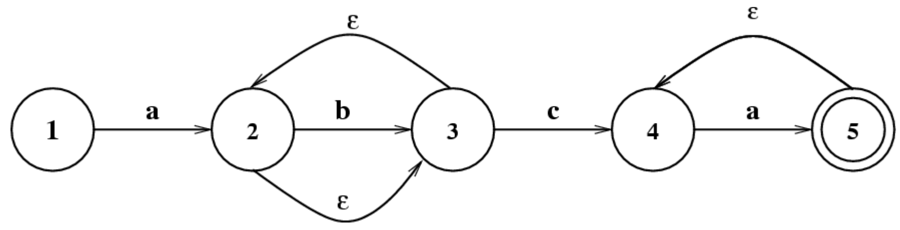
- Transition Table

	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

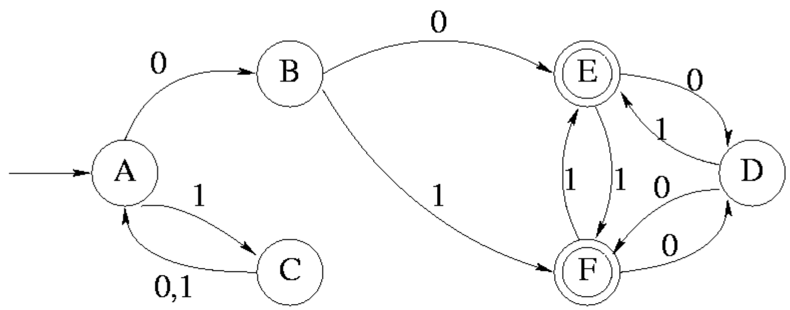


Minimizing a DFA

Given a DFA M
  If some M states ignore some inputs, add transitions to a "dead" state.
  Let P = { M's final states, M's non-final states}
  Let P´ = {}
loop:
  For each group G ϵ P do
    Partition G into subgroups so that s, t ϵ G are in the same subgroup iff each input *a* moves s and t to states of the same P-group.
  Put these new subgroups in P´.
  If P ≠ P´
    assign P´ to P.
    goto loop.

These subgroups denote the states of the minimized DFA.
Remove any dead states and unreachable states.

Example of Minimizing a DFA



	0	1
A	B	C
B	E	F
C	A	A
D	F	E
E	D	F
F	D	E

Example of Minimizing a DFA (cont.)

a: {A, B, C, D}   b: {E, F}

A B C D   E F
aa bb aa bb   ab ab

a: {A, C}   b: {B, D}   c: {E, F}

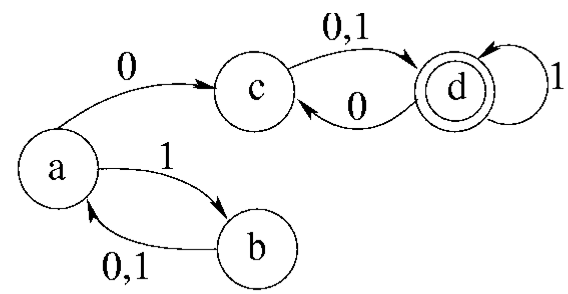
A C B D   E F
ba aa cc cc   bc bc

a: {A}   b: {C}   c: {B, D}   d: {E, F}

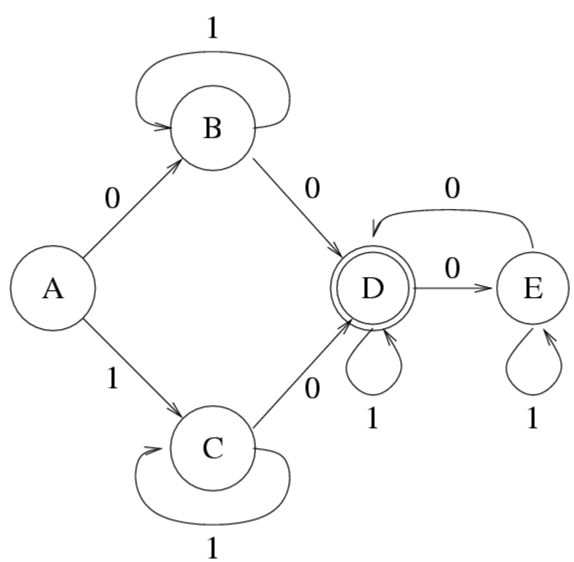
_ _ B D E F
dd dd cd cd

Example of Minimizing a DFA (cont.)

	0	1
a	c	b
b	a	a
c	d	d
d	c	d

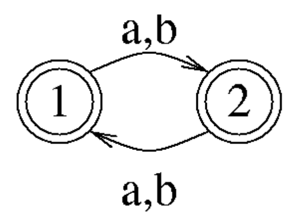


Another Example of Minimizing a DFA



Example of Minimizing a DFA with All Accepting States and No Dead States

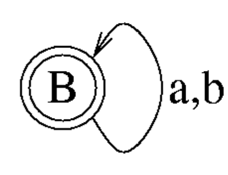
(a | b)*



	a	b
1	2	2
2	1	1

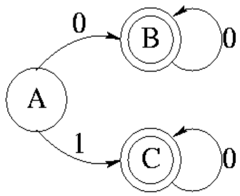
A   B
{ } {1, 2}
BB BB

	a	b
B	B	B



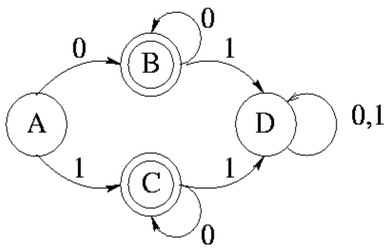
Example of Minimizing a DFA with a Dead State

- Original Transition Diagram



	0	1
A	B	C
B	B	–
C	C	–

- After Adding a Dead State



	0	1
A	B	C
B	B	D
C	C	D
D	D	D

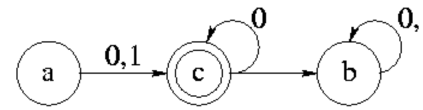
Example of Minimizing a DFA with a Dead State (cont.)

a	b
{A, D}	{B, C}
bb aa	ba ba

a	b	c
{A}	{D}	{B, C}
cc	bb	cb cb

	0	1
a	c	c
b	b	b
c	c	b

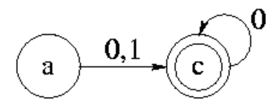
or



Remove dead state

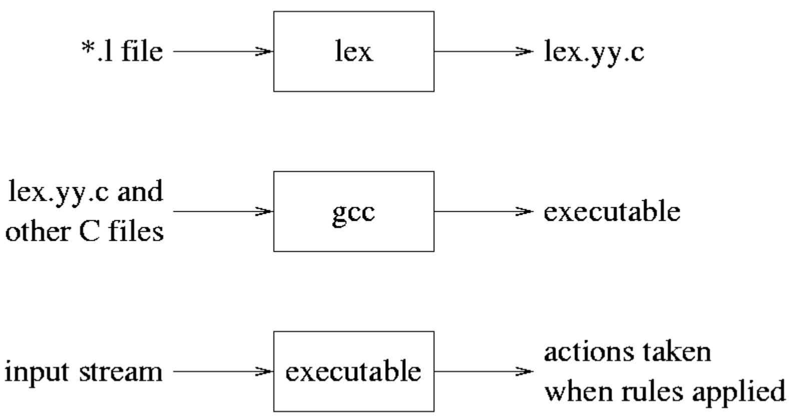
	0	1
a	c	c
c	c	–

or



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 occurances of r
- r+ means one or more occurances 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 occurences 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

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]
```

Start Conditions in Lex

- Start conditions are a mechanism for conditionally activating rules.
- Start conditions are declared in the definitions section. The INITIAL start condition is implicitly declared and is initially active. The %x means that the condition is exclusive.

  %x NAME

- Start conditions are activated using the BEGIN action. You can also refer to these conditions by number, where INITIAL has the value of zero.

  BEGIN NAME;

- Rules with a pattern that has a <NAME> as a prefix are only applied when the NAME condition is active.

Example of Using Start Conditions

```
%x CPP

%%

^#                  BEGIN CPP;
…
<CPP>[\n]           BEGIN INITIAL;
```

The <CPP> rules are only applied for C preprocessor commands.

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.

Implementation Details

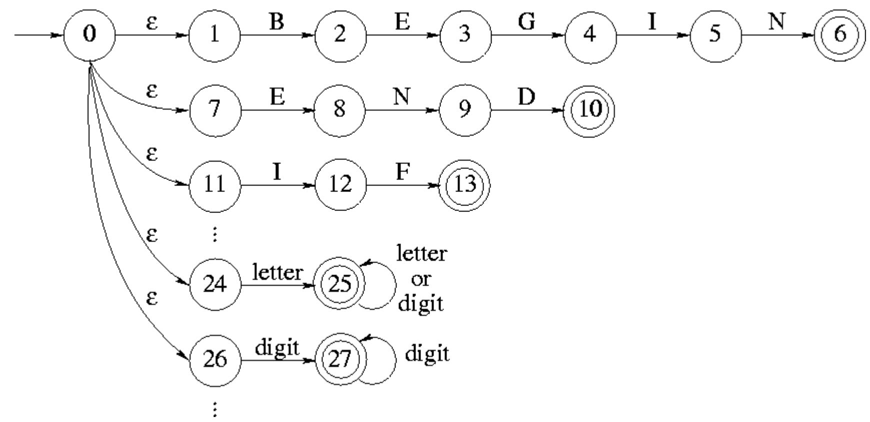
1. Construct an NFA to recognize the sum of the Lex patterns.
2. Convert the NFA to a DFA.
3. Minimize the DFA, but separate distinct tokens in the initial pattern.
4. Simulate the DFA to termination (i.e. no further transitions).
5. Find the last DFA state entered that holds an accepting NFA state. (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); }
```

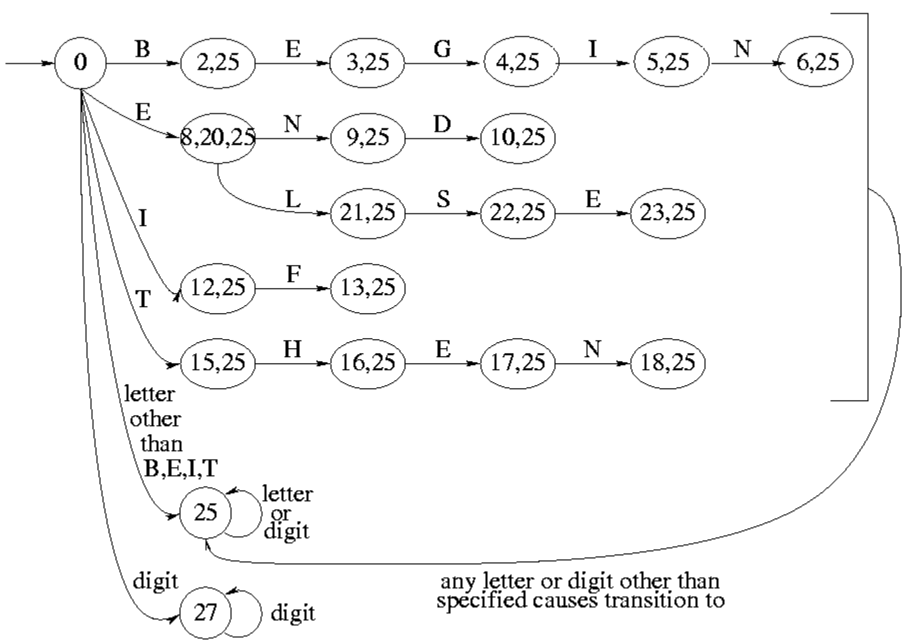
Implementation Details (cont.)

- NFA



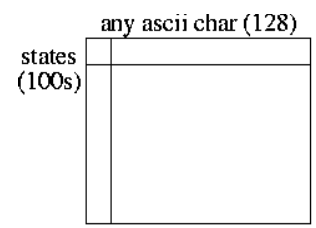
Implementation Details (cont.)

- DFA

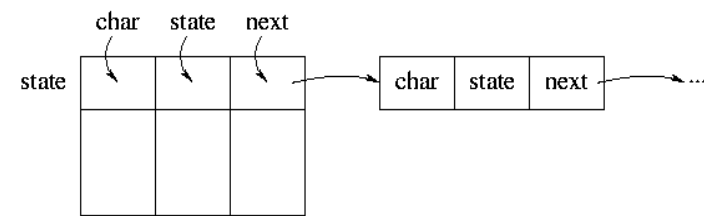


Representing the Transition Diagram

- 2D array
  fastest, but too much space

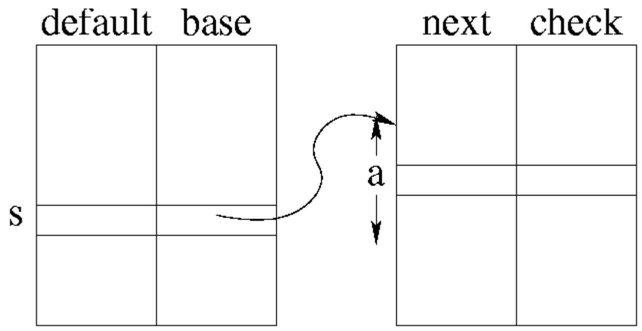


- linked list to store transitions out of each state



Representing the Transition Diagram

- combines fast array access with linked list compactness




```
procedure nextstate(s, a);
  if  check[base[s]+a] = s  then
      return next[base[s]+a];
  else
      return nextstate(default[s],a);
```