Scanning/Lexical Analysis

CIS*4650 (Winter 2020)

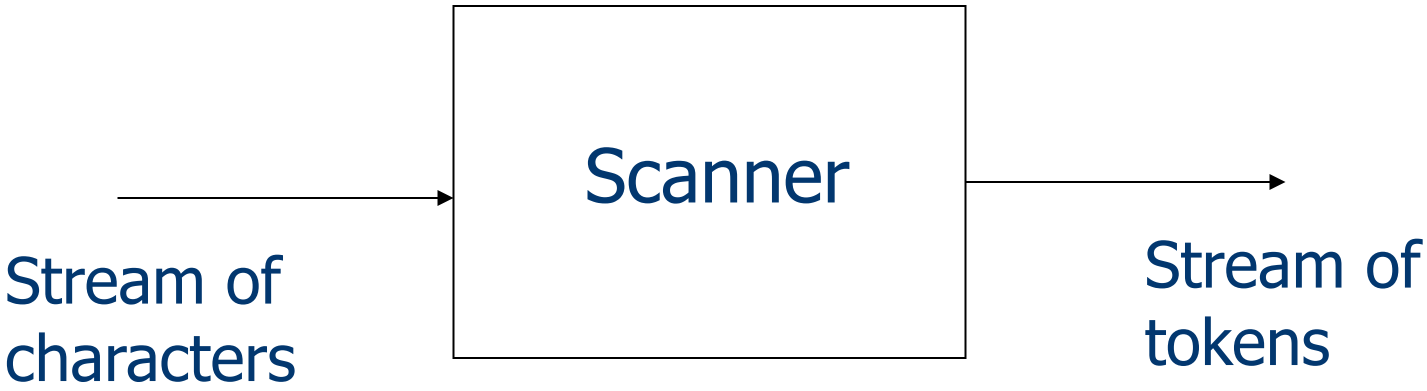
Review

- Compiler: translates a source program into a target machine code
  - Front-End Analysis: language-specific, focused on analysis, and systematic solutions with tools
  - Back-End Synthesis: machine-specific, focused on synthesis, and ad hoc solutions through coding

- Front-End Analysis:
  - Scanning/Lexical Analysis: break an input stream into a token sequence
  - Parsing/Syntactic Analysis: parse the phrase structure of a program
  - Semantic Analysis: calculate the meaning and generate the intermediate code of a program

What is a Scanner?

- Split input characters into a sequence of tokens:



What is a Token?

- A minimal sequence of characters that represents a unit of information
- Each language specifies a finite set of token types

Type	Examples
ID	foo n14 last
INTEGER	73 0 515 082
REAL	66.1 .5 10. 1e67 5.5e-10
IF	if
COMMA	,
NOTEQ	!=
LPAREN	(
RPAREN	)

Non-token Examples

- Some character sequences are not tokens:

comment	/* try again */
preprocessor directive	#include <stdio.h>
preprocessor directive	#define NUMS 5 , 6
macro	NUMS
blanks, tabs, newlines	

Example Input/Output

- Input program:

```
/* find a zero */
float match0(char *s)
{
    if( !strncmp(s, “0.0”, 3) )
        return(0.0);
}
```

- Output tokens:

FLOAT ID(match0) LPAREN CHAR STAR ID(s) RPAREN LBRACE
IF LPAREN BANG ID(strncmp) LPAREN ID(s) COMMA
STRING(0.0) COMMA INTEGER(3) RPAREN RPAREN RETURN
REAL(0.0) SEMI RBRACE EOF

Specifying Tokens

- Token structures can be complex, and English descriptions can be tedious, imprecise, and incomplete
  - E.g., identifiers and real numbers
- Need a formal system to specify them without ambiguity:
  - Allow review of design and validation of implementation
- Regular expressions:
  - succinct, precise
  - capable of representing infinite sets of strings

English Rules for Identifiers

An identifier is a sequence of letters and digits; the first character must be a letter.

The underscore _ counts as a letter.

Upper- and lower-case letters are different.

If the input stream has been parsed into tokens up to a given character, the next token is taken to include the longest string of characters that could possibly constitute a token.

Blanks, tabs, newlines, and comments are ignored except as they serve to separate tokens.

Some white space is required to separate otherwise adjacent identifiers, keywords, and constants.

Regular Expressions

- Basic Regular Expressions:
  - Given any character a from an alphabet Σ, a itself denotes a basic regular expression and the language it recognizes is $L(a) = \{a\}$.
  - Empty string: ε (also λ)

    $L(\varepsilon) = \{\varepsilon\}$
  - Empty set: Φ

    $L(\Phi) = \{\}$
  - What is the difference between ε and Φ?

Regular Expressions

- **Basic Operations:**
  - concatenation: ab (also a·b)

    $L(ab) = L(a) \times L(b) = \{ab\}$

  - choice/alternation: a|b

    $L(a|b) = L(a) \cup L(b) = \{a, b\}$

  - Kleene closure: a*

    $L(a^*) = L(\varepsilon) \cup L(a) \cup L(aa) \cup L(aaa) \ldots = \{\varepsilon, a, aa, aaa, \ldots\}$

  - Combinations:

    $(a|b)a = \{aa, ba\}$

    $((a|b)a)^* = \{\varepsilon, aa, ba, aaaa, aaba, baaa, baba, aaaaaa, \ldots\}$

Regular Expressions

- Example 1:
  - Regular expression: b*(abb*)*(a|ε)
  - Language: strings of a's and b's with no consecutive a's
- Example 2:
  - Language: strings of a's and b's containing consecutive a's
  - Regular expression: (a|b)*aa(a|b)*
- Precedence:
  - Highest to lowest: closure, concatenation, and choice.
  - Without precedence: a|b* = (a|b)* or a|b* = a|(b*)?
  - With precedence: a|b* = a|(b*)

Extended Notations

- One or more repetitions: $a^+ = L(aa^*) = \{a, aa, aaa, \ldots\}$
- Any character: . = {any character in the alphabet}
- Range of characters:
  - [a-z] = {all lowercase letters}
  - [a-zA-Z] = {all lowercase and uppercase letters}
- Any characters not in the alphabet:
  - ~(a|b|c) = {any character that is not a or b or c}
- Optional expression: $a? = a \mid \varepsilon$
- Strings: "a.+" = {the string itself}

Regular Expressions for Tokens

- Some token categories:

<table>
<tr><td>if</td><td>IF-TOKEN</td></tr>
<tr><td>[a-z][a-z0-9]*</td><td>ID-TOKEN</td></tr>
<tr><td>[0-9]+</td><td>NUM-TOKEN</td></tr>
<tr><td>([0-9]+”.”[0-9]*)|([0-9]*”.”[0-9]+)</td><td>REAL-TOKEN</td></tr>
<tr><td>(“--”[a-z]*”\n”)|(“ “|”\n”|”\t”)+</td><td>WHITE-SPACE</td></tr>
<tr><td>.</td><td>ERROR</td></tr>
</table>

- Disambiguation Rules:
  - Longest match: e.g., if8 is taken as an identifier
  - Rule-Order Priority: e.g., if is taken as a reserved word.

Requirements for a Scanner

- The set of patterns should form a partition of all possible token classes: mutually exclusive and exhaustive

- No two classes have any intersections
- The union of all classes fill up the whole universe

- Implication: any stream of characters can be tokenized, and each token only belongs to one class
  - A filler pattern at the end is often needed for certain scanners

Deterministic Finite Automata

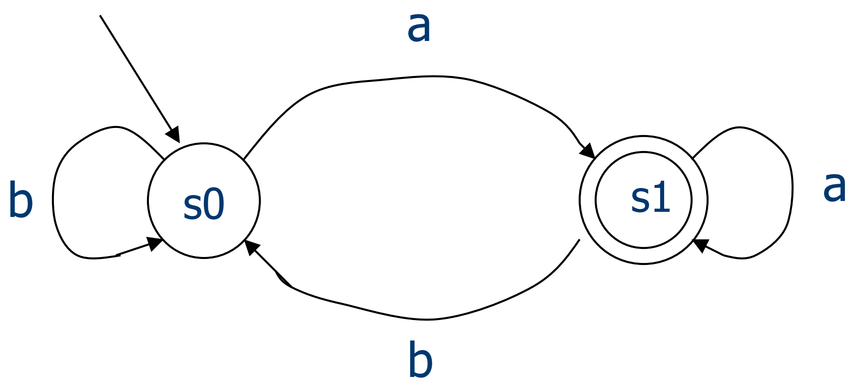
- Regular expressions do not provide a control mechanism for implementation
- Deterministic Finite Automaton (DFA): a simple machine that recognizes strings of regular sets
  - A finite set of states S
  - A finite alphabet Σ
  - A transition function T: $S \times \Sigma$ -> S
  - A specific start state s0 in S
  - A set of accepting or final states: $F \subseteq S$
- The set of strings accepted by a DFA defines a language

Graphical Representation

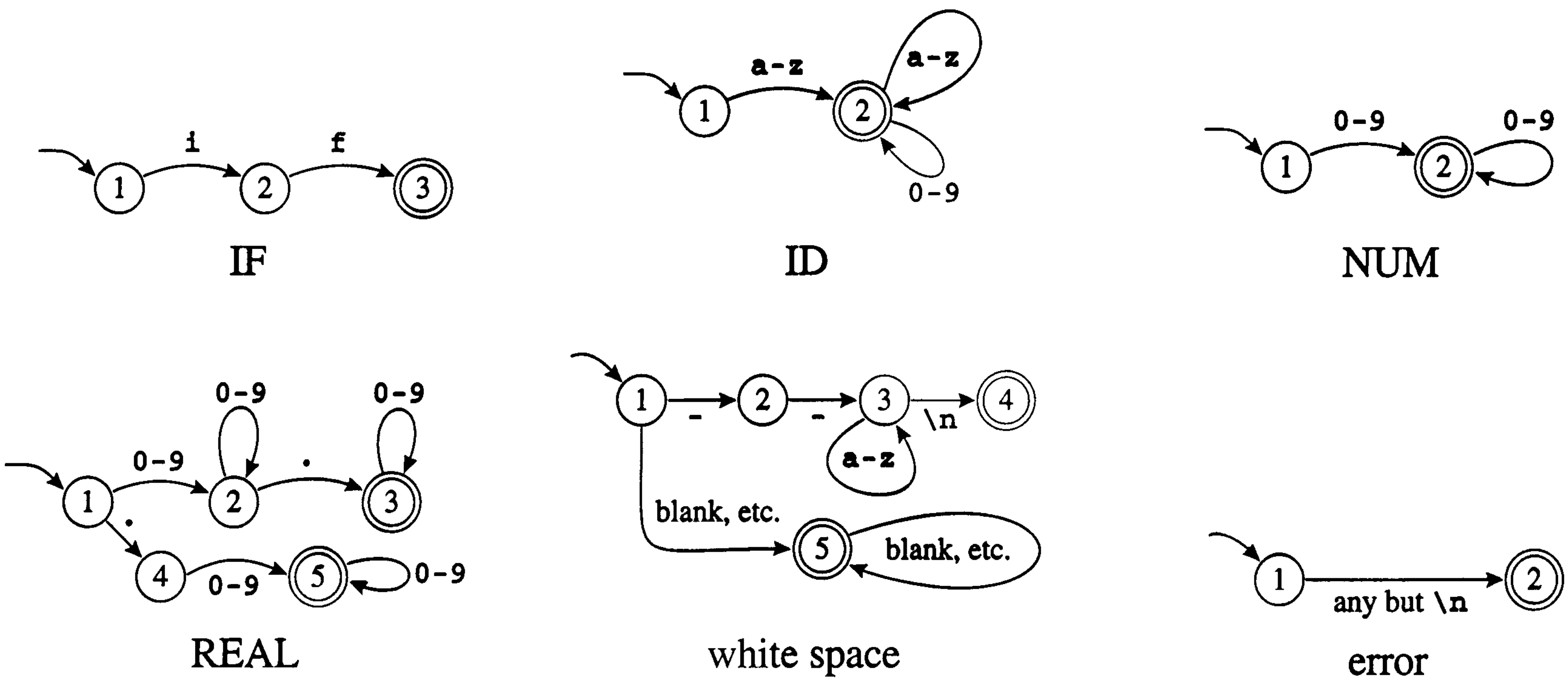
S = {s0, s1}
Σ = {a, b}
F = {s1}

T(s0, a) = s1
T(s0, b) = s0
T(s1, a) = s1
T(s1, b) = s0

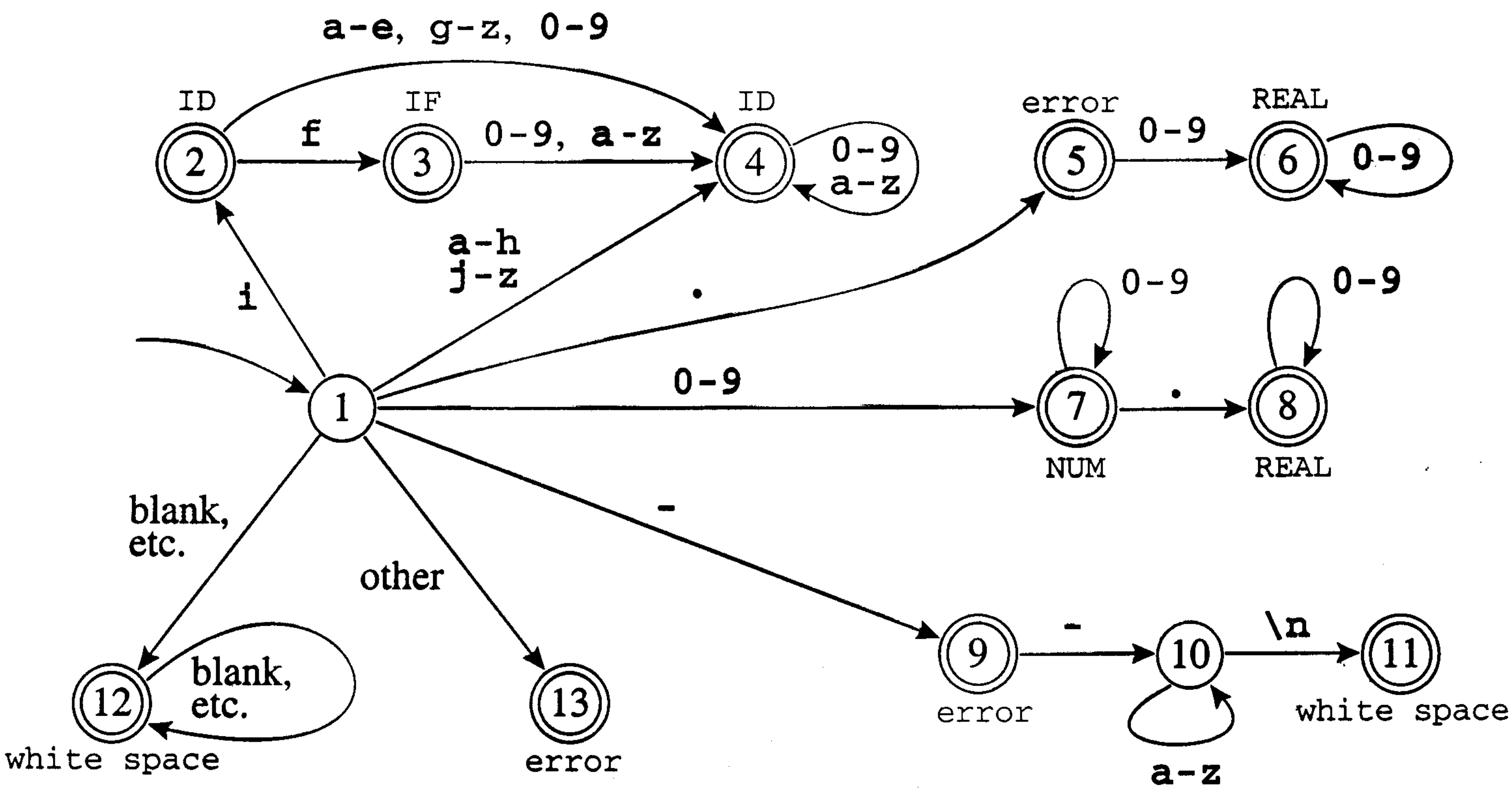
- Given the following DFA:



DFAs for Tokens

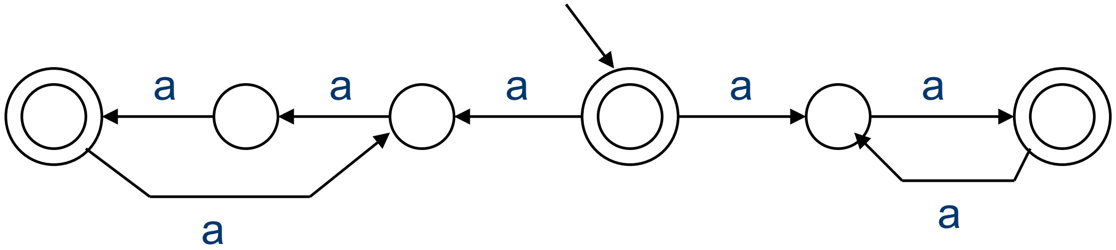


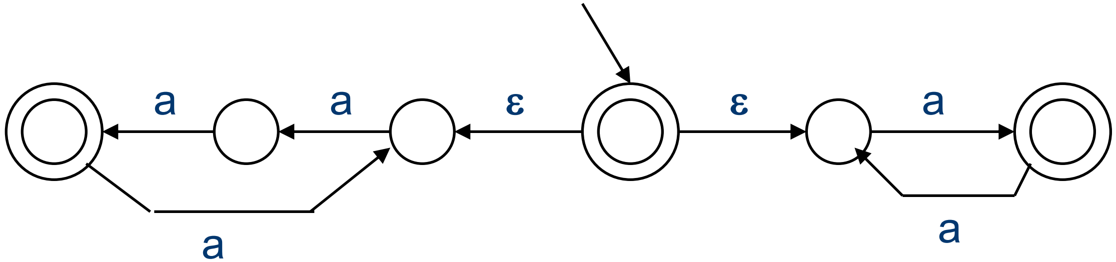
Combined DFA



Non-Deterministic Finite Automata

- An NFA is different from a DFA:
  - More than one edge from a given state can be labeled with the same input symbol.
  - Edges can be labeled with ε (transition without consuming input).

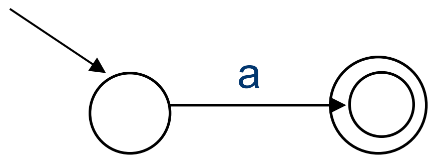




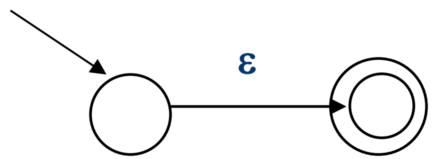
Regular Expressions to NFAs

- DFAs and NFAs have the same expressive power (i.e., accepting only regular sets).
- Why bother with NFAs?
  - Easier to convert regular expressions to NFAs
- Thompson's Construction:

a

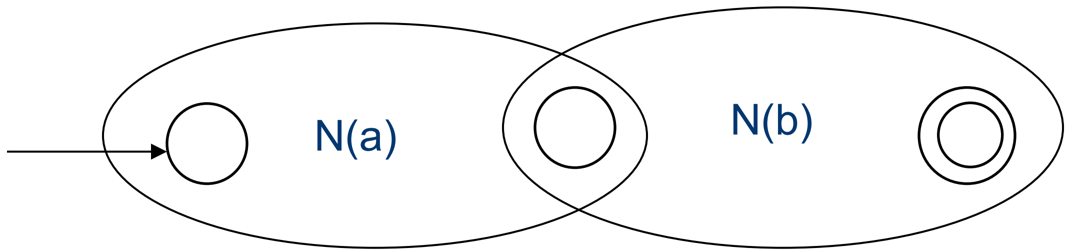


ε

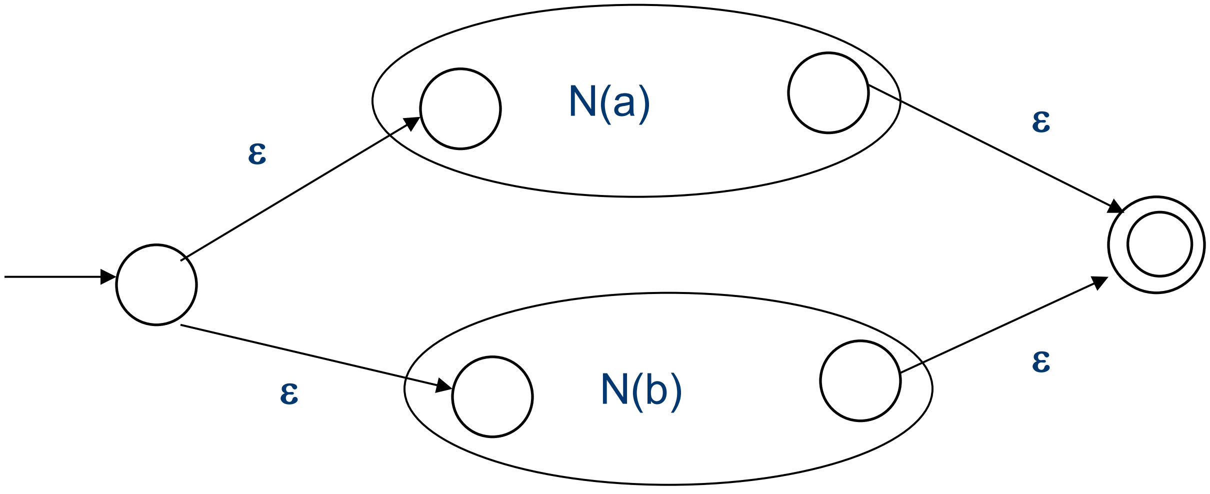


Regular Expressions and NFAs

- Concatenation: ab

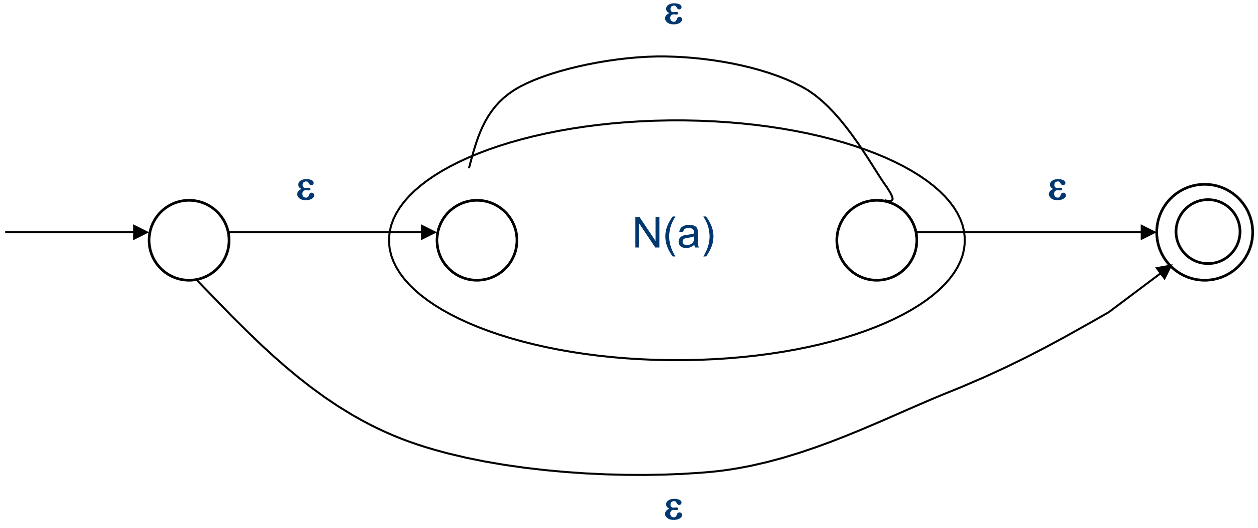


- Choice: a|b

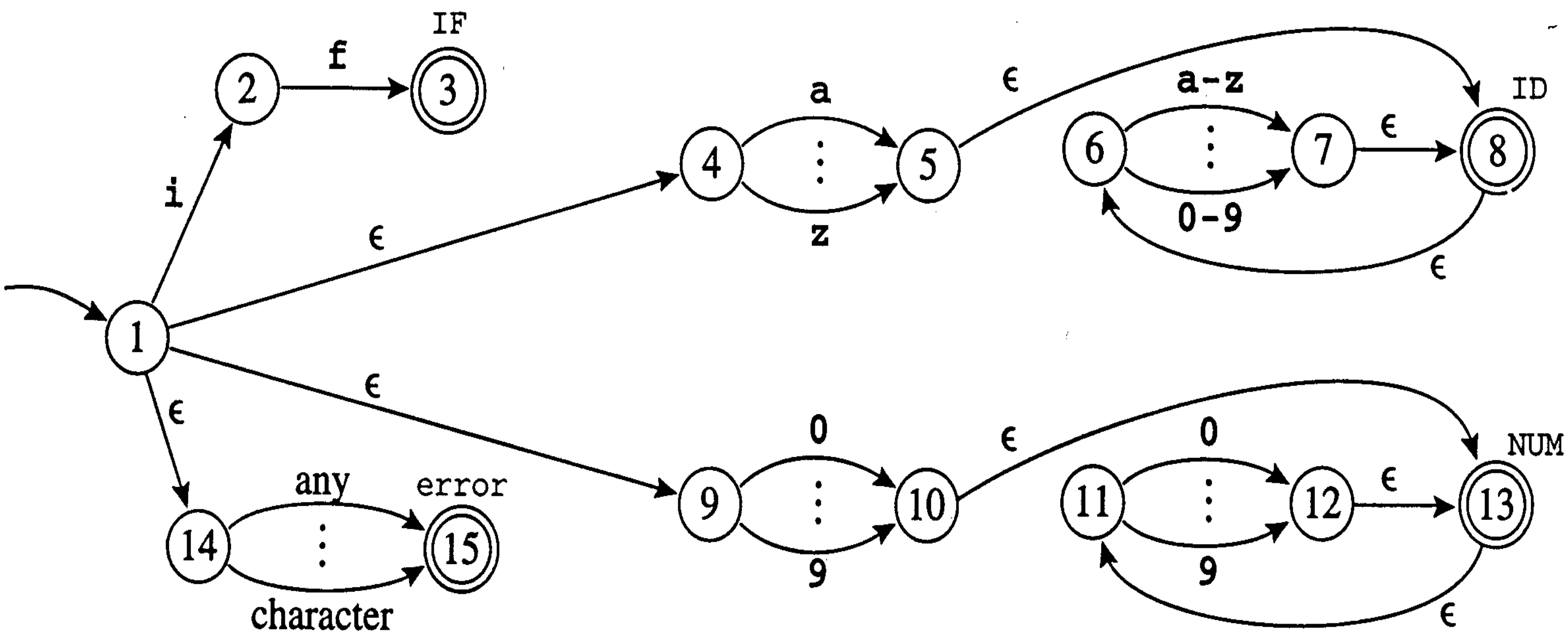


Regular Expressions and NFAs

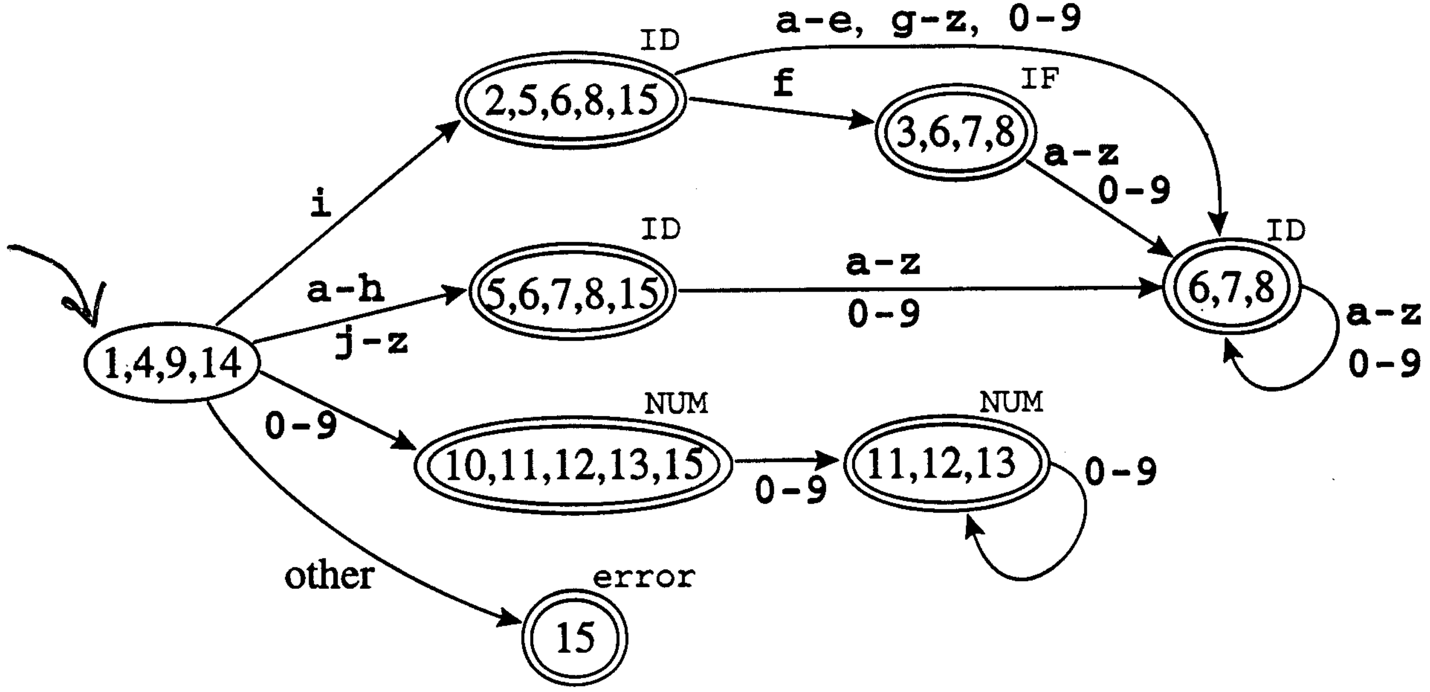
- Kleene Closure: a*



NFA for Tokens



Converted DFA

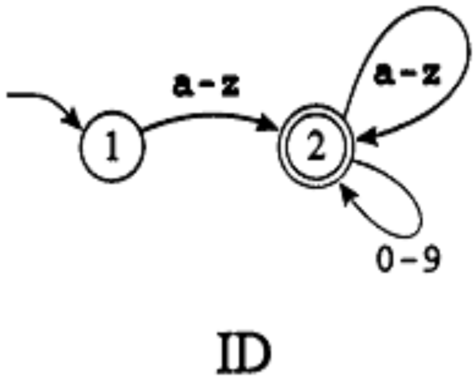


Desirable and Optimal Solution

- Formulate all the regular expressions
- Convert regular expressions into a combined NFA
- Convert the NFA to a DFA
- Minimize/optimize the DFA for the computation

DFA: Case-Based Implementation

```
state := 1;  ch := next input char;
while( state != 0 and ch != EOF ) do
    case state of
    1: case ch of
        letter:  state := 2;  ch := next input char;
        else state := 0;
    2: case ch of
        letter, digit:  state := 2;  ch := next input char;
        else state := 0;
end while;
if( ch == EOF and state is final )
    accept;
else
    report error;
```



DFA: Table-Based Implementation

```
state := 1;  ch := next input char;
while( state != 0 and ch != EOF ) do
   state := T[state, ch];
   if( state != 0 )
       ch := next input char;
end while;
if( ch == EOF and state is final )
    accept;
else
    report error;
```

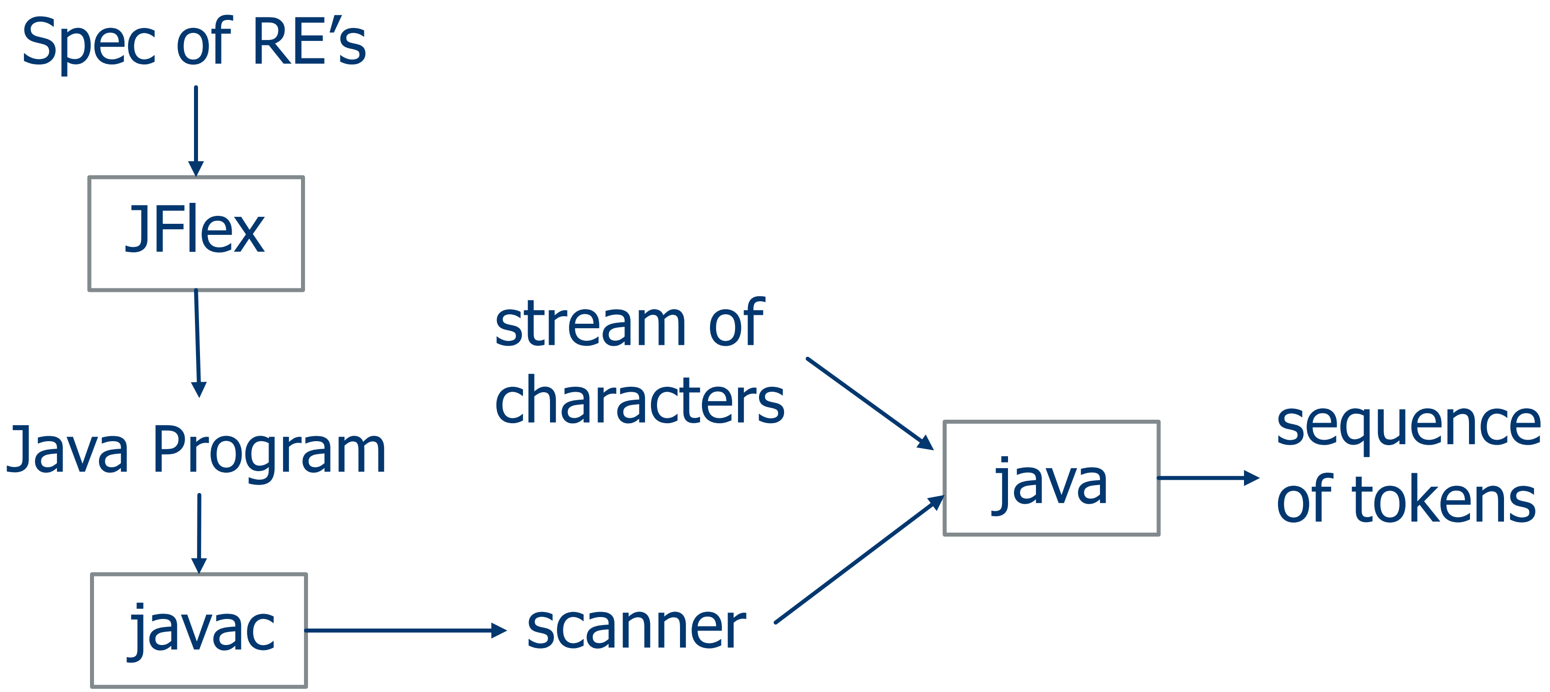
state \ input	letter	digit	other
1	2		
2	2	2	

Scanner Generator

- Writing scanners is a common requirement
  - Parsing is a ubiquitous task
- Process is repetitive, resulting in similar code structure
- Process is not hard to automate
- Scanner generators receive a specification file
  - definitions of tokens to be scanned
  - non-procedural programming: focus on what, not how
  - e.g., Lex, Flex, and JFlex.

Scanner Generator

- Automatically convert a specification file into a program that implements a scanner



Sample Jflex Specification (1/2)

```
/* JFlex specification for LISTL language */
%%

%class Lexer
%type Token
%line
%column

%eofval{
   return null;
%eofval}

digit = [0-9]
number = {digit}+
newline = \n|\r|\r\n
whitespace = [ \t]+
sign = (“+”|”-”)?
```

Sample Jflex Specification (2/2)

```
%%

{sign}{number}"."?  {
    return new Token(Token.INTEGER, yytext(), yyline, yycolumn); }
{sign}{digit}*"."{number}(e{sign}{number})?  {
    return new Token(Token.FLOAT, yytext(), yyline, yycolumn); }
{sign}{number}"/"{number}  {
    return new Token(Token.RATIO, yytext(), yyline, yycolumn); }

"("  { return new Token(Token.LPAREN, yytext(), yyline, yycolumn); }
")"  { return new Token(Token.RPAREN, yytext(), yyline, yycolumn); }
car  { return new Token(Token.CAR, yytext(), yyline, yycolumn); }
cdr  { return new Token(Token.CDR, yytext(), yyline, yycolumn); }

{newline}  { /* skip newline */ }
{whitespace}   { /* skip whitespaces */ }
[^ \t\n()]+  { return new Token(Token.SYMBOL, yytext(), yyline, yycolumn); }
```

Token Class

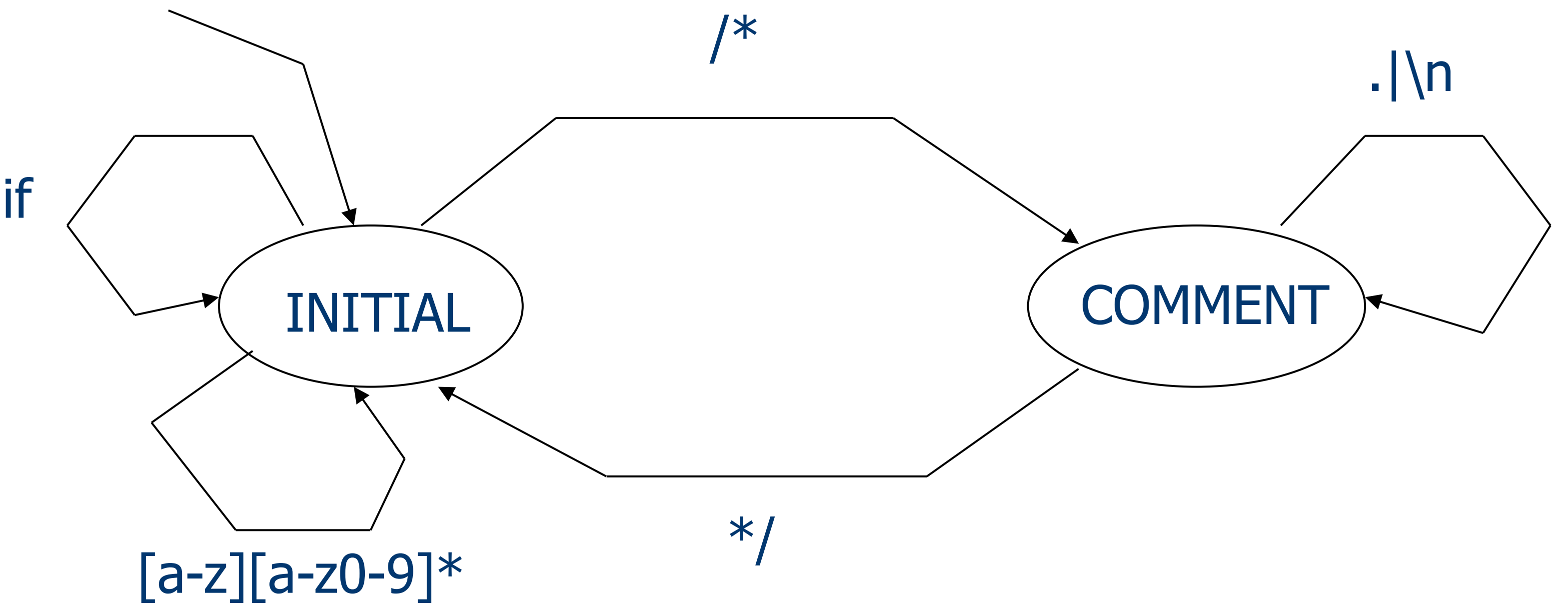
```
class Token {
   public static final int INTEGER = 0;
   public static final int FLOAT = 1;
   public static final int RATIO = 2;
   public static final int LPAREN = 3;
   public static final int RPAREN = 4;
   public static final int CAR = 5;
   public static final int CDR = 6;
   public static final int SYMBOL = 7;

   public int index;
   public String value;
   public int line;
   public int column;

   Token( int index, String value, int line, int column ) {
       this.index = index;  this.value = value;  this.line = line;  this.column = column;
   }
}
```

Macro States in Scanner Tools

- Using Macro States:



Macro States in JFlex

```
/* the usual preamble ... */

%%
/* some definitions */
%state COMMENT
/* other definitions */

%%
/* regular expressions and actions */
<YYINITIAL>if                { return symbol(sym.IF); }
<YYINITIAL>[a-z][a-z0-9]*    { return symbol(sym.ID, yytext()); }
<YYINITIAL>"/*"              { yybegin(COMMENT); }
<YYINITIAL>[ \t\n]*          { /* whitespaces */ }
<YYINITIAL>.                 { return symbol(sym.ERROR, yytext()); }
<COMMENT>"*/"                { yybegin(YYINITIAL); }
<COMMENT>.|\n                { /* skip comments */ }
```