

Lexical Analysis

Textbook: Modern Compiler Design

Chapter 2.1

<http://www.cs.tau.ac.il/~msagiv/courses/wcc11-12.html>

A motivating example

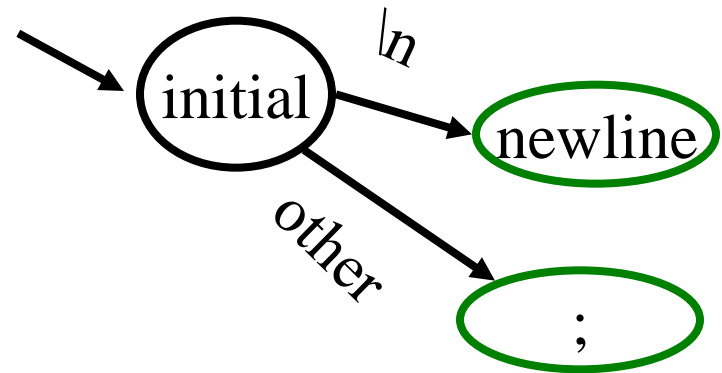
- Create a program that counts the number of lines in a given input text file

Solution (Flex)

```
        int num_lines = 0;
%%
\n    ++num_lines;
.    ;
%%
main()
{
    yylex();
    printf( "# of lines = %d\n", num_lines);
}
```

Solution(Flex)

```
int num_lines = 0;
%%
\n  ++num_lines;
.  ;
%%
main()
{
  yylex();
  printf( "# of lines = %d\n", num_lines);
}
```



JLex Spec File

User code

- Copied directly to Java file

%%

JLex directives

- Define macros, state names

%%

Lexical analysis rules

- Optional state, regular expression, action
- How to break input to tokens
- Action when token matched

Possible source
of javac errors
down the road

DIGIT= [0-9]
LETTER= [a-zA-Z]

YYINITIAL

{LETTER}
({LETTER}{DIGIT})*

File: lineCount

Jlex linecount

```
import java_cup.runtime.*;
%%
%cup
%{
    private int lineCounter = 0;
}%

%eofval{
    System.out.println("line number=" + lineCounter);
    return new Symbol(sym.EOF);
%eofval}

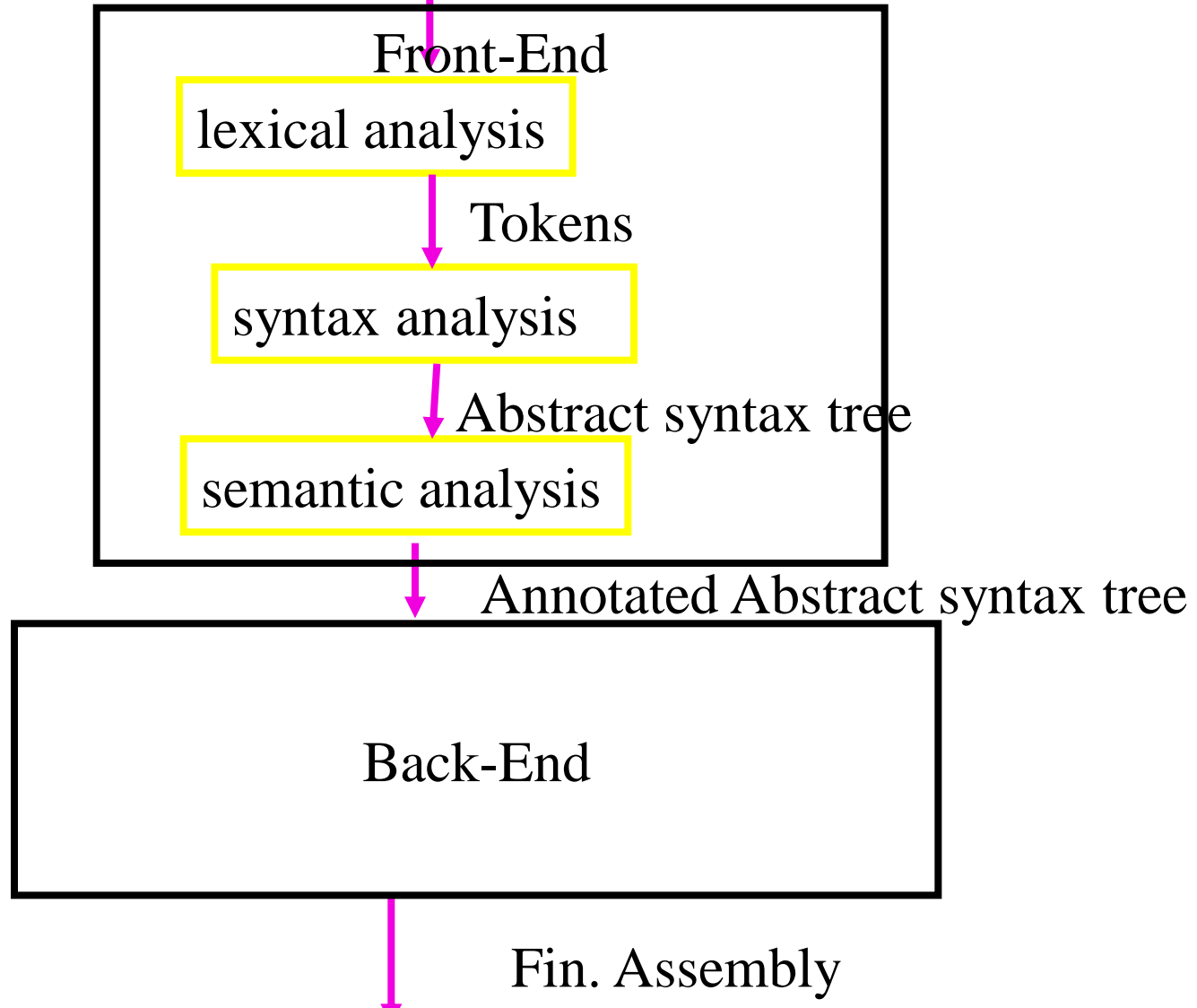
NEWLINE=\n
%%
{NEWLINE} {
    lineCounter++;
}
[^{NEWLINE}] { }
```

Outline

- Roles of lexical analysis
- What is a token
- Regular expressions
- Lexical analysis
- Automatic Creation of Lexical Analysis
- Error Handling

Basic Compiler Phases

Source program (string)



Example Tokens

Type	Examples
ID	foo n_14 last
NUM	73 00 517 082
REAL	66.1 .5 10. 1e67 5.5e-10
IF	if
COMMA	,
NOTEQ	!=
LPAREN	(
RPAREN)

Example Non Tokens

Type	Examples
comment	<code>/* ignored */</code>
preprocessor directive	<code>#include <foo.h></code>
	<code>#define NUMS 5, 6</code>
macro	<code>NUMS</code>
whitespace	<code>\t \n \b</code>

Example

```
void match0(char *s) /* find a zero */
{
    if (!strncmp(s, "0.0", 3))
        return 0. ;
}
```

```
VOID ID(match0) LPAREN CHAR Deref ID(s)
RPAREN LBRACE IF LPAREN NOT ID(strncmp)
LPAREN ID(s) COMMA STRING(0.0) COMMA NUM(3)
RPAREN RPAREN RETURN REAL(0.0) SEMI RBRACE
EOF
```

Lexical Analysis (Scanning)

- input
 - program text (file)
- output
 - sequence of tokens
- Read input file
- Identify language keywords and standard identifiers
- Handle include files and macros
- Count line numbers
- Remove whitespaces
- Report illegal symbols
- [Produce symbol table]

Why Lexical Analysis

- Simplifies the syntax analysis
 - And language definition
- Modularity
- Reusability
- Efficiency

What is a **token**?

- Defined by the programming language
- Can be separated by spaces
- Smallest units
- Defined by regular expressions

A simplified scanner for C

```
Token nextToken()
{
char c ;
loop: c = getchar();
switch (c){
    case ` `:goto loop ;
    case `;`: return SemiColumn;
    case `+`: c = getchar() ;
        switch (c) {
            case `+`: return PlusPlus ;
            case `=` return PlusEqual;
            default: ungetc(c);
                    return Plus;
        }
    case `<`:
    case `w`:
}
}
```

Regular Expressions

Basic patterns	Matching
x	The character x
.	Any character except newline
[xyz]	Any of the characters x, y, z
R?	An optional R
R*	Zero or more occurrences of R
R+	One or more occurrences of R
R ₁ R ₂	R ₁ followed by R ₂
R ₁ R ₂	Either R ₁ or R ₂
(R)	R itself

Escape characters in regular expressions

- `\` converts a single operator into text
 - `a\+`
 - `(a\+\)*`
- Double quotes surround text
 - `“a+*”`
- Esthetically ugly
- But standard

Ambiguity Resolving

- Find the longest matching token
- Between two tokens with the same length use the one declared first

The Lexical Analysis Problem

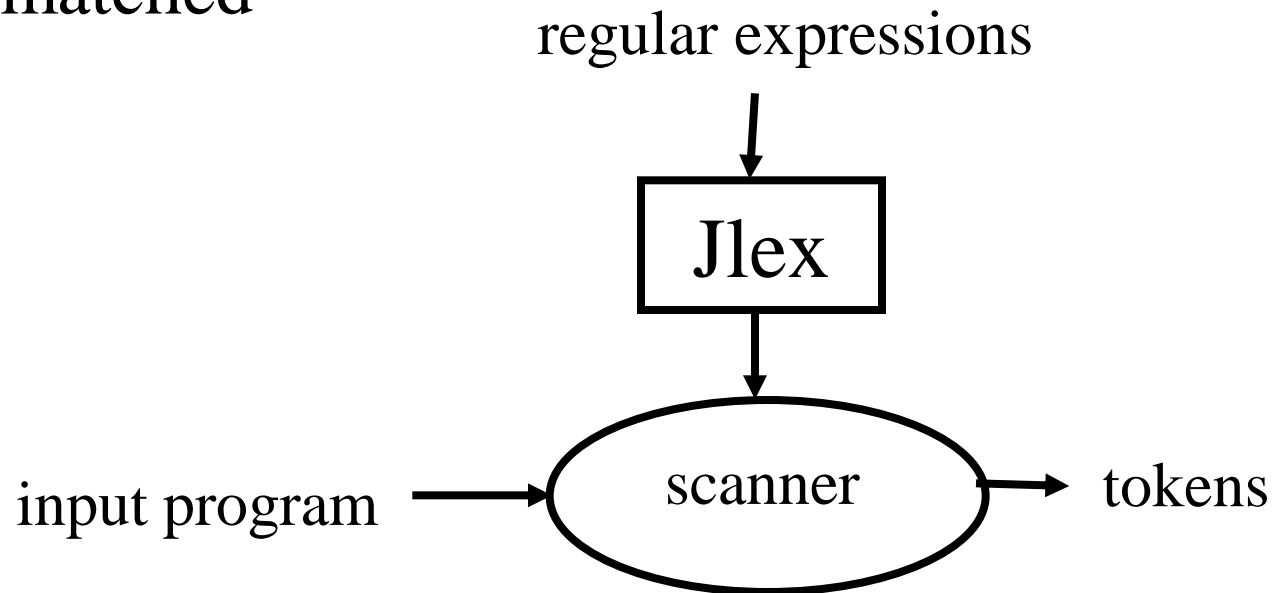
- Given
 - A set of token descriptions
 - Token name
 - Regular expression
 - An input string
- Partition the strings into tokens (class, value)
- Ambiguity resolution
 - The longest matching token
 - Between two equal length tokens select the first₁₉

A Jflex specification of C Scanner

```
import java_cup.runtime.*;
%%
%cup
% {
    private int lineNumber = 0;
% }
Letter= [a-zA-Z_]
Digit= [0-9]
%%
"\t" { }
"\n"  { lineNumber++; }
";"   { return new Symbol(sym.SemiColumn); }
"++"  { return new Symbol(sym.PlusPlus); }
"+="  { return new Symbol(sym.PlusEq); }
"+"   { return new Symbol(sym.Plus); }
"while" { return new Symbol(sym.While); }
{Letter}({Letter}|{Digit})*
    { return new Symbol(sym.Id, yytext() ); }
"<="  { return new Symbol(sym.LessOrEqual); }
"<"   { return new Symbol(sym.LessThan); }
```

Jlex

- Input
 - regular expressions and actions (Java code)
- Output
 - A scanner program that reads the input and applies actions when input regular expression is matched



How to Implement Ambiguity Resolving

- Between two tokens with the same length use the one declared first
- Find the longest matching token

Pathological Example

if	{ return IF; }
[a-z][a-z0-9]*	{ return ID; }
[0-9]+	{ return NUM; }
[0-9]"."[0-9]* [0-9]*"."[0-9]+	{ return REAL; }
(\-\-[a-z]*\n) (" "\n\t)	{ ; }
.	{ error(); }

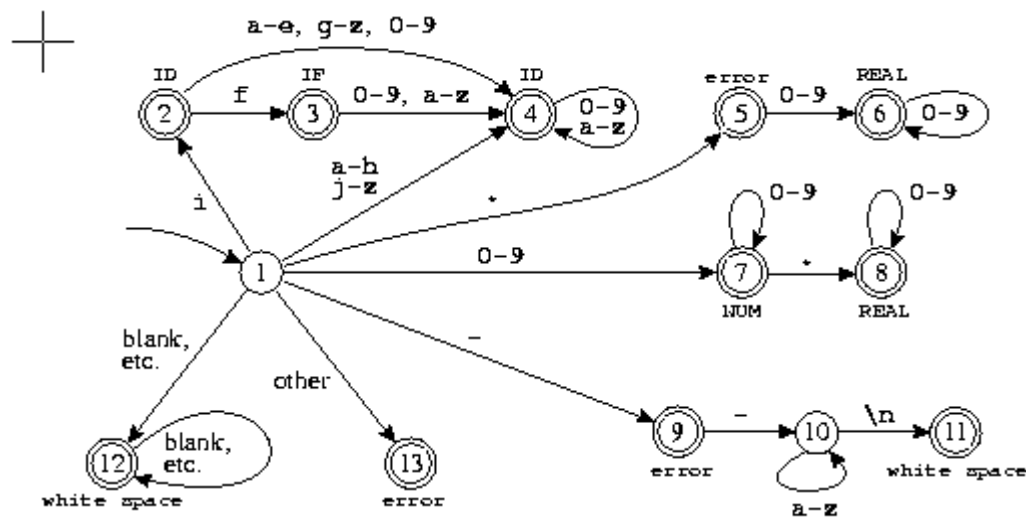


FIGURE 2.4. Combined finite automaton.
 From *Modern Compiler Implementation in ML*,
 Cambridge University Press, ©1998 Andrew W. Appel

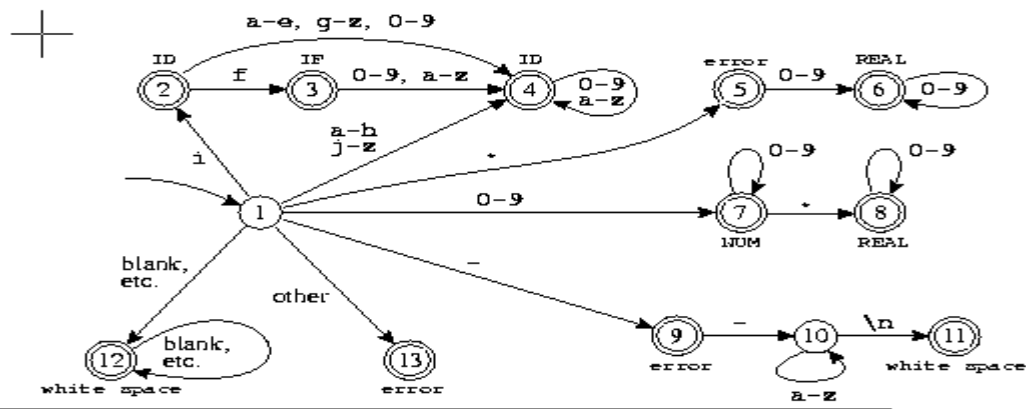


FIGURE 2.4. Combined finite automaton.
 From *Modern Compiler Implementation in ML*,
 Cambridge University Press, ©1998 Andrew W. Appel

```

int edges[][256] = { /* ..., 0, 1, 2, 3, ..., -, e, f, g, h, i, j, ... */
/* state 0 */      {0, ..., 0, 0, ..., 0, 0, 0, 0, 0, ..., 0, 0, 0, 0, 0, 0}
/* state 1 */      {13, ..., 7, 7, 7, 7, ..., 9, 4, 4, 4, 4, 2, 4, ..., 13, 13}
/* state 2 */      {0, ..., 4, 4, 4, 4, ..., 0, 4, 3, 4, 4, 4, 4, ..., 0, 0}
/* state 3 */      {0, ..., 4, 4, 4, 4, ..., 0, 4, 4, 4, 4, 4, 4, ..., 0, 0}
/* state 4 */      {0, ..., 4, 4, 4, 4, ..., 0, 4, 4, 4, 4, 4, 4, ..., 0, 0}
/* state 5 */      {0, ..., 6, 6, 6, 6, ..., 0, 0, 0, 0, 0, 0, 0, ..., 0, 0}
/* state 6 */      {0, ..., 6, 6, 6, 6, ..., 0, 0, 0, 0, 0, 0, 0, ..., 0, 0}
/* state 7 */
...
/* state 13 */     {0, ..., 0, 0, 0, 0, ..., 0, 0, 0, 0, 0, 0, 0, ..., 0, 0}

```


Pseudo Code for Scanner

```
Token nextToken()
{
  lastFinal = 0;
  currentState = 1 ;
  inputPositionAtLastFinal = input;
  currentPosition = input;
  while (not(isDead(currentState))) {
    nextState = edges[currentState][*currentPosition];
    if (isFinal(nextState)) {
      lastFinal = nextState ;
      inputPositionAtLastFinal = currentPosition; }
    currentState = nextState;
    advance currentPosition;
  }
  input = inputPositionAtLastFinal ;
  return action[lastFinal];
}
```

Example

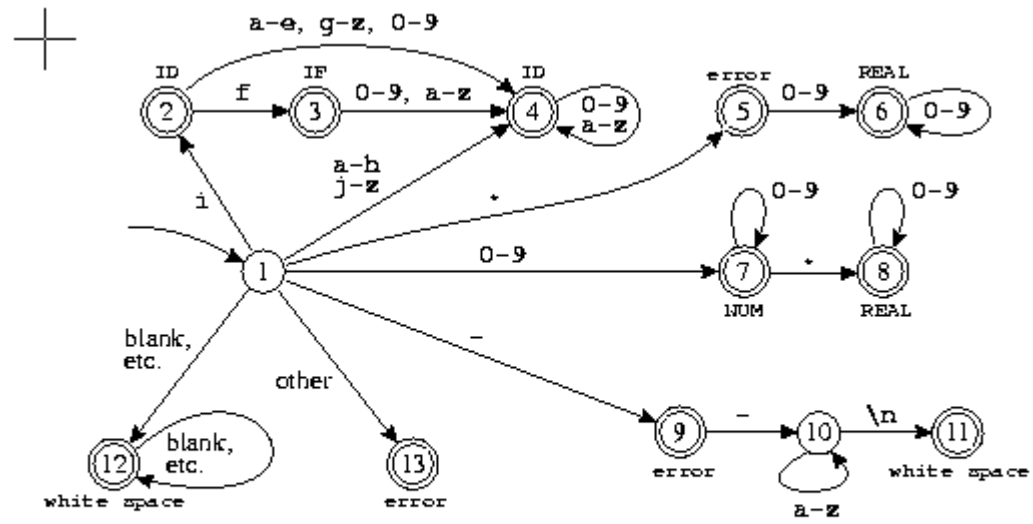


FIGURE 24. Combined finite automaton.
From *Modern Compiler Implementation in ML*,
Cambridge University Press, ©1998 Andrew W. Appel

Input: “if --not-a-com”

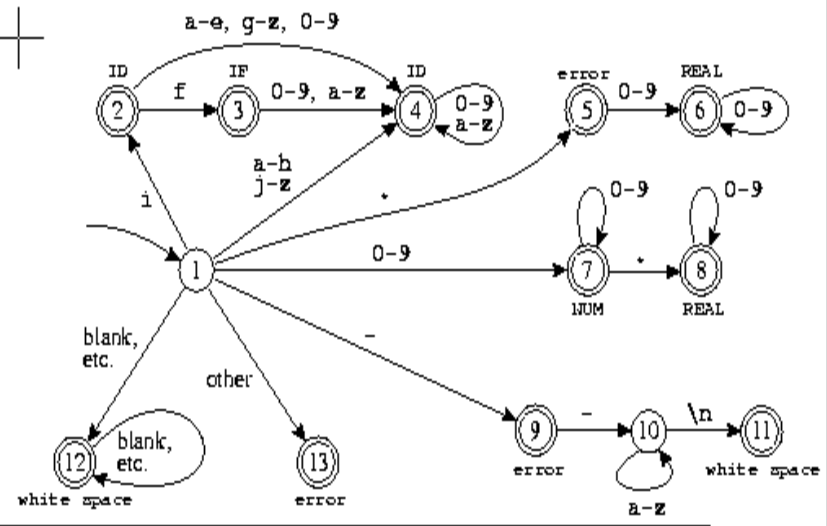


FIGURE 2.4. Combined finite automaton.
 From *Modern Compiler Implementation in ML*,
 Cambridge University Press, ©1998 Andrew W. Appel

return IF

final	state	input
0	1	if --not-a-com
2	2	if --not-a-com
3	3	if --not-a-com
3	0	if --not-a-com

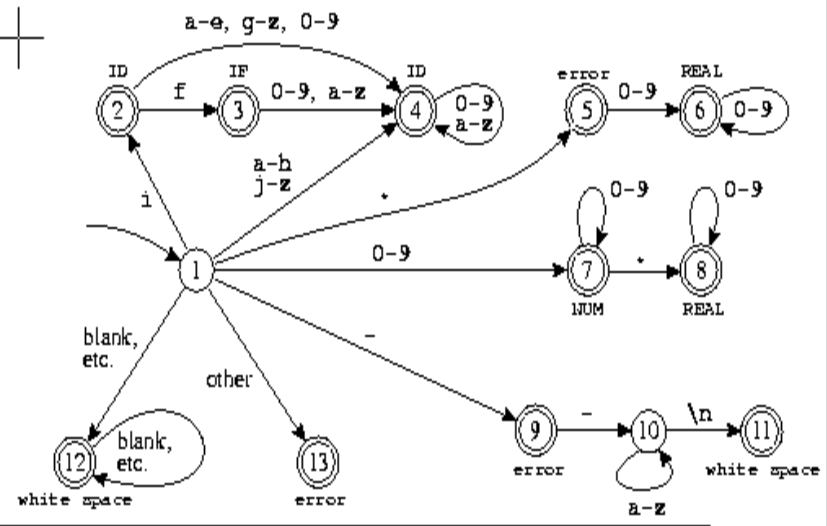


FIGURE 2.4. Combined finite automaton.
 From *Modern Compiler Implementation in ML*,
 Cambridge University Press, ©1998 Andrew W. Appel

final	state	input
0	1	--not-a-com
12	12	--not-a-com
12	0	--not-a-com

found whitespace

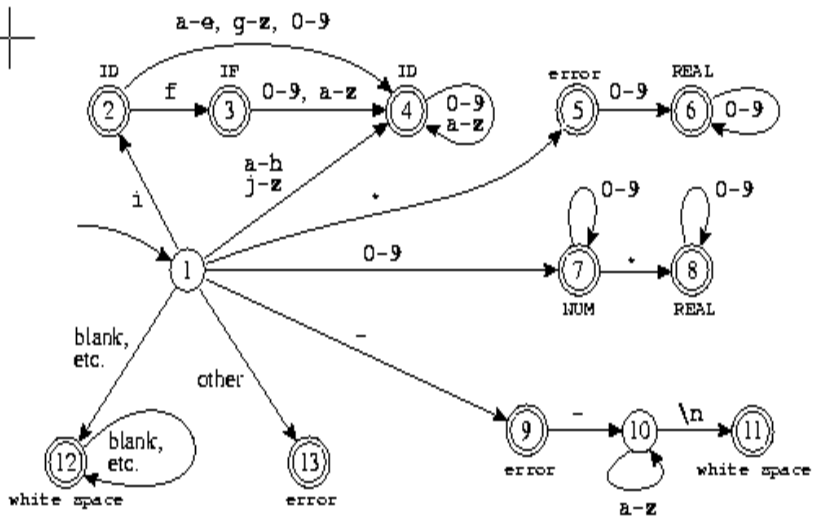


FIGURE 2.4. Combined finite automaton.
 From *Modern Compiler Implementation in ML*,
 Cambridge University Press, ©1998 Andrew W. Appel

error

final	state	input
0	1	--not-a-com
9	9	--not-a-com
9	10	--not-a-com
9	10	--not-a-com
9	10	--not-a-com
9	0	--not-a-com

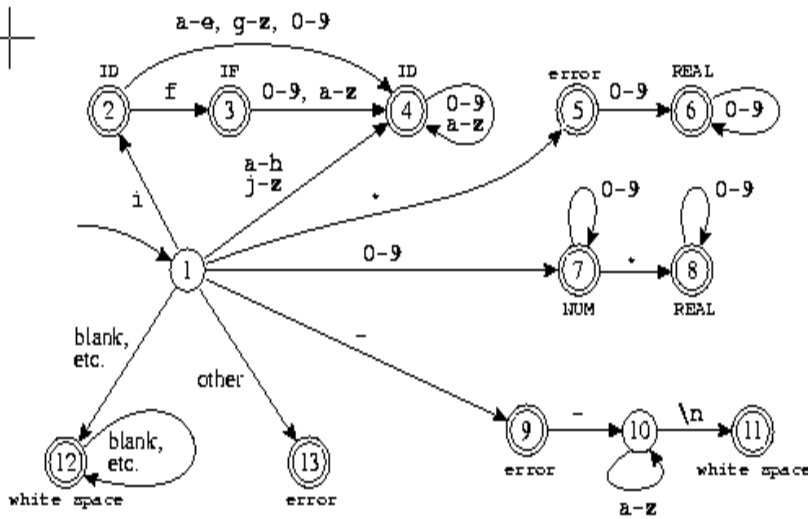


FIGURE 2.4. Combined finite automaton.
 From *Modern Compiler Implementation in ML*,
 Cambridge University Press, ©1998 Andrew W. Appel

final	state	input
0	1	not-a-com
9	9	not-a-com
9	0	not-a-com

error

Efficient Scanners

- Efficient state representation
- Input buffering
- Using switch and gotos instead of tables

Constructing Automaton from Specification

- Create a non-deterministic automaton (NFA) from every regular expression
- Merge all the automata using epsilon moves (like the $|$ construction)
- Construct a deterministic finite automaton (DFA)
 - State priority
- Minimize the automaton starting with separate accepting states

NDFA Construction

if	{ return IF; }
[a-z][a-z0-9]*	{ return ID; }
[0-9]+	{ return NUM; }

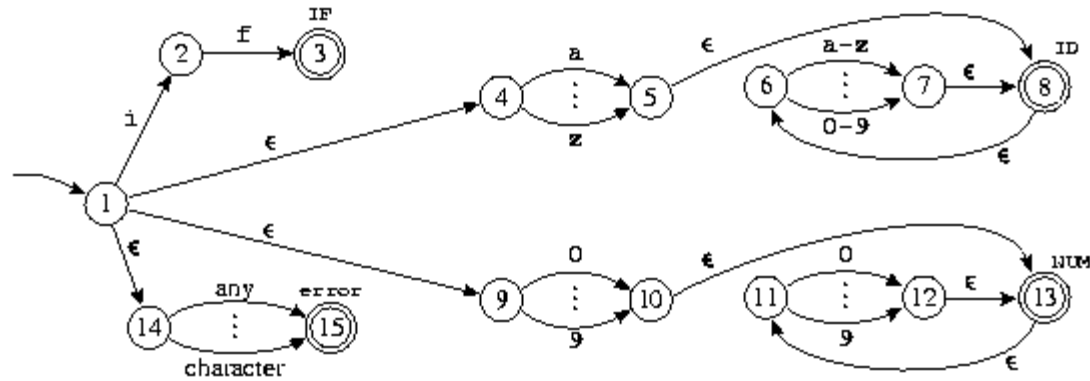


FIGURE 2.7. Four regular expressions translated to an NFA.
 From *Modern Compiler Implementation in ML*,
 Cambridge University Press, ©1998 Andrew W. Appel

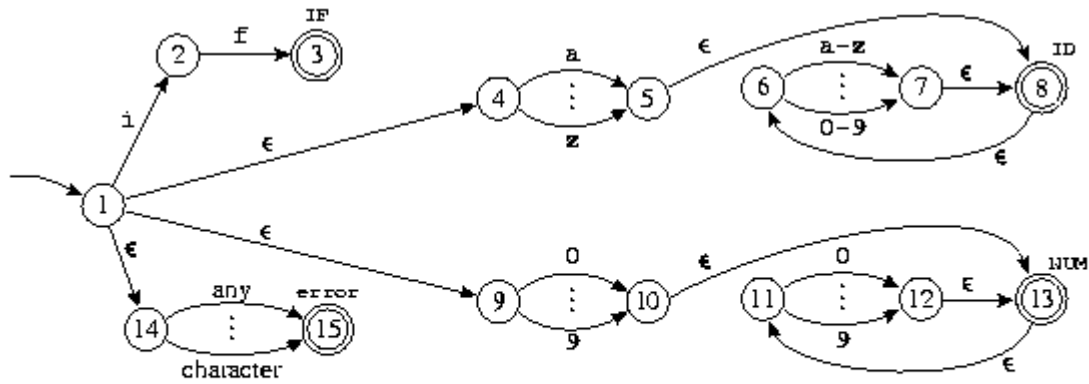


FIGURE 2.7. Four regular expressions translated to an NFA.
 From *Modern Compiler Implementation in ML*,
 Cambridge University Press, ©1998 Andrew W. Appel

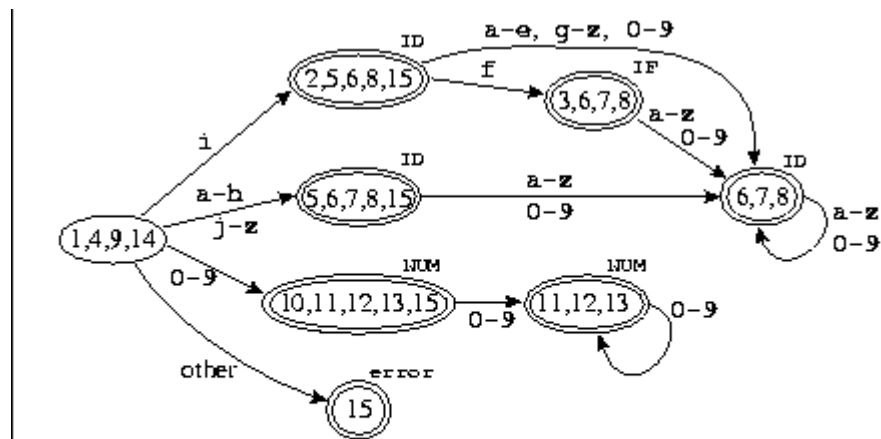


FIGURE 2.8. NFA converted to DFA.
 From *Modern Compiler Implementation in ML*,
 Cambridge University Press, ©1998 Andrew W. Appel

Minimization

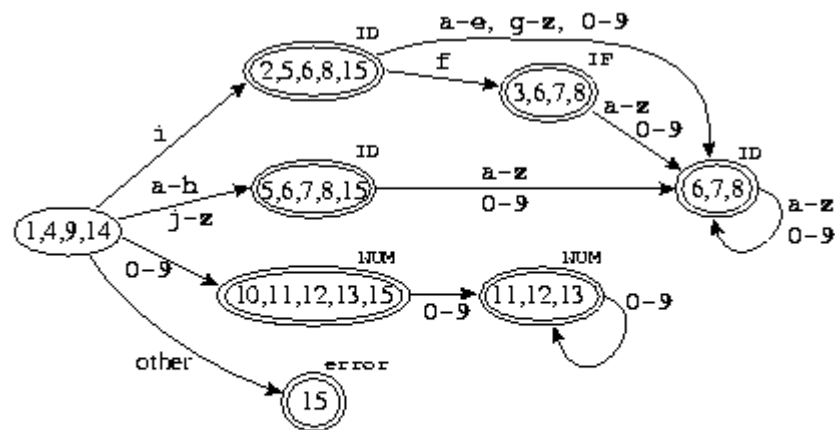


FIGURE 2.8.

NFA converted to DFA.

From *Modern Compiler Implementation in ML*,

Cambridge University Press, ©1998 Andrew W. Appel

Start States

- It may be hard to specify regular expressions for certain constructs
 - Examples
 - Strings
 - Comments
- Writing automata may be easier
- Can combine both
- Specify partial automata with regular expressions on the edges
 - No need to specify all states
 - Different actions at different states

Missing

- Creating a lexical analysis by hand
- Table compression
- Symbol Tables
- Nested Comments
- Handling Macros

Summary

- For most programming languages lexical analyzers can be easily constructed automatically
- Exceptions:
 - Fortran
 - PL/1
- Lex/Flex/Jlex are useful beyond compilers