Lexical Analysis

Lexical Phase

- Scanning
  - Deletion of comments, and compaction of consecutive white space characters into one
- Lexical Analysis
  - Complex portion, to produce tokens from the output of the scanner

Lexical Analysis

- Input
  - program text (file)
- Output
  - sequence of tokens
- Read input file
- Identify language keywords and standard identifiers

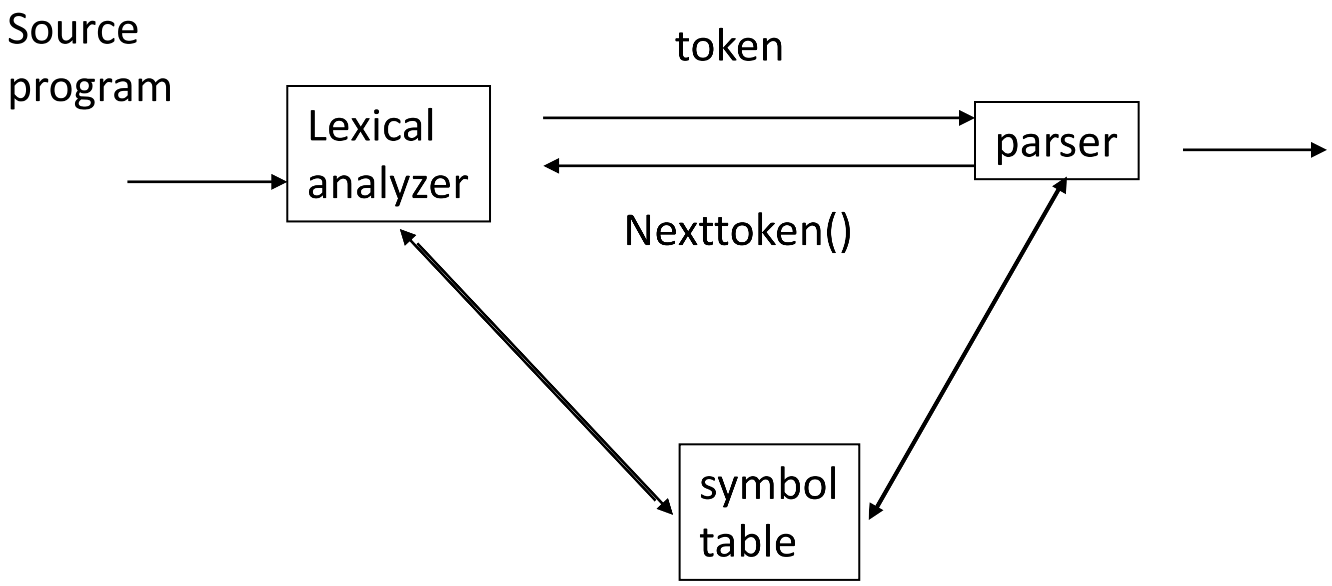
Lexical Analysis

- Handle include files and macros
- Count line numbers
- Remove whitespaces
- Report illegal symbols
- Create symbol table

Lexical Analyzer

- Lexical analyzer does not have to be an individual phase.
- But having a separate phase simplifies the design and improves the efficiency and portability.

Interaction of Lexical analyzer with parser



Why Lexical Analysis

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

Definitions

- Lexeme is a particular instant of a token.
- Token: a group of characters having a collective meaning.
  - token: identifier, lexeme: area, rate etc.
- Pattern: the rule describing how a token can be formed.
  - identifier: ([a-z]|[A-Z]) ([a-z]|[A-Z]|[0-9])*

Issues in lexical Analyzer

- How to identify tokens?
  - Patterns as RE, NFA, DFA
- How to recognize the tokens giving a token specification (how to implement the nexttoken() routine)?
  - Integrate the first two phases of the compiler

The Lexical Analysis Problem

- Given
  - A set of token descriptions
    - Token name
    - Regular expression defining the pattern for a lexeme
  - An input string
- Partition the strings into tokens
  (class, value)

Lexical Analysis problem

- Ambiguity resolution
  - The longest matching token
  - Between two equal length tokens select the first

$\Leftarrow$

$a < b$

$a <= b$

$<$ $\quad$ $<=$

$==$ $\quad$ $>$ $\quad$ $>=$

Example of Token

TOKEN	Description	Sample lexeme
if	Character i, f	If
else	Characters e, l, s, e	else
Comparison	< or > or < = or >= or == or !=	<=
id	Letter followed by letters and digits	Pi, score, a123
Number	Any numeric constant	3.14, 9.08
Literal	Anything within “ ”	“Seg fault”

Classes covering most of the tokens

- One token for each keyword. The pattern for a keyword is the same as the keyword itself.
- Tokens for the operators, either individually or in classes such as the token comparison
- One token representing all identifiers.
- One or more tokens representing constants, such as numbers and literal strings.
- Tokens for each punctuation symbol, such as left and right parentheses, comma, and semicolon.

Attributes for Tokens

- A pointer to the symbol-table entry in which the information about the token is kept

E.g E=M*C**2

<**id**, pointer to symbol-table entry for E>

<**assign_op**>

<**id**, pointer to symbol-table entry for M>

<**mult_op**>

<**id**, pointer to symbol-table entry for C>

<**exp_op**>

<**num**, integer value 2>

Lexical Errors

fi

fi

- It is hard for a lexical analyzer to tell, without the aid of other components, that there is a source-code error

else2

- Ex: fi (a == f(x)) . . .   if (condition)
- simplest recovery strategy is “panic mode" recovery

else1

else

else

- Other possible error-recovery actions are:
  - Delete one character from the remaining input.
  - Insert a missing character into the remaining input.
  - Replace a character by another character.
  - Transpose two adjacent characters

els1

abc

fi

abd

if

c

Strings and Languages

- An alphabet is any finite set of symbols
  - Typical examples of symbols are letters, digits, and punctuation
- A string over an alphabet is a finite sequence of symbols drawn from that alphabet
- The length of a string s, |s|, is the number of occurrences of symbols in s
- The empty string, denoted ϵ, is the string of length zero
- A language is any countable set of strings over some fixed alphabet

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

$\epsilon | R$

Regular expression

R^*	Zero or more occurrences of R
R^+	One or more occurrences of R
R_1R_2	R_1 followed by R_2
$R_1\|R_2$	Either R_1 or R_2
(R)	R itself

$$R^* = \bigcup_{i=0}^{\infty} R^i \qquad \{R^0, R^1, R^2, R^3, \ldots\ldots R^\infty\}$$

$$R = ab \qquad R^* = \{\epsilon, ab, abab, ababab, \ldots\ldots\}$$

$$R = a|b \qquad R^* = \{\epsilon, a, b, \underbrace{\overset{RR}{aa, ab, ba, bb}}, \ldots\ldots\}$$

$$R^+ = \bigcup_{i=1}^{\infty} R^i \qquad \{R^1, R^2, R^3, \ldots\ldots R^\infty\}$$

Properties of Regular Expression

- L(r) U L (s) is also a RE
- L(r) L(s) is also RE
- R* is also RE if R is one
- If ∑ = {a, b}, then
- L1 = a* = {ϵ, a, aa, aaa, …}
- L2 = a | b = {a, b}

Regular Expression

- Pascal language identifiers

L(r) = letter (letter | digit)*

Language for defining C language identifiers

- * has the highest precedence, followed by concatenation followed by |
- ϵ is a regular expression which is a string of length 0

Regular Definitions

- Names given to certain regular expressions and use these names later
- Regular definition is a sequence of the form

  $d1 \rightarrow r1, d2 \rightarrow r2, d3 \rightarrow r3 \ldots$

- Each di is a symbol not in the input alphabet
- Each ri is a regular expression

$d1 \rightarrow r1$ Σ

$d2 \rightarrow r2$ $\Sigma \cup d_1$

$d3 \rightarrow r3$ $\Sigma \cup d_1 \cup d_2$

Regular Definitions

- letter → A | B | …| z | _
- digit → 0 | 1 | 2 | 3 … | 9
- id → letter (letter | digit)*

id → ([a-z] | [A-Z] | _)([a-z] | [A-Z] | _ | [0-9])*

Example

1234

1

10.23 ✓

10E2

$x = y + \boxed{3.29}$

- digit → 0 | 1 | … | 9
- digits → digit digit*
- optionalFraction → .digits | ϵ
- optionalExponent → (E(+ | - | ϵ)digits) | ϵ
- number → digits optionalFraction optionalExponent

Token Recognition

if (expr) stmt   if (expr) stmt else stmt

- Stmt → if expr then Stmt | if expr then Stmt else Stmt | ϵ ✓
- expr → term relop term | term
- term → id | number
- id → letter (letter|digits)*
- relop → < | > | <= | >= | == | !=
- number → digits

if ( )
{
}

if ( )
{
}
else
{
}

if (a)

if(a>b)

if (a>5)

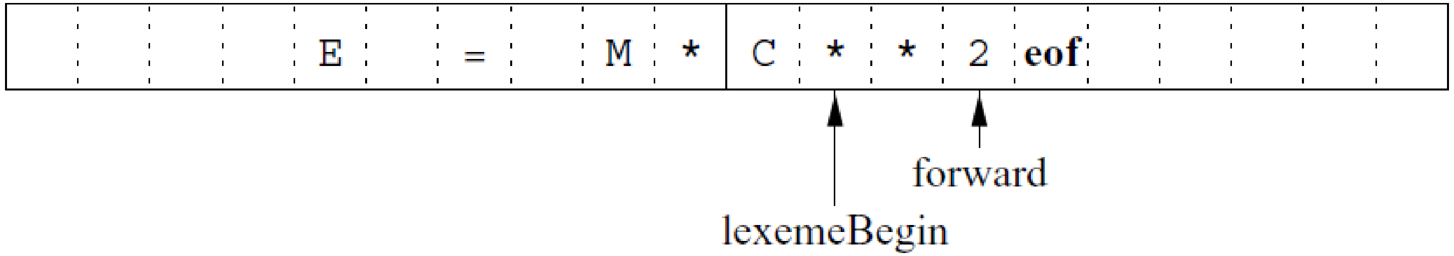
S → S ; S | S

Input Buffering

- Have to look one or more characters beyond the next lexeme before we can be sure we have the right lexeme
  - Ex: can’t determine the end of an identifier until we see a character that is not a letter or digit
  - Ex: In C, single-character operators like -, =, or < could also be the beginning of a two-character operator like ->, ==, or <=

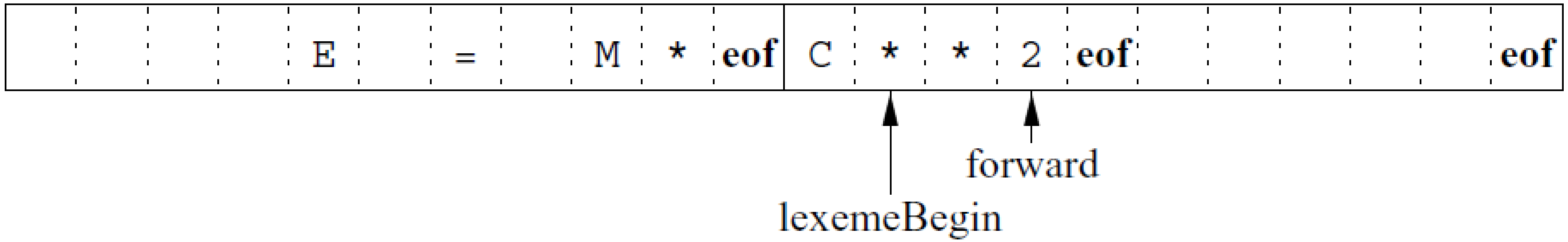
Buffer Pairs

- two buffers that are alternately reloaded
- Each buffer is of the same size N, and N is usually the size of a disk block
- Using one system read command we can read N characters into a buffer, rather than using one system call per character
- If fewer than N characters remain in the input file, then a special character, represented by eof, marks the end of the source file
- Two pointers to the input are maintained:
  - *lexemeBegin*, marks the beginning of the current lexeme, whose extent we are attempting to determine
  - *forward* scans ahead until a pattern match is found



Sentinels

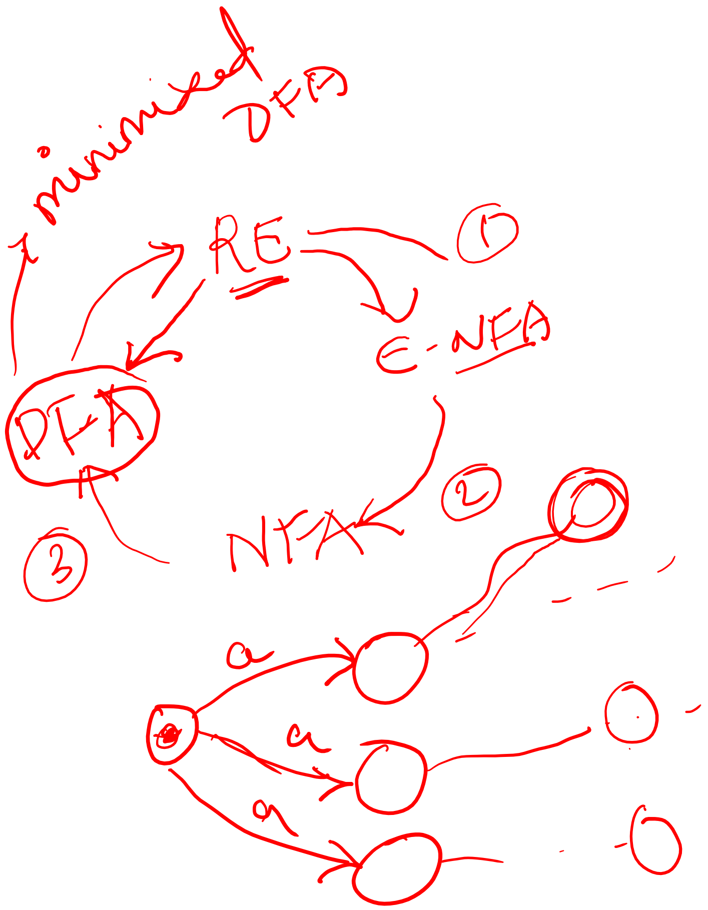
- Before advancing ***forward***, test whether end of one of the buffers is reached, if so, reload the other buffer from the input, and move ***forward*** to the beginning of the newly loaded buffer
- For each character read, we make two tests: one for the end of the buffer, and one to determine what character is read
- Can combine the buffer-end test with the test for the current character if we extend each buffer to hold a sentinel character at the end
- The sentinel is a special character that cannot be part of the source program, and a natural choice is the character eof
- Any eof that appears other than at the end of a buffer means end of input



```
switch ( *forward++ ) {
    case eof:
        if (forward is at end of first buffer ) {
            reload second buffer;
            forward = beginning of second buffer;
        }
        else if (forward is at end of second buffer ) {
            reload first buffer;
            forward = beginning of first buffer;
        }
        else /* eof within a buffer marks the end of input */
            terminate lexical analysis;
        break;
    Cases for the other characters
}
```

Why Automata?

- It may be hard to specify regular expressions for certain constructs
  - Examples
    - Strings
    - Comments
- Writing automata may be easier
- Can combine both



Why Automata?

- Specify partial automata with regular expressions on the edges
  - No need to specify all states
  - Different actions at different states

Constructing Automaton from Specification

- Create a non-deterministic automaton (NDFA) 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

Finite Automata

- By default a Deterministic one.
- Five tuple representation

  (Q, $\sum$, δ, $q0$, F), $q0$ belongs to Q and F is a subset of Q

δ is a mapping from $Q \times \sum$ to Q

- Every string has exactly one path and hence faster string matching

Non-deterministic Finite automata

- Same as deterministic, gives some flexibility.
- Five tuple representation

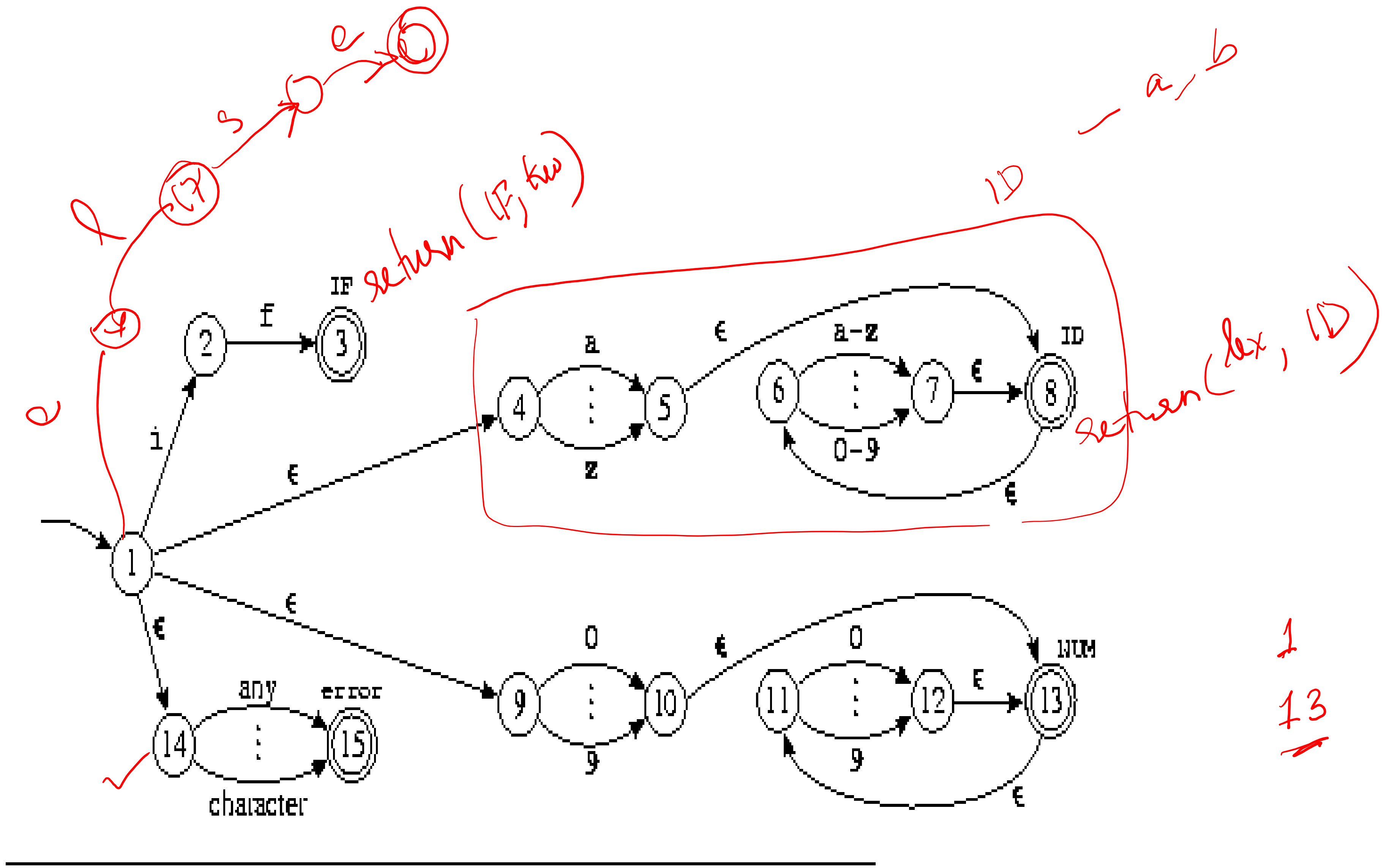
  $(Q, \sum, \delta, q0, F)$, $q0$ belongs to Q and F is a subset of Q

  δ is a mapping from $Q \times \sum$ to 2^Q
- More time for string matching as multiple paths exist.

Non-Deterministic Finite automata with ϵ

- Same as NFA. Still more flexible in allowing to change state without consuming any input symbol.
- δ is a mapping from $Q \times \sum \cup \{\epsilon\}$ to 2^Q
- Slower than NFA for string matching

Example



IF
f
i
ID
a-z
0-9
a
z
NUM
0
9
any
character
error
return (IF, kw)
return (lex, ID)
ID

Summary

- The work involved in lexical phase
- Constructing Regular expression
- Introduction to DFA, NFA and NFA with ϵ