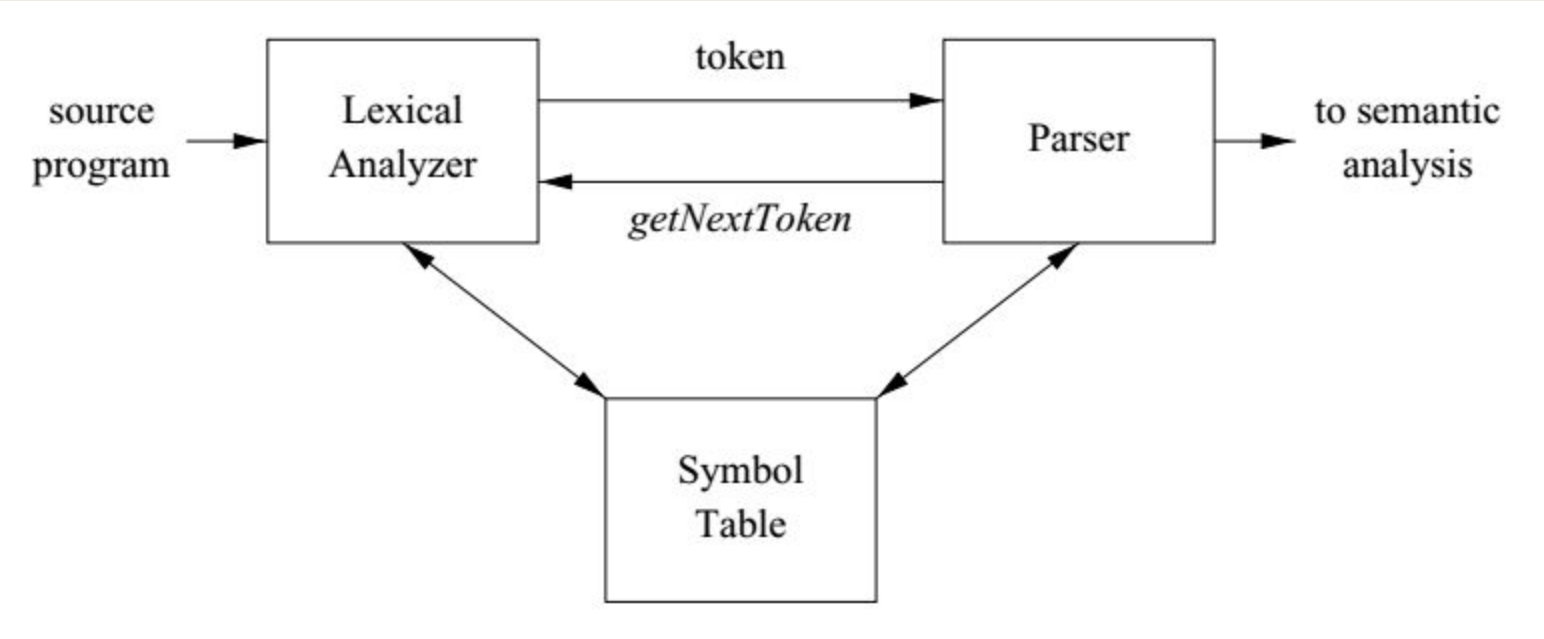
LEXICAL ANALYSIS

Overview of how a Lexical Analyzer works

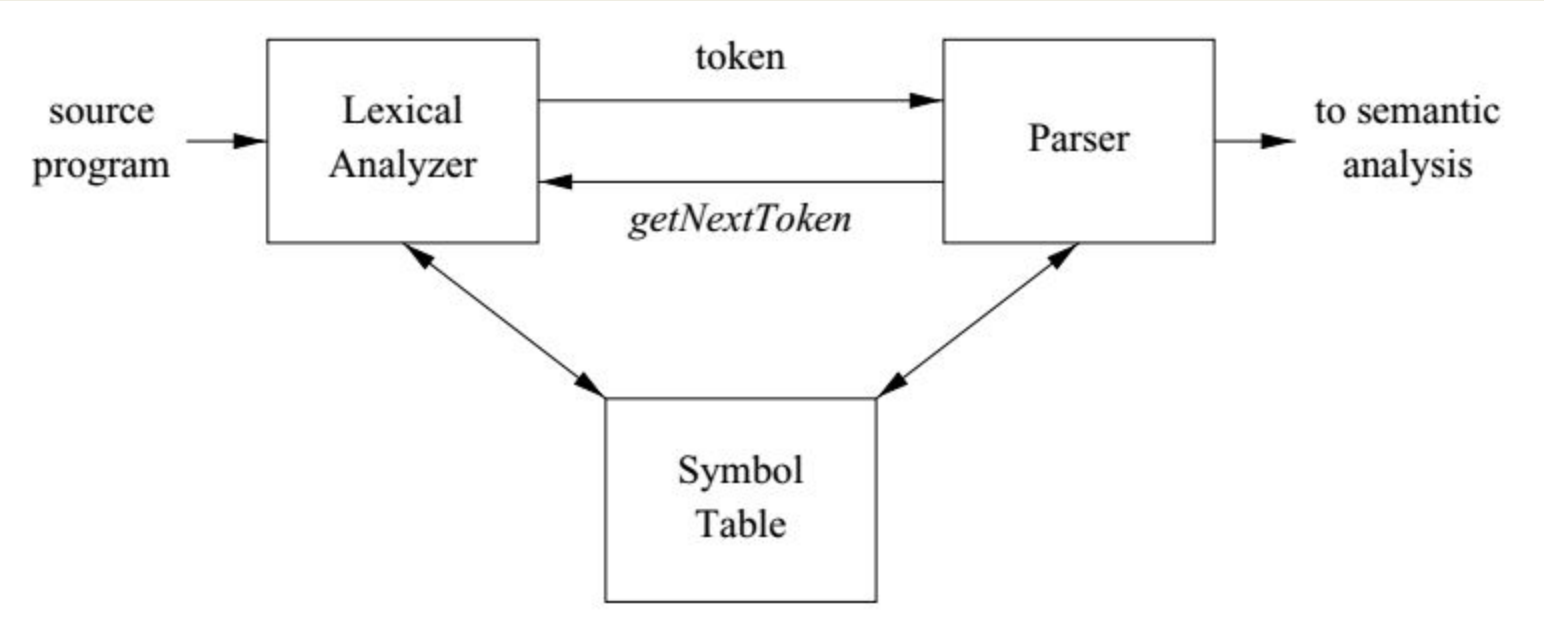
- Sometimes, lexical analyzers are divided into a cascade of two processes:
  - *Scanning consists of the simple processes that do not require tokenization of the input, such as deletion of comments and compaction of consecutive whitespace characters into one.*
  - *Lexical analysis proper is the more complex portion, which produces tokens from the output of the scanner.*

Overview of how a Lexical Analyzer works



- The main task of the lexical analyzer is to read the input characters of the source program, group them into lexemes, and produce as output a sequence of tokens for each lexeme in the source program.
- When the lexical analyzer discovers a lexeme constituting an identifier, it needs to enter that lexeme into the symbol table.

Overview of how a Lexical Analyzer works



- In some cases, information regarding the kind of identifier may be read from the symbol table by the lexical analyzer to assist it in determining the proper token it must pass to the parser.
- The call, suggested by the getNextToken command, causes the lexical analyzer to read characters from its input until it can identify the next lexeme and produce for it the next token, which it returns to the parser.

Tokens, Patterns and Lexemes

- A token is a pair consisting of a token name and an optional attribute value. The token name is an abstract symbol representing a kind of lexical unit, e.g., a particular keyword, or a sequence of input characters denoting an identifier.
- A pattern is a description of the form that the lexemes of a token may take.
- A lexeme is a sequence of characters in the source program that matches the pattern for a token and is identified by the lexical analyzer as an instance of that token.

Tokens, Patterns and Lexemes

Token	Informal Description	Sample Lexemes
if	characters `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`, `D2`
number	any numeric constant	`3.14159`, `0`, `6.02e23`
literal	anything but ", surrounded by "'s	`"core dumped"`

Attributes for Tokens

- When more than one lexeme can match a pattern, the lexical analyzer must provide the subsequent compiler phases additional information about the particular lexeme that matched.
- For example, the pattern for token number matches both 0 and 1, but it is extremely important for the code generator to know which lexeme was found in the source program.
- Thus, in many cases the lexical analyzer returns to the parser not only a token name, but an attribute value that describes the lexeme represented by the token.

Attributes for Tokens

Example 3.2 : The token names and associated attribute values for the Fortran statement

```
E = M * C ** 2
```

are written below as a sequence of pairs.

<**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**>
<**number**, integer value 2>

Symbol Table Management

- An essential function of a compiler is to record the variable names used in the source program and collect information about various attributes of each name. These attributes may provide information about the storage allocated for a name, its type, its scope (where in the program its value may be used), and in the case of procedure names, such things as the number and types of its arguments, the method of passing each argument (for example, by value or by reference), and the type returned.
- The symbol table is a data structure containing a record for each variable name, with fields for the attributes of the name. The data structure should be designed to allow the compiler to find the record for each name quickly and to store or retrieve data from that record quickly.

Lexical Errors

- It is hard for a lexical analyzer to tell, without the aid of other components, that there is a source-code error.
- For instance, if the string fi is encountered for the first time in a C program in the context:

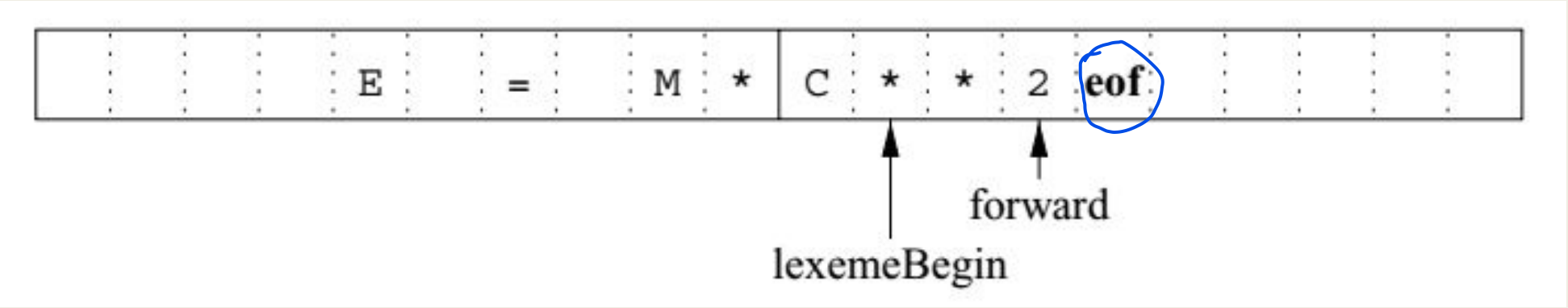
```
fi ( a == f(x)) ...
```

A lexical analyzer cannot tell whether fi is a misspelling of the keyword "**if**" or an undeclared function identifier. Since fi is a valid lexeme for the token id, the lexical analyzer must return the token id to the parser and let some other phase of the compiler, probably the parser in this case, handle an error due to transposition of the letters.

Input Buffering

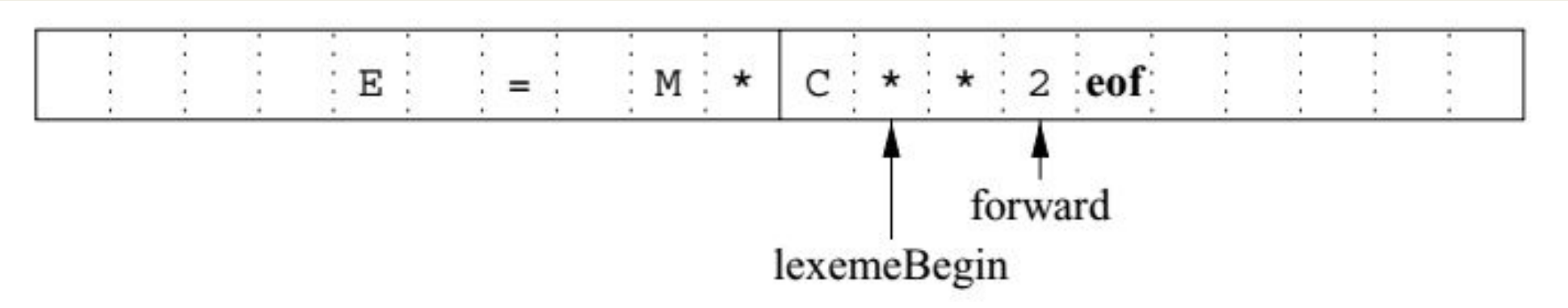
- Before discussing the problem of recognizing lexemes in the input, let us examine some ways that the simple but important task of reading the source program can be sped up.
- This task is made difficult by the fact that we often have to look one or more characters beyond the next lexeme before we can be sure we have the right lexeme. For instance, we cannot be sure we've seen the end of an identifier until we see a character that is not a letter or digit, and therefore is not part of the lexeme for id.
- Because of the amount of time taken to process characters and the large number of characters that must be processed during the compilation of a large source program, specialized buffering techniques have been developed to reduce the amount of overhead required to process a single input character.

Input Buffering



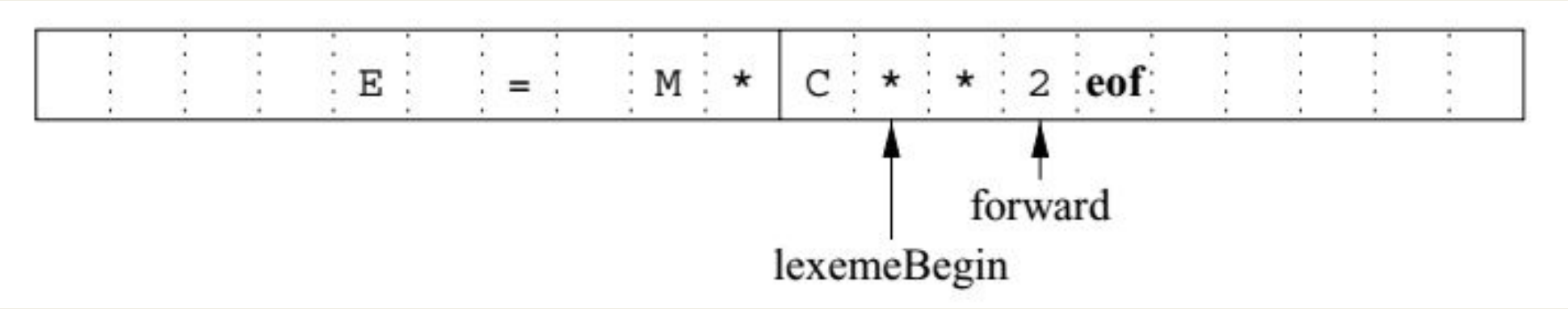
- An important scheme involves two buffers that are alternately reloaded
- Each buffer is of the same size N , and N is usually the size of a disk block, e.g., 4096 bytes. Using one system read command we can read N characters into a buffer, rather than using one system call per character. If fewer than N characters remain in the input file, then a special character, represented by **eof**, marks the end of the source file.

Input Buffering



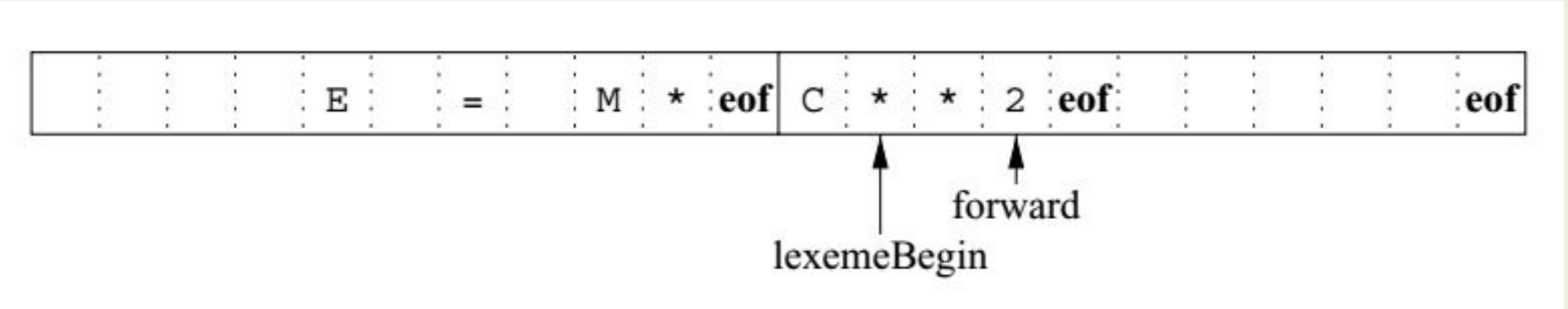
- Two pointers to the input are maintained:
  1. *Pointer **lexemeBegin**, marks the beginning of the current lexeme, whose extent we are attempting to determine.*
  2. *Pointer **forward** scans ahead until a pattern match is found.*
- Once the next lexeme is determined, **forward** is set to the character at its right end. Then, after the lexeme is recorded as an attribute value of a token returned to the parser, **lexemeBegin** is set to the character immediately after the lexeme just found.

Input Buffering



- Advancing forward requires that we first test whether we have reached the end of one of the buffers, and if so, we must reload the other buffer from the input, and move forward to the beginning of the newly loaded buffer.
- As long as we never need to look so far ahead of the actual lexeme that the sum of the lexeme's length plus the distance we look ahead is greater than N , we shall never overwrite the lexeme in its buffer before determining it.

Sentinels



- We 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 that the input is at an end.

Specification of Tokens

- Regular expressions are an important notation for specifying lexeme patterns. While they cannot express all possible patterns, they are very effective in specifying those types of patterns that we actually need for tokens.
- Let’s go through a brief overview of strings, languages and regular languages.

Strings and Languages

- An alphabet is any finite set of symbols. ASCII is an important example of an alphabet; it is used in many software systems.
- A string over an alphabet is a finite sequence of symbols drawn from that alphabet. The length of a string s, usually written |s|, is the number of occurrences of symbols in s. If x and y are strings, then the concatenation of x and y, denoted xy, is the string formed by appending y to x. For example, if x = dog and y = house, then xy = doghouse.
- A language is any countable set of strings over some fixed alphabet.

Terms for Parts of Strings

The following string-related terms are commonly used:

1. A *prefix* of string s is any string obtained by removing zero or more symbols from the end of s. For example, `ban`, `banana`, and ϵ are prefixes of `banana`.

2. A *suffix* of string s is any string obtained by removing zero or more symbols from the beginning of s. For example, `nana`, `banana`, and ϵ are suffixes of `banana`.

3. A *substring* of s is obtained by deleting any prefix and any suffix from s. For instance, `banana`, `nan`, and ϵ are substrings of `banana`.

4. The *proper* prefixes, suffixes, and substrings of a string s are those, prefixes, suffixes, and substrings, respectively, of s that are not ϵ or not equal to s itself.

5. A *subsequence* of s is any string formed by deleting zero or more not necessarily consecutive positions of s. For example, `baan` is a subsequence of `banana`.

Operations on Languages

- In lexical analysis, the most important operations on languages are union, concatenation, and closure, which are defined formally below:

Operation	Definition and Notation
Union of L and M	$L \cup M = \{s \mid s \text{ is in } L \text{ or } s \text{ is in } M\}$
Concatenation of L and M	$LM = \{st \mid s \text{ is in } L \text{ and } t \text{ is in } M\}$
Kleene closure of L	$L^* = \cup_{i=0}^{\infty} L^i$
Positive closure of L	$L^+ = \cup_{i=1}^{\infty} L^i$

Regular Expressions

- Each regular expression r denotes a language $L(r)$, which is also defined recursively from the languages denoted by r's subexpressions.
- ε is a regular expression, and $L(\varepsilon)$ is $\{\varepsilon\}$, that is, the language whose sole member is the empty string.
- If *a* is a symbol in the (given) alphabet, then *a* is a regular expression, and *L(a)* = {*a*}.
- $(r)\,|\,(s)$ is a regular expression denoting the language $L(r) \cup L(s)$.
- $(r)(s)$ is a regular expression denoting the language $L(r)L(s)$.
- $(r)^*$ is a regular expression denoting $\big(L(r)\big)^*$.
- (r) is a regular expression denoting $L(r)$.

Regular Expressions

Example 3.4: Let $\Sigma = \{a, b\}$.

1. The regular expression $\mathbf{a}|\mathbf{b}$ denotes the language $\{a, b\}$.
2. $(\mathbf{a}|\mathbf{b})(\mathbf{a}|\mathbf{b})$ denotes $\{aa, ab, ba, bb\}$, the language of all strings of length two over the alphabet Σ. Another regular expression for the same language is $\mathbf{aa}|\mathbf{ab}|\mathbf{ba}|\mathbf{bb}$.
3. $\mathbf{a}^*$ denotes the language consisting of all strings of zero or more a's, that is, $\{\epsilon, a, aa, aaa, \ldots\}$.
4. $(\mathbf{a}|\mathbf{b})^*$ denotes the set of all strings consisting of zero or more instances of a or b, that is, all strings of a's and b's: $\{\epsilon, a, b, aa, ab, ba, bb, aaa, \ldots\}$. Another regular expression for the same language is $(\mathbf{a}^*\mathbf{b}^*)^*$.
5. $\mathbf{a}|\mathbf{a}^*\mathbf{b}$ denotes the language $\{a, b, ab, aab, aaab, \ldots\}$, that is, the string a and all strings consisting of zero or more a's and ending in b.

Regular Expressions

LAW	DESCRIPTION
$r\|s = s\|r$	$\|$ is commutative
$r\|(s\|t) = (r\|s)\|t$	$\|$ is associative
$r(st) = (rs)t$	Concatenation is associative
$r(s\|t) = rs\|rt;\ (s\|t)r = sr\|tr$	Concatenation distributes over $\|$
$\epsilon r = r\epsilon = r$	ϵ is the identity for concatenation
$r^* = (r\|\epsilon)^*$	ϵ is guaranteed in a closure
$r^{**} = r^*$	$*$ is idempotent

Figure 3.7: Algebraic laws for regular expressions

Regular Expressions

- For notational convenience, we may wish to give names to certain regular expressions and use those names in subsequent expressions, as if the names were themselves symbols.

Example 3.5: C identifiers are strings of letters, digits, and underscores. Here is a regular definition for the language of C identifiers. We shall conventionally use italics for the symbols defined in regular definitions.

$$
\begin{aligned}
\mathit{letter\_} &\rightarrow \texttt{A} \mid \texttt{B} \mid \cdots \mid \texttt{Z} \mid \texttt{a} \mid \texttt{b} \mid \cdots \mid \texttt{z} \mid \texttt{\_} \\
\mathit{digit} &\rightarrow \texttt{0} \mid \texttt{1} \mid \cdots \mid \texttt{9} \\
\mathit{id} &\rightarrow \mathit{letter\_} \, (\, \mathit{letter\_} \mid \mathit{digit} \,)^*
\end{aligned}
$$

Regular Expressions

- For notational convenience, we may wish to give names to certain regular expressions and use those names in subsequent expressions, as if the names were themselves symbols.

Example 3.6: Unsigned numbers (integer or floating point) are strings such as `5280`, `0.01234`, `6.336E4`, or `1.89E-4`. The regular definition

$$
\begin{array}{rcl}
\mathit{digit} & \rightarrow & \texttt{0} \mid \texttt{1} \mid \cdots \mid \texttt{9} \\
\mathit{digits} & \rightarrow & \mathit{digit}\ \mathit{digit}^{*} \\
\mathit{optionalFraction} & \rightarrow & \texttt{.}\ \mathit{digits} \mid \epsilon \\
\mathit{optionalExponent} & \rightarrow & (\ \texttt{E}\ (\ \texttt{+} \mid \texttt{-} \mid \epsilon\ )\ \mathit{digits}\ ) \mid \epsilon \\
\mathit{number} & \rightarrow & \mathit{digits}\ \mathit{optionalFraction}\ \mathit{optionalExponent}
\end{array}
$$

Some extra notations that are used for convenience

- $(r)^+$ is a regular expression denoting $\big(L(r)\big)^+$. (One or more instances)
- $(r)?$ is a regular expression denoting $L(r) \cup \{\varepsilon\}$. (Zero or one occurrence)
- A regular expression $a_1|a_2|\cdots|a_n$, where the a_i's are each symbols of the alphabet, can be replaced by the shorthand $[a_1 a_2 \cdots a_n]$. More importantly, when $a_1, a_2, \ldots, a_n$ form a logical sequence, e.g., consecutive uppercase letters, lowercase letters, or digits, we can replace them by $[a_1 - a_n]$. Thus, $[abc]$ is shorthand for $a|b|c$, and $[a - z]$ is shorthand for $a|b|\cdots|z$.

Some extra notations that are used for convenience

Example 3.7 : Using these shorthands, we can rewrite the regular definition of Example 3.5 as:

$$
\begin{array}{rcl}
\mathit{letter\_} & \rightarrow & \texttt{[A-Za-z\_]} \\
\mathit{digit} & \rightarrow & \texttt{[0-9]} \\
\mathit{id} & \rightarrow & \mathit{letter\_}\ (\ \mathit{letter\_} \mid \mathit{digit}\ )^{*}
\end{array}
$$

The regular definition of Example 3.6 can also be simplified:

$$
\begin{array}{rcl}
\mathit{digit} & \rightarrow & \texttt{[0-9]} \\
\mathit{digits} & \rightarrow & \mathit{digit}^{+} \\
\mathit{number} & \rightarrow & \mathit{digits}\ (\texttt{.}\ \mathit{digits})?\ (\ \texttt{E}\ \texttt{[+-]}?\ \mathit{digits}\ )?
\end{array}
$$

Finding regular expressions

Exercise 3.3.5 : Write regular definitions for the following languages:

a) All strings of lowercase letters that contain the five vowels in order.

b) All strings of lowercase letters in which the letters are in ascending lexicographic order.

c) Comments, consisting of a string surrounded by /* and */, without an intervening */, unless it is inside double-quotes (").

Recognition of Tokens

- The terminals of the grammar, which are **if**, **then**, **else**, **relop**, **id**, and **number**, are the names of tokens as far as the lexical analyzer is concerned. The patterns for these tokens are described using regular definitions given below:

digit	→	[0-9]
digits	→	*digit*$^+$
number	→	*digits* (. *digits*)? (E [+-]? *digits*)?
letter	→	[A-Za-z]
id	→	*letter* (*letter* | *digit*)*
if	→	if
then	→	then
else	→	else
relop	→	< | > | <= | >= | = | <>

Figure 3.11: Patterns for tokens of Example 3.8

Recognition of Tokens

- To simplify matters, we make the common assumption that keywords are also reserved words : that is, they are not identifiers, even though their lexemes match the pattern for identifiers.
  In addition, we assign the lexical analyzer the job of stripping out whitespace, by recognizing the token **ws** defined by:

$$ws \rightarrow (\ \mathbf{blank} \mid \mathbf{tab} \mid \mathbf{newline}\ )^{+}$$

- Here, **blank**, **tab**, and **newline** are abstract symbols that we use to express the ASCII characters of the same names. Token **ws** is different from the other tokens in that, when we recognize it, we do not return it to the parser, but rather restart the lexical analysis from the character that follows the whitespace.

Recognition of Tokens

LEXEMES	TOKEN NAME	ATTRIBUTE VALUE
Any *ws*	–	–
`if`	**if**	–
`then`	**then**	–
`else`	**else**	–
Any *id*	**id**	Pointer to table entry
Any *number*	**number**	Pointer to table entry
`<`	**relop**	`LT`
`<=`	**relop**	`LE`
`=`	**relop**	`EQ`
`<>`	**relop**	`NE`
`>`	**relop**	`GT`
`>=`	**relop**	`GE`

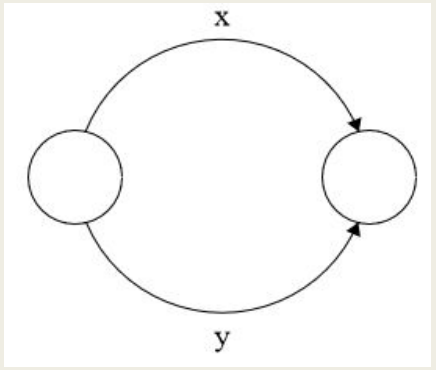
Transition Diagrams

- As an intermediate step in the construction of a lexical analyzer, we first convert patterns into stylized graphs or flowcharts, called “transition diagrams."
- Transition diagrams have a collection of nodes or circles, called states. Each state represents a condition that could occur during the process of scanning the input looking for a lexeme that matches one of several patterns.
- Edges are directed from one state of the transition diagram to another. Each edge is labeled by a symbol or set of symbols. If we are in some state s, and the next input symbol is a, we look for an edge out of state s labeled by a. If we find such an edge, we advance the forward pointer and enter the state of the transition diagram to which that edge leads.
- We have a start state from where we begin and a set of final states ending in which accepts the string.

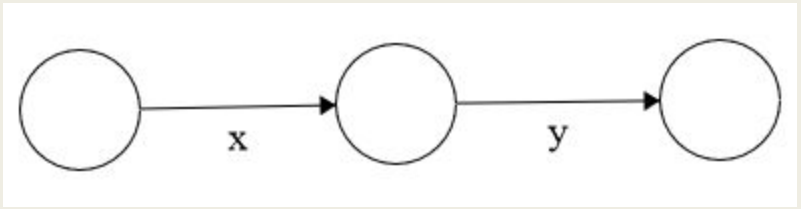
Transition Diagrams

- Let's convert regular expressions to DFAs.

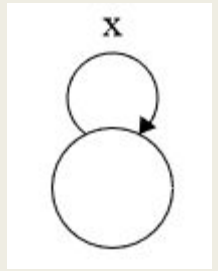
$x|y$



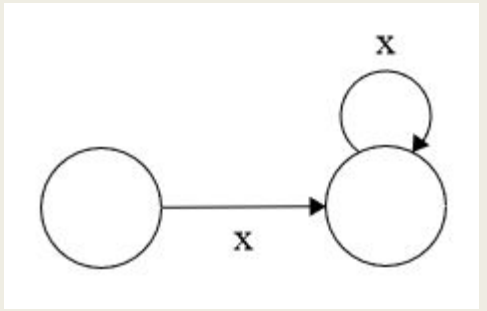
xy



x^*



x^+



Transition Diagrams

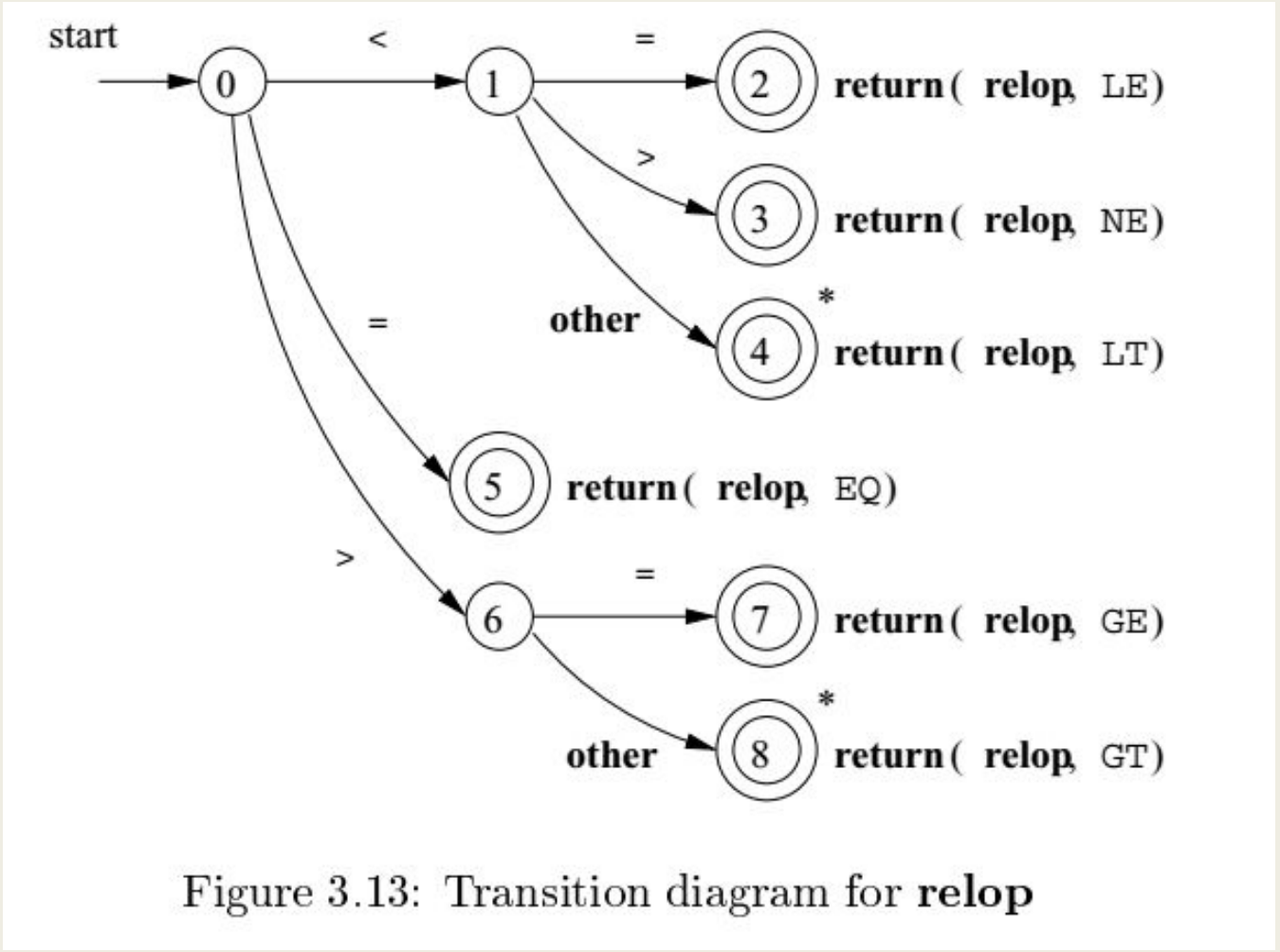


Figure 3.13: Transition diagram for **relop**

Transition Diagrams

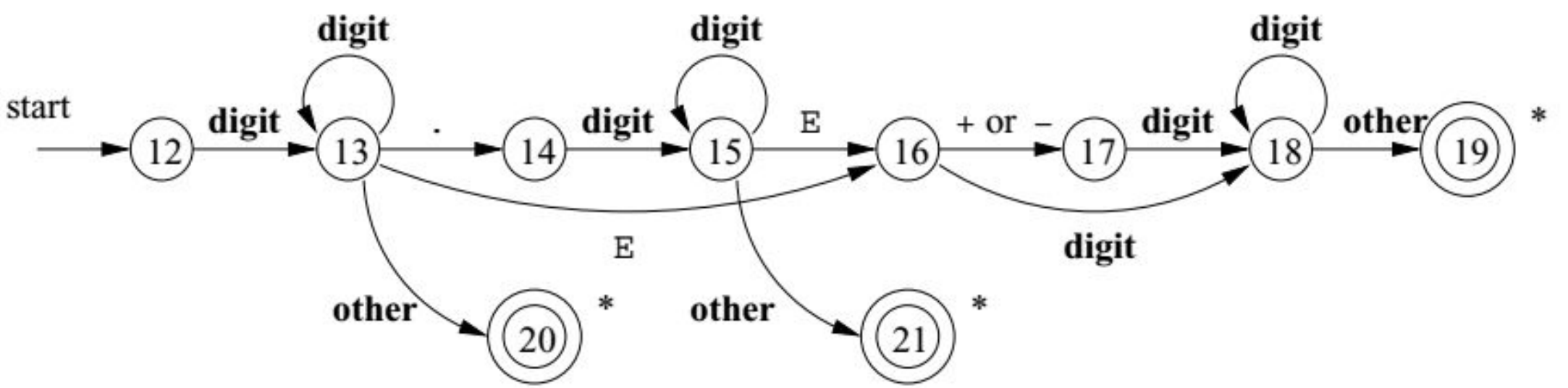


Figure 3.16: A transition diagram for unsigned numbers

Recognition of reserved words

- Usually, keywords like **if** or **then** are reserved (as they are in our running example), so they are not identifiers even though they look like identifiers.
- There are two ways to handle reserved words that look like identifiers:
  - *Install the reserved words in the symbol table initially. A field of the symbol-table entry indicates that these strings are never ordinary identifiers, and tells which token they represent.*
  - *Create separate transition diagrams for each keyword and prioritize the tokens so that the reserved-word tokens are recognized in preference to **id**, when the lexeme matches both patterns.*

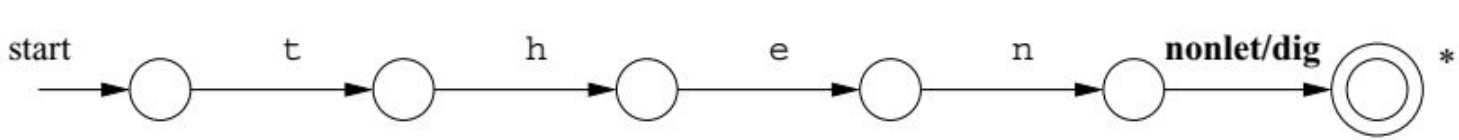


Figure 3.15: Hypothetical transition diagram for the keyword `then`

Architecture of a Transition-Diagram-Based Lexical Analyzer

- There are several ways that a collection of transition diagrams can be used to build a lexical analyzer.
- Regardless of the overall strategy, each state is represented by a piece of code. We may imagine a variable state holding the number of the current state for a transition diagram. A switch based on the value of state takes us to code for each of the possible states, where we find the action of that state. Often, the code for a state is itself a switch statement or multiway branch that determines the next state by reading and examining the next input character.

Architecture of a Transition-Diagram-Based Lexical Analyzer

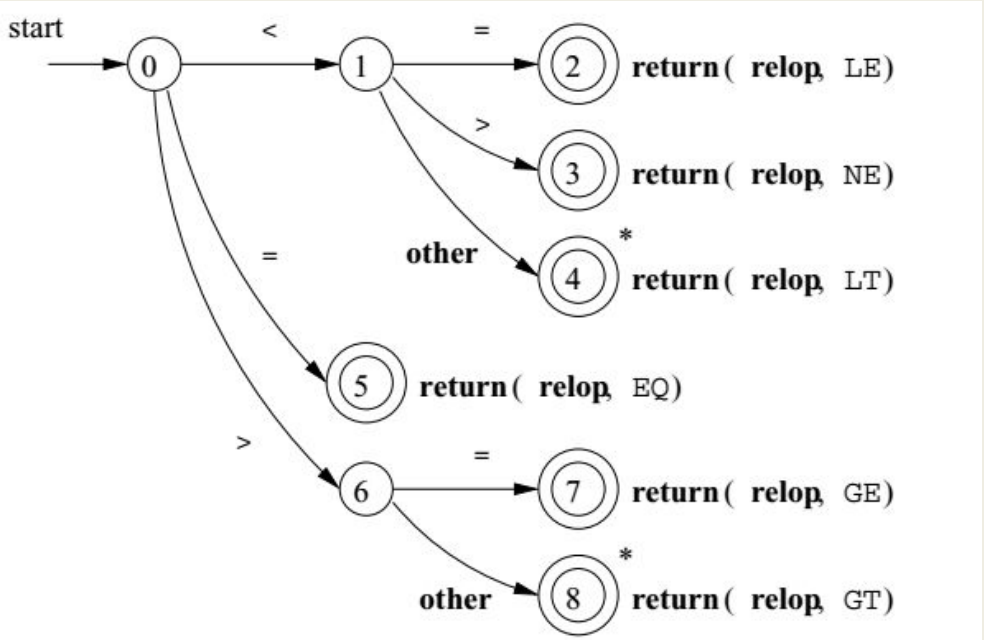


Figure 3.13: Transition diagram for **relop**

```
TOKEN getRelop()
{
    TOKEN retToken = new(RELOP);
    while(1) { /* repeat character processing until a return
                  or failure occurs */
        switch(state) {
            case 0: c = nextChar();
                    if ( c == '<' ) state = 1;
                    else if ( c == '=' ) state = 5;
                    else if ( c == '>' ) state = 6;
                    else fail(); /* lexeme is not a relop */
                    break;
            case 1: ...
            ...
            case 8: retract();
                    retToken.attribute = GT;
                    return(retToken);
        }
    }
}
```

Architecture of a Transition-Diagram-Based Lexical Analyzer

- To place the simulation of one transition diagram in perspective, let us consider the ways code like the one given in the previous slide could fit into the entire lexical analyzer.

1. We could arrange for the transition diagrams for each token to be tried sequentially. Then, the function **fail()** resets the pointer forward and starts the next transition diagram, each time it is called. This method allows us to use transition diagrams for the individual keywords, like the one given in previous slide. We have only to use these before we use the diagram for id, in order for the keywords to be reserved words.

Architecture of a Transition-Diagram-Based Lexical Analyzer

2. We could run the various transition diagrams "in parallel," feeding the next input character to all of them and allowing each one to make whatever transitions it required. If we use this strategy, we must be careful to resolve the case where one diagram finds a lexeme that matches its pattern, while one or more other diagrams are still able to process input. The normal strategy is to take the longest prefix of the input that matches any pattern. That rule allows us to prefer identifier **thenext** to keyword **then**, or the operator **->** to **-**, for example.
3. The preferred approach is to combine all the transition diagrams into one. We allow the transition diagram to read input until there is no possible next state, and then take the longest lexeme that matched any pattern.

Information about Symbol Table

- A symbol table may serve the following purposes depending upon the language in hand:
  - *To store the names of all entities in a structured form at one place which can passed from the frontend to the backend of the compiler.*
  - *To verify if a variable has been declared.*
  - *To implement type checking, by verifying assignments and expressions in the source code are semantically correct.*
  - *To determine the scope of a name (scope resolution).*

If a symbol table has to store information about the following variable declaration:

static int interest;

then it should store the entry such as:

<interest, int, static>

Information about Symbol Table

- A symbol table can be implemented in one of the following ways:
  - *Linear (sorted or unsorted) list*
  - *Binary Search Tree*
  - *Hash table*
- Among all, symbol tables are mostly implemented as hash tables, where the source code symbol itself is treated as a key for the hash function and the return value is the information about the symbol.

Information about Symbol Table

- A symbol table, either linear or hash, must provide the following operations:
  - **insert():** This operation is more frequently used by analysis phase, i.e., the first half of the compiler where tokens are identified and names are stored in the table. This operation is used to add information in the symbol table about unique names occurring in the source code. The insert() function takes the symbol and the information associated with that symbol - value, state, scope, and type - as arguments and stores the information in the symbol table.
  - **lookup():** lookup() operation is used to search a name in the symbol table to determine:
    - if the symbol exists in the table.
    - if it is declared before it is being used.
    - if the name is used in the scope.
    - if the symbol is initialized.
    - if the symbol declared multiple times.

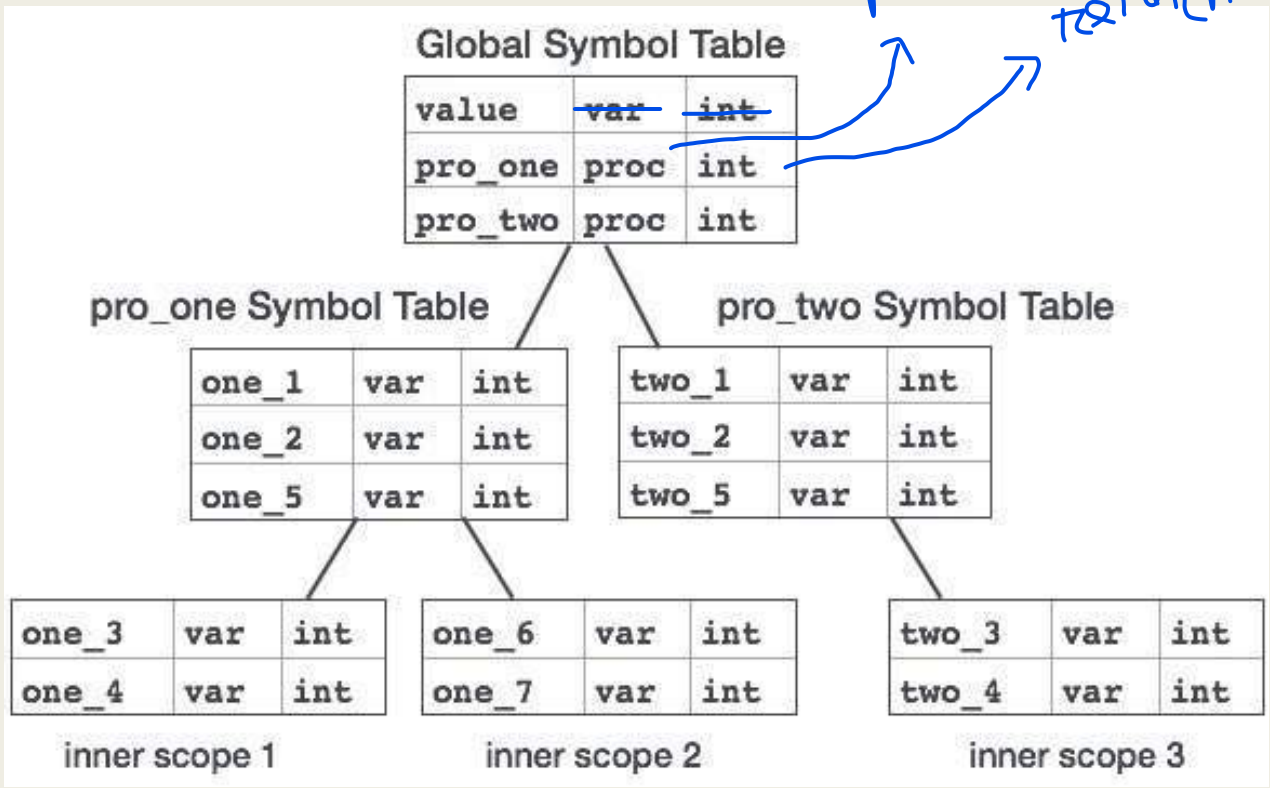
    This method returns 0 (zero) if the symbol does not exist in the symbol table. If the symbol exists in the symbol table, it returns its attributes stored in the table.

Symbol Table for scope

- **Scope** refers to the visibility of **variables**. In other words, which parts of your program can see or use it.
- A compiler maintains two types of symbol tables: a **global symbol table**which can be accessed by all the procedures and **scope symbol tables** that are created for each scope in the program.
- To determine the scope of a name, symbol tables are arranged in hierarchical structure as shown in the next slide.

Symbol Table for scope

```
. . .
int value=10;

void pro_one()
   {
   int one_1;
   int one_2;

      {                  \
      int one_3;          |_  inner scope 1
      int one_4;          |
      }                  /

   int one_5;

      {                  \
      int one_6;          |_  inner scope 2
      int one_7;          |
      }                  /
   }

void pro_two()
   {
   int two_1;
   int two_2;

      {                  \
      int two_3;          |_  inner scope 3
      int two_4;          |
      }                  /

   int two_5;
   }
. . .
```



Symbol Table for scope

- The global symbol table contains names for one global variable (int value) and two procedure names, which should be available to all the child nodes shown above. The names mentioned in the pro_one symbol table (and all its child tables) are not available for pro_two symbols and its child tables.
- This symbol table data structure hierarchy is stored in the semantic analyzer and whenever a name needs to be searched in a symbol table, it is searched using the following algorithm:
  - *first a symbol will be searched in the current scope, i.e. current symbol table.*
  - *if a name is found, then search is completed, else it will be searched in the parent symbol table until,*
  - *either the name is found or global symbol table has been searched for the name.*

What is Lex?

- Lex is a program generator designed for lexical processing of character input streams.
- It accepts a high-level, problem oriented specification for character string matching, and produces a program in a general purpose language which recognizes regular expressions.
- Lex source is a table of regular expressions and corresponding program fragments. The table is translated to a program which reads an input stream, copying it to an output stream and partitioning the input into strings which match the given expressions. As each such string is recognized the corresponding program fragment is executed.

Use of lex

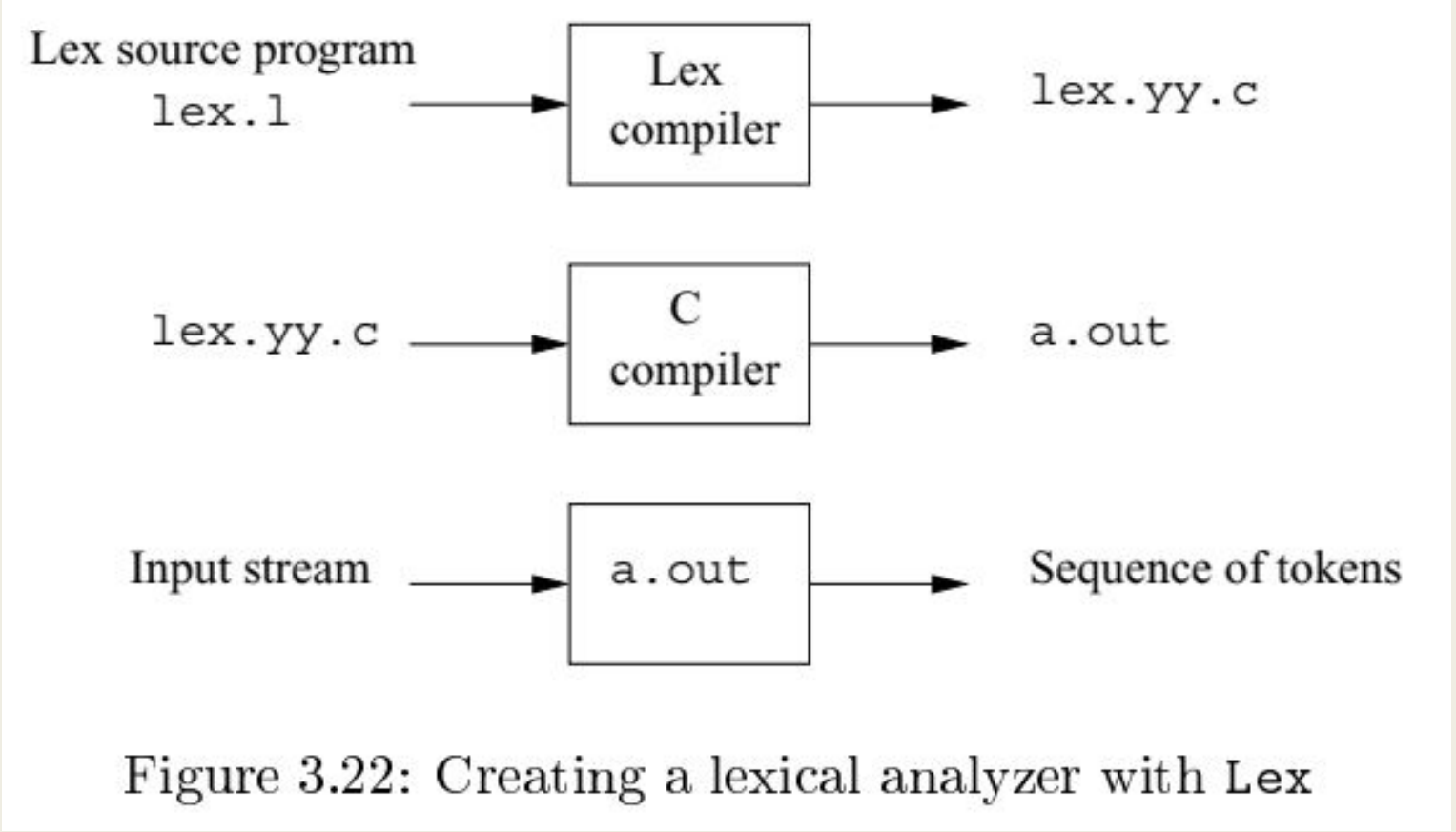


Figure 3.22: Creating a lexical analyzer with Lex

Use of lex

- An input file, which we call **lex.l**, is written in the Lex language and describes the lexical analyzer to be generated. The Lex compiler transforms **lex.l** to a C program, in a file that is always named **lex.yy.c**. The latter file is compiled by the C compiler into a file always called **a.out**. The C-compiler output is a working lexical analyzer that can take a stream of input characters and produce a stream of tokens.
- The normal use of the compiled C program, referred to as **a.out**, is as a subroutine of the parser. It is a C function that returns an integer, which is a code for one of the possible token names. The attribute value, whether it be another numeric code, a pointer to the symbol table, or nothing, is placed in a global variable **yylval**, which is shared between the lexical analyzer and parser, thereby making it simple to return both the name and an attribute value of a token.

Structure of a Lex program

- A Lex program has the following form:

```
declarations
%%
translation rules
%%
auxiliary functions
```

Structure of a Lex program

- The declarations section includes declarations of variables, manifest constants (identifiers declared to stand for a constant, e.g., the name of a token), and regular definitions as we have seen before.
- The translation rules each have the form

```
Pattern   { Action }
```

  - *Each pattern is a regular expression, which may use the regular definitions of the declaration section. The actions are fragments of code that is run when a pattern is matched.*
- The third section holds whatever additional functions are used in the actions.

```
%{
    /* definitions of manifest constants
    LT, LE, EQ, NE, GT, GE,
    IF, THEN, ELSE, ID, NUMBER, RELOP */
%}

/* regular definitions */
delim     [ \t\n]
ws        {delim}+
letter    [A-Za-z]
digit     [0-9]
id        {letter}({letter}|{digit})*
number    {digit}+(\.{digit}+)?(E[+-]?{digit}+)?

%%

{ws}      {/* no action and no return */}
if        {return(IF);}
then      {return(THEN);}
else      {return(ELSE);}
{id}      {yylval = (int) installID(); return(ID);}
{number}  {yylval = (int) installNum(); return(NUMBER);}
"<"       {yylval = LT; return(RELOP);}
"<="      {yylval = LE; return(RELOP);}
"="       {yylval = EQ; return(RELOP);}
"<>"      {yylval = NE; return(RELOP);}
">"       {yylval = GT; return(RELOP);}
">="      {yylval = GE; return(RELOP);}

%%

int installID() {/* function to install the lexeme, whose
                    first character is pointed to by yytext,
                    and whose length is yyleng, into the
                    symbol table and return a pointer
                    thereto */
}

int installNum() {/* similar to installID, but puts numer-
                     ical constants into a separate table */
}
```

Conflict resolution in Lex

- The two rules that Lex uses to decide on the proper lexeme to select, when several prefixes of the input match one or more patterns are:
  - *Always prefer a longer prefix to a shorter prefix.*
  - *If the longest possible prefix matches two or more patterns, prefer the pattern listed first in the Lex program.*
- The first rule tells us to continue reading letters and digits to find the longest prefix of these characters to group as an identifier. It also tells us to treat <= as a single lexeme, rather than selecting < as one lexeme and = as the next lexeme.
- The second rule makes keywords reserved, if we list the keywords before id in the program.

The Lookahead Operator

- Lex automatically reads one character ahead of the last character that forms the selected lexeme, and then retracts the input so only the lexeme itself is consumed from the input.
- However, sometimes, we want a certain pattern to be matched to the input only when it is followed by a certain other characters.
- If so, we may use the slash in a pattern to indicate the end of the part of the pattern that matches the lexeme. What follows / is additional pattern that must be matched before we can decide that the token in question was seen, but what matches this second pattern is not part of the lexeme.

The Lookahead Operator

Example 3.13: In Fortran and some other languages, keywords are not reserved. That situation creates problems, such as a statement

```
IF(I,J) = 3
```

where `IF` is the name of an array, not a keyword. This statement contrasts with statements of the form

```
IF( condition ) THEN ...
```

where `IF` is a keyword. Fortunately, we can be sure that the keyword `IF` is always followed by a left parenthesis, some text — the condition — that may contain parentheses, a right parenthesis and a letter. Thus, we could write a `Lex` rule for the keyword `IF` like:

```
IF / \( .* \) {letter}
```

How does Lex actually match the patterns?

- The recognition of the regular expressions is performed by a deterministic finite automaton generated by Lex.
- It converts the regular expressions given into deterministic finite automata and simulates these automata to recognize each type of string.

Overview of Lex