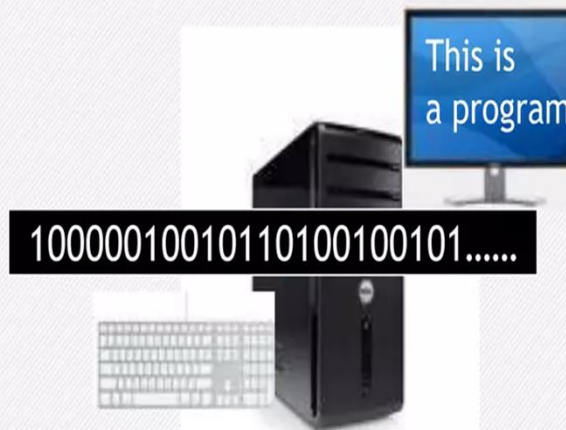


- **Language Processors:** Overview of language processing system: – preprocessors – compiler – assembler – Linkers & loaders, difference between compiler and interpreter-structure of a compiler:–phases of a compiler.
- **Lexical Analysis:** - Role of Lexical Analysis: Lexical analysis Versus Parsing – Tokens, Patterns, and Lexemes – Attributes for Tokens – Lexical errors - Input Buffering: Buffer Pairs – Sentinels
- **Specification of Tokens:** Strings and Languages – Operations on Languages – Regular Expressions – Regular Definitions
- **Recognition of Tokens:** Transition Diagrams – Recognition of Reserved Words and Identifiers - Completion of the Running Example – Architecture of a Transition-Diagram-Based Lexical Analyzer
- **The Lexical Analyzer Generator (LEX):** Use of Lex – Structure of Lex Programs

Why Use a compiler?

2

- All computers only understand machine language

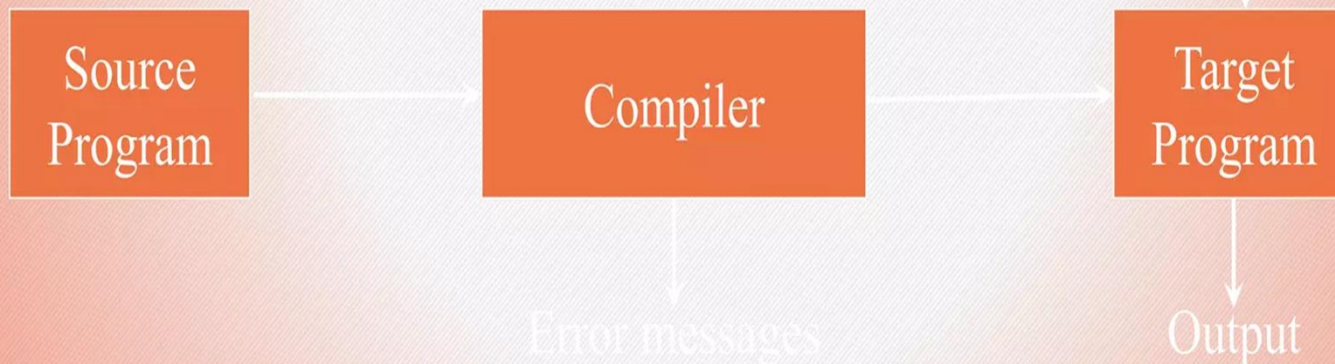


- Therefore, high-level language instructions must be translated into machine language prior to execution

Compiler

3

- A compiler is a large program that can read a program in one language the *source* language - and translate it into an equivalent program in another language - the *target* language;
- An important role of the compiler is to report any errors in the source program that it detects during the translation process



- If the target program is an executable machine-language program, it can then be called by the user to process inputs and produce outputs.

Example

4

```
sum = 0;  
for (x = 3; x < 5; x++)  
{ cout << "x is " << x;  
  cout << endl;  
  sum += x;  
  a *= b / 2;
```

Source Code



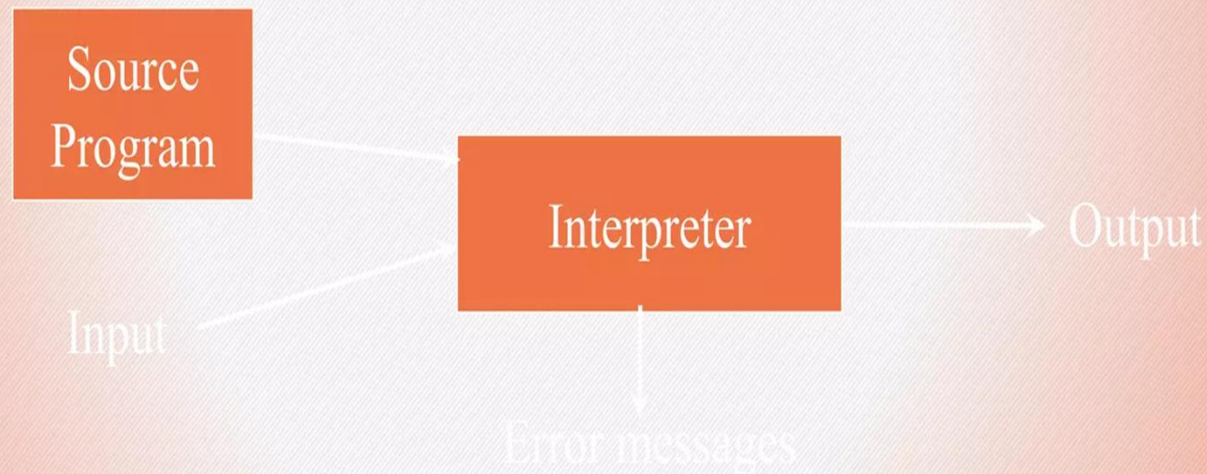
```
10011101  
01011011  
10111100  
01100110  
10101100  
00011011
```

Target Code

Interpreter

5

An interpreter is another common kind of language processor. Instead of producing a target program as a translation, an interpreter appears to directly execute the operations specified in the source program or inputs supplied by the user

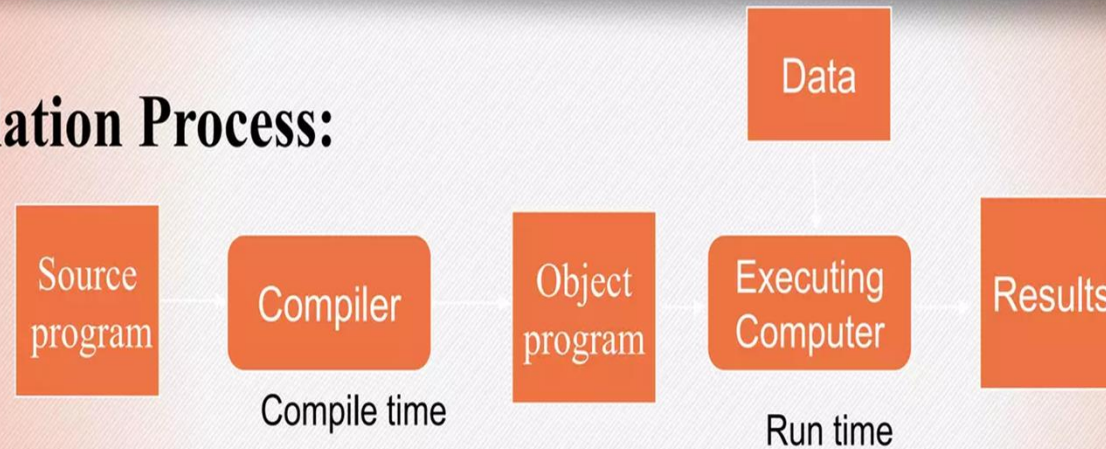


The machine-language target program produced by a compiler is usually much faster than an interpreter at mapping inputs to outputs . An interpreter, however, can usually give better error diagnostics than a compiler, because it executes the source program statement by statement.

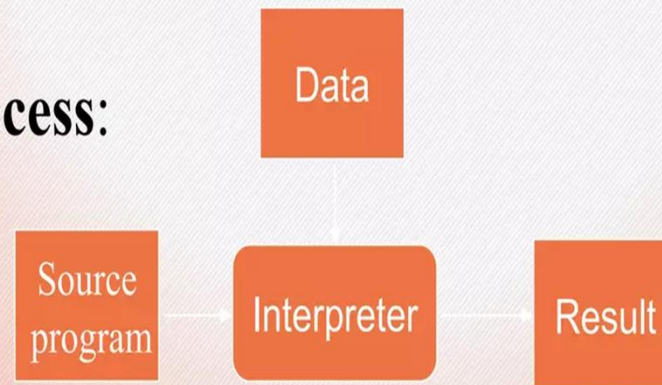
Working Process of Compilers Vs Interpreter

6

Compilation Process:



Interpretive Process:

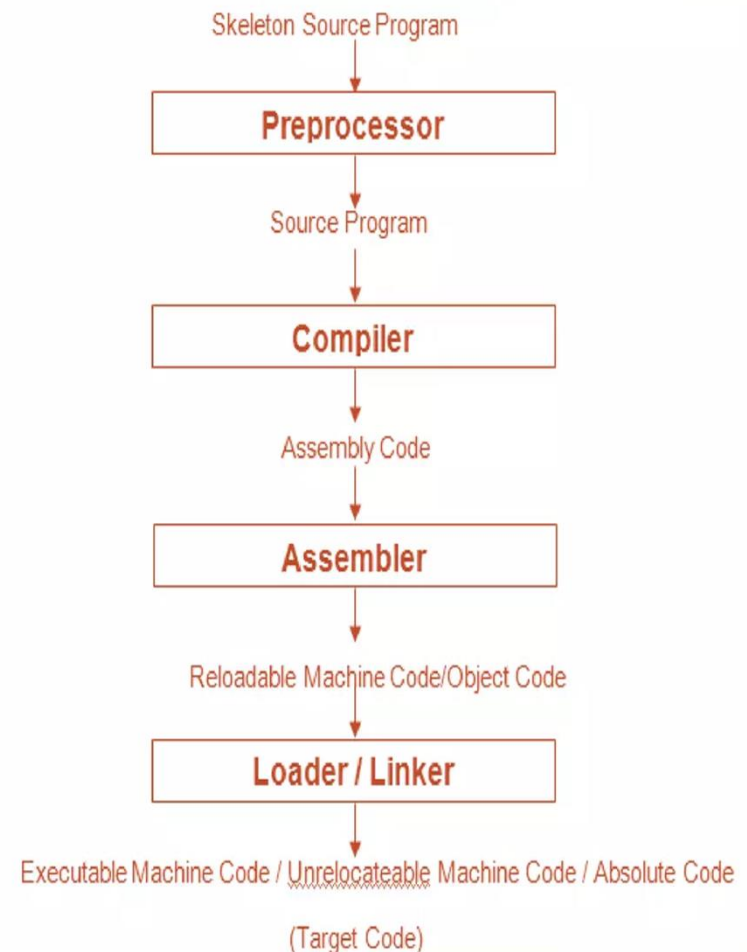


Sr.	Compiler	Interpreter
1	Compiler Takes Entire program as input	Interpreter Takes Single instruction as input .
2	Intermediate Object Code is Generated	No Intermediate Object Code is Generated
3	Conditional Control Statements are Executes faster	Conditional Control Statements are Executes slower
4	Memory Requirement : More (Since Object Code is Generated)	Memory Requirement is Less
5	Program need not be compiled every time	Every time higher level program is converted into lower level program
6	Errors are displayed after entire program is checked	Errors are displayed for every instruction interpreted (if any)
7	Programming language like C, C++ use compilers.	Programming language like Python, Ruby use interpreters.

Context of a Compiler

8

- The programs which assist the compiler to convert a skeletal source code into executable form make the context of a compiler and is as follows:
- **Preprocessor:**
The preprocessor scans the source code and includes the header files which contain relevant information for various functions.
- **Compiler:**
The compiler passes the source code through various phases and generates the target assembly code.



- A preprocessor produce input to compilers. They may perform the following functions.
- 1. *Macro processing*: A preprocessor may allow a user to define macros that are short hands for
- longer constructs.
- 2. *File inclusion*: A preprocessor may include header files into the program text.
- 3. *Rational preprocessor*: these preprocessors augment older languages with more modern flow-ofcontrol
- and data structuring facilities.
- 4. *Language Extensions*: These preprocessor attempts to add capabilities to the language by certain
- amounts to build-in macro

Cont....

9

- **Assembler:**

The assembler converts the assembly code into relocatable machine code or object code. Although this code is in 0 and 1 form, but it cannot be executed because this code has not been assigned the actual memory addresses.

- **Loader/Link Editor:**

It performs two functions. The process of loading consists of taking machine code, altering the relocatable addresses and placing the altered instructions and data in memory at proper location.

The link editor makes a single program from several files of relocatable machine code. These files are library files which the program needs.

The loader/link editor produces the executable or absolute machine code.

Phases of Compiler Design

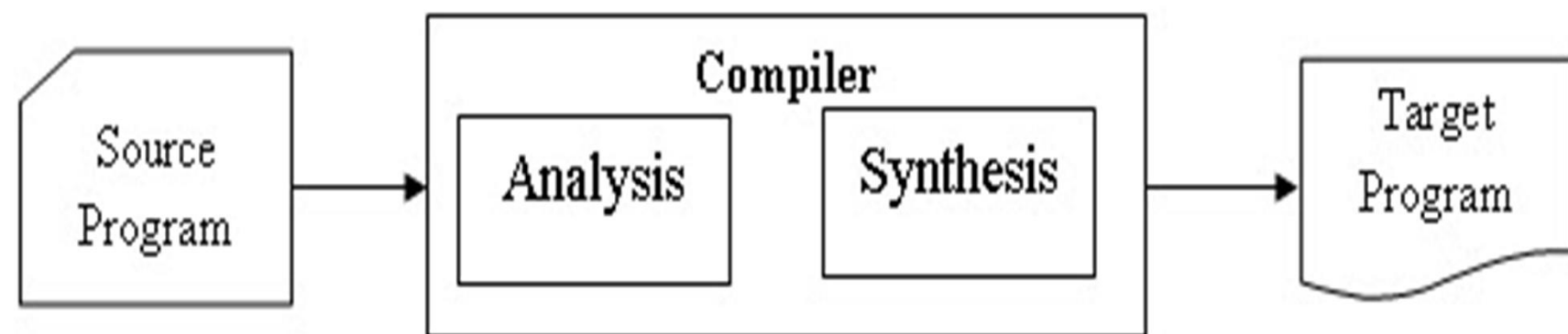
10

A compiler operates in phases. A phase is a logically interrelated operation that takes source program in one representation and produces output in another representation. The phases of a compiler are shown in below

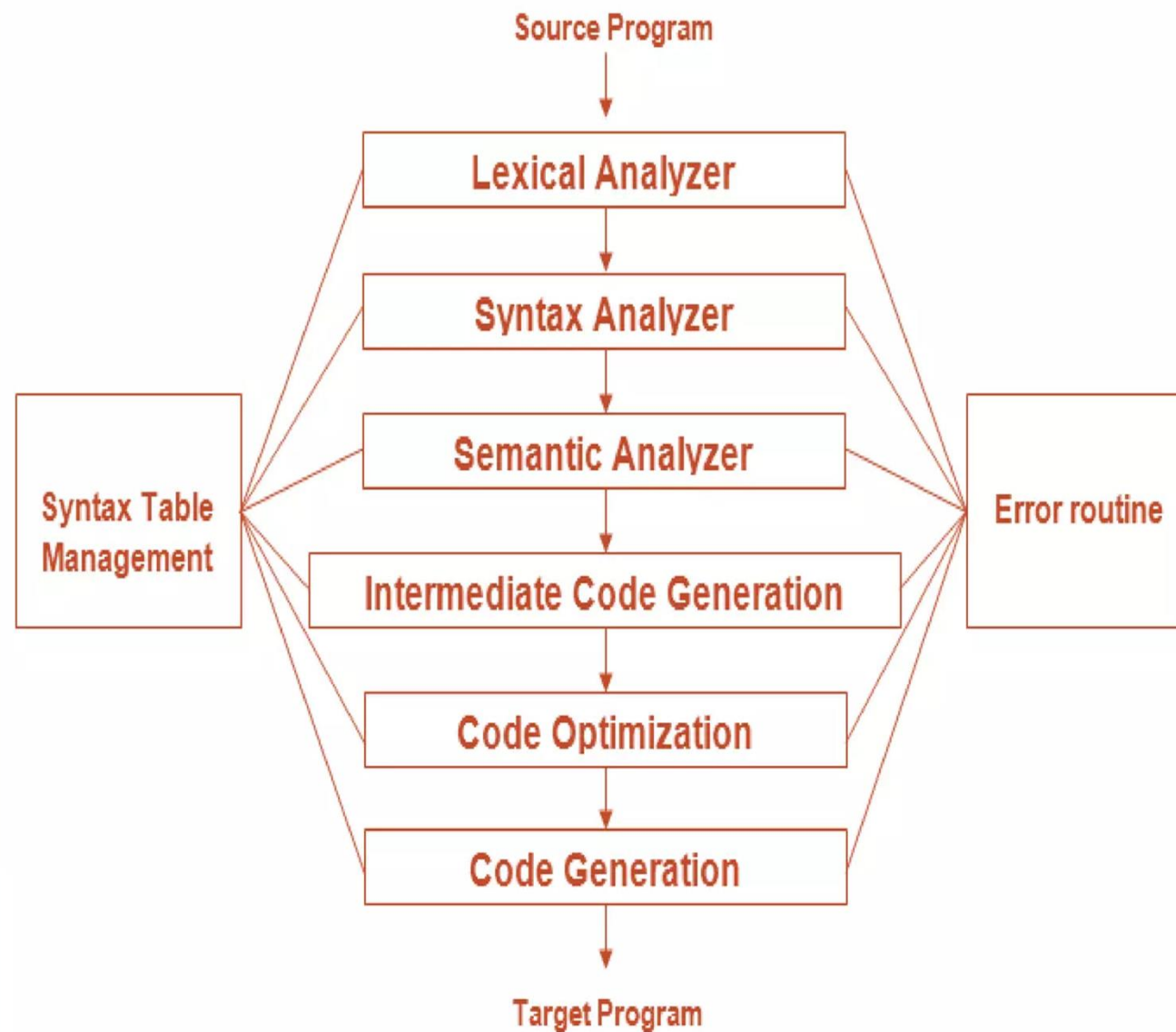
There are two phases of compilation.

- Analysis (Machine Independent/Language Dependent)
- Synthesis(Machine Dependent/Language independent)

Compilation process is partitioned into no-of-sub processes called '**phases**'.



Analysis and Synthesis model



Phase-1: Lexical Analysis

12

- Lexical analyzer reads the stream of characters making up the source program and groups the characters into meaningful sequences called *lexeme*
- For each lexeme, the lexical analyzer produces a **token** of the form that it passes on to the subsequent phase, syntax analysis

(token-name, attribute-value)

- Token-name: an abstract symbol is used during syntax analysis.
- attribute-value: points to an entry in the symbol table for this token.

Phase-2: Syntax Analysis

15

- Also called **Parsing or Tokenizing**.
- The parser uses the first components of the tokens produced by the lexical analyzer to create a tree-like intermediate representation that depicts the grammatical structure of the token stream.
- A typical representation is a syntax tree in which each interior node represents an operation and the children of the node represent the arguments of the operation

Example:

13

newval := oldval + 12 —————→ **Tokens:**

newval Identifier

= Assignment operator

oldval Identifier

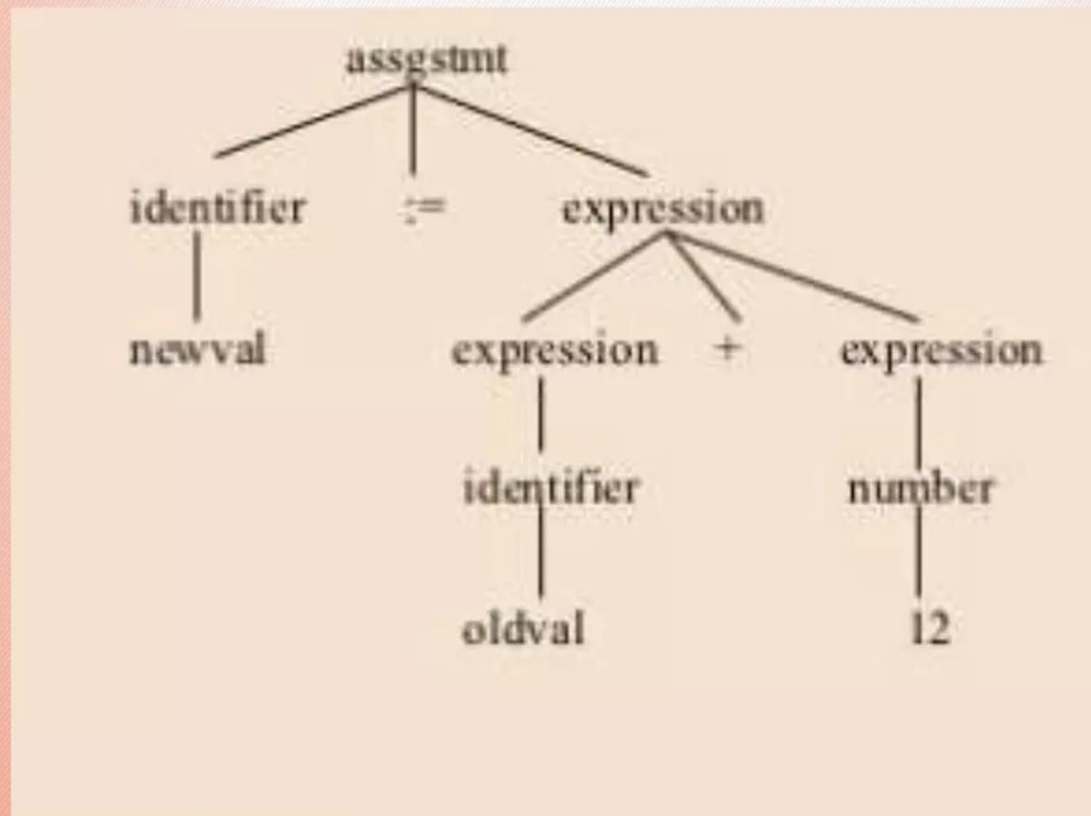
+ Add operator

12 Number

Lexical analyzer truncates white spaces and also removes errors.

Example:

16



Phase-3: Semantic Analysis

17

- The semantic analyzer uses the syntax tree and the information in the symbol table to check the source program for semantic consistency with the language definition.
- Gathers type information and saves it in either the syntax tree or the symbol table, for subsequent use during intermediate-code generation.
- An important part of semantic analysis is **type checking**, where the compiler checks that each operator has matching operands.
- For example, many programming language definitions require an array index to be an integer; the compiler must report an error if a floating-point number is used to index an array.
- Example: **newval := oldval+12**

The type of the identifier newval must match with the type of expression (oldval+12).

Example:

18

- Semantic analysis
 - Syntactically correct, but semantically incorrect

example:

```
sum = a + b;
```

```
int a;  
double sum;  
char b;
```

data type mismatch

Semantic records

a	integer
sum	double
b	char

p	cpu
sum	double

Phase-4: Intermediate Code Generation

19

After syntax and semantic analysis of the source program, many compilers generate an explicit low-level or machine-like intermediate representation (a program for an abstract machine). This intermediate representation should have two important properties:

- it should be easy to produce and
- it should be easy to translate into the target machine.

The considered intermediate form called **three-address code**, which consists of a sequence of assembly-like instructions with three **operands per instruction**. Each operand can act like a **register**.

This phase bridges the analysis and synthesis phases of translation.

Example:

20

newval := oldval + fact * 1



Id1 := Id2 + Id3 * 1



Temp1 = into real (1)

Temp2 = Id3 * Temp1

Temp3 = Id2 + Temp2

Id1 = Temp3

Phase-5: Code Optimization

21

- The compiler looks at large segments of the program to decide how to improve performance
- The machine-independent code-optimization phase attempts to improve the intermediate code so that better target code will result.
- Usually better means:
 - faster, shorter code, or target code that consumes less power.
- There are simple optimizations that significantly improve the running time of the target program without slowing down compilation too much.
- Optimization cannot make an inefficient algorithm efficient - “only makes an efficient algorithm more efficient”

Example:

22

- The above intermediate code will be optimized as:

```
Temp1    =    Id3    *    1
Id1       =    Id2    +    Temp1
```


Phase-6: Code Generation

23

- The last phase of translation is code generation.
- Takes as input an intermediate representation of the source program and maps it into the target language
- If the target language is machine, code, registers or memory locations are selected for each of the variables used by the program.
- Then, the intermediate instructions are translated into sequences of machine instructions that perform the same task.
- A crucial aspect of code generation is the judicious assignment of registers to hold variables.

Example:

24

$ld1 := ld2 + ld3 * 1$

MOV R1,ld3

MUL R1,#1

MOV R2,ld2

ADD R1,R2

MOV ld1,R1

Symbol-Table Management

26

- 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
- 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.

new Val	Id1 & attribute
old Val	Id2 & attribute
fact	Id3 & attribute

Error Handling Routine:

27

- One of the most important functions of a compiler is the detection and reporting of errors in the source program. The error message should allow the programmer to determine exactly where the errors have occurred. Errors may occur in all or the phases of a compiler.
- Whenever a phase of the compiler discovers an error, it must report the error to the error handler, which issues an appropriate diagnostic message. Both of the table-management and error-Handling routines interact with all phases of the compiler.

Code optimization

- Code optimization phase attempts to improve the intermediate code, so that faster-running machine code will result.
- Faster/shorter/Less power consumable target code.
- Compiler spent significant amount of time on this phase.
- Optimized Three address code after Code Optimization phase for the example statement is

- Example:



Code Generation

- It takes intermediate representation of the source program as input and maps it into the target language.
- If the target language is machine code, registers or memory locations are selected for each of the variables used by the program.
- Intermediate instructions are translated into sequences of machine instructions

- Crucial part is assignment of registers to hold variables.
- First operand of each instruction specifies destination.
- F-> floating point number
- #-> 60.0 consider as immediate constant

- MOVF id3, R2
- MULF #60.0, R2
- MOVF id2, R1
- ADDF R2, R1
- MOVF R1, id1

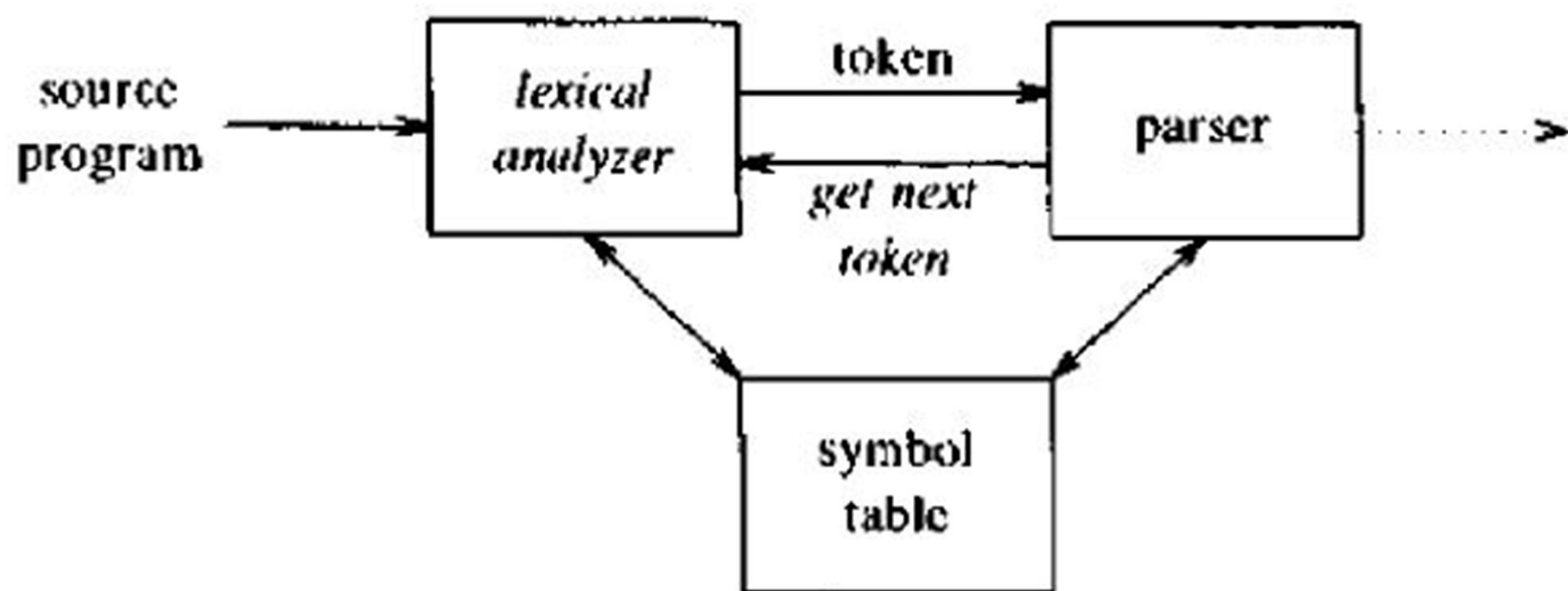
Error handler

- Each phase encounters errors.
- After detecting an error, a phase must somehow deal with that error, so that compilation can proceed, allowing further errors in the source program to be detected.
- Lexical analysis phase can detect errors that do not form any token of the language.
- Syntax analysis phase can detect the token stream that violates the (structure (or) syntax rules of the language.
- Semantic analysis phase detects the constructs that have no meaning to the operation involved.

Phase	Pass
<p>The process of compilation is carried in various steps. Each step is called a phase</p>	<p>One complete scan of the source language is called pass It includes reading an input file and writing to an output file</p>
<p>Different phases include: LA,SA,SeA,ICG,CO,CG</p>	<p>Many phases can be grouped as one pass The task of compilation may be carried out in single pass or multiple passes</p>

Role of Lexical Analysis

- The lexical analyzer is the **first phase** of a compiler.
- Its main task is to read the input characters and produce as output a sequence of tokens that the parser uses for syntax analysis.
- Another task of lexical analyzer is **stripping out** from the source program **comments and white space in the form of blank and tab and newline characters**.
- **Correlating error messages** from the compiler with the source program.



- The lexical analyzer may keep track of the **number of newline characters** seen, so that line number can be **associated with an error message**.
- In some compilers, the lexical analyzer is in charge of making **a copy of the source program with the error messages marked in it**.
- If the lexical analyzer finds a token invalid, **it generates an error**.
- The lexical analyzer works closely with the syntax analyzer.
- It reads character streams from the source code, checks for legal tokens, and passes the data to the syntax analyzer when it demands.
- The lexical analyzer collects information about tokens into their associated attributes.
- After identifying the tokens, the strings are entered into database called a **symbol table**.
- **It works in two phases:**
 1. Scan
 2. Separation of tokens

Lexical analysis Vs Parsing

- All compilers separate the task of analyzing syntax into two different parts.
- Lexical and syntax
- Lexical-> small scale language constructs
 - Names and literals
- Syntax-> large scale language constructs
 - expressions, statements and program units

Why lexical analysis is separated from syntax analysis?

1. Simplicity

- lexical analysis is simplified because it is less complex than syntax analyser
- Syntax analyser can be smaller and cleaner by removing low level details of lexical analysis

2. Efficiency

- lexical analysis should be optimized (requires significant portion of total compile time)
- Syntax analysis should not be optimized

3. Portability

- Lexical analysis may not be portable because input device-specific peculiarities can be restricted to scanner
- Syntax analysis is always portable

Token

- Token is a sequence of characters that can be treated as a single logical entity. Sequence of characters having the collective meaning in the source program
- Typical tokens are identifiers, keywords, operators, special symbols, constants.
- **Pattern:** Set of rules that describe tokens
- **Lexeme:** Sequence of characters in the source program that are matched with a pattern of the token

- Suppose we have a simple programming language that includes the following tokens:
 - **Keywords:** if, else, while
 - **Identifiers:** Any sequence of letters and digits, starting with a letter
 - **Constants:** Integers
- **Patterns**
 - **Pattern for Keywords:** if|else|while
 - **Pattern for Identifiers:** [a-zA-Z][a-zA-Z0-9]*
 - **Pattern for Constants:** [0-9]+
- Now, let's identify lexemes in the given code based on these patterns:
 - **Keywords:**
 - if
 - else
 - **Identifiers:**
 - x
 - y
 - **Constants:**
 - 0
 - 2
 - 1

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"

Figure 3.2: Examples of tokens

Attributes for Tokens:

- A token has only a single attribute – a pointer to the symbol-table entry in which the information about the token is kept.
- The token names and associated attribute values for the 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>

<add_op>

<number, integer value 2>

Lexical errors

- It is hard for lexical analyzer to tell without aid of other computers, that there is a source code error.
- Some errors are out of power of lexical analyzer to recognize: – `fi (a == f(x)) ...`
- Lexical analyzer can not tell whether `fi` is a misspelling keyword `if` or an undeclared function identifier. Since `fi` is valid lexeme.
- Such errors are recognized when no pattern for tokens matches a character sequence.

- Other phase of the compiler probably parser handle this type of error.
- If lexical analyser unable to proceed because of none of the patterns for tokens matches any prefix of the remaining input ,
- The simplest recovery strategy is **panic mode recovery**

Error recovery

- Panic mode: successive characters are ignored until we reach to a well formed token
 - 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

Input Buffering: Buffer Pairs – Sentinels

Input Buffering

- There are times when a lexical analyzer needs to look ahead several characters beyond the lexeme for a token before a match can be announced.
- Buffering techniques can be used to reduce the overhead required to process input characters.
- The buffer is divided into two N -character halves.

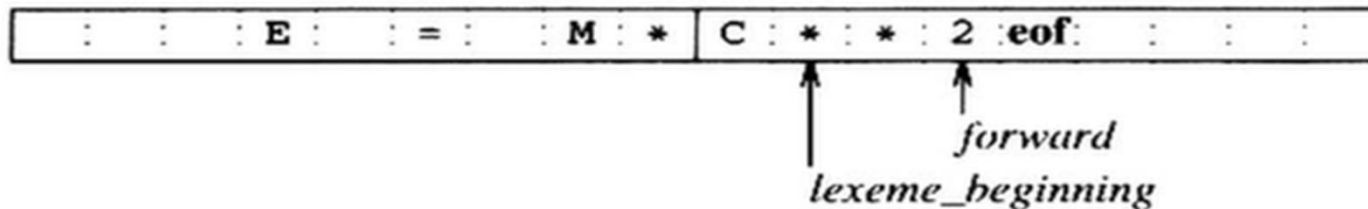


Fig. 3.3. An input buffer in two halves.

Buffer pairs

- Because of the amount of time taken to process characters and number of characters must be processed during the compilation of large source program, specialized buffering techniques have been introduced.
- We need to introduce a two buffer scheme to handle large look-aheads safely

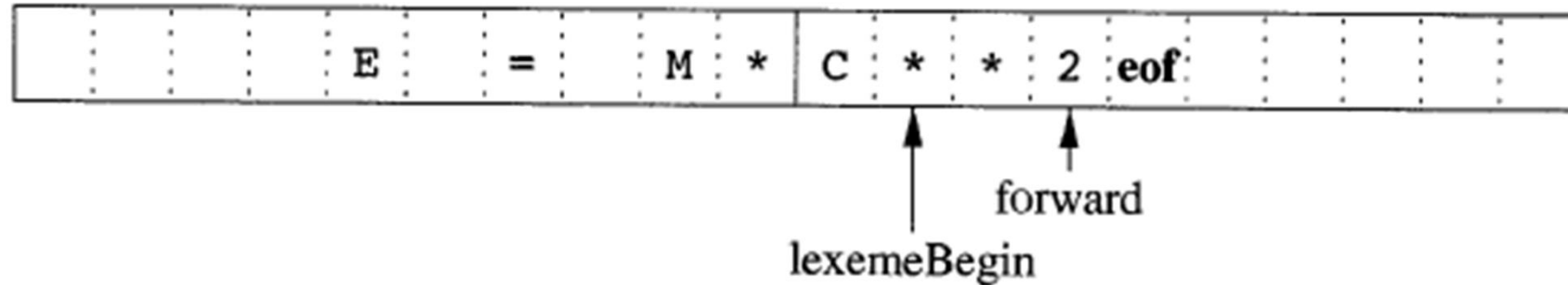


Figure 3.3: Using a pair of input buffers

- Each buffer is of same size N
- N is usually size of disk block
- We can read N characters into a buffer
- 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:
- 1. **Pointer lexeme begin** :marks the beginning of the current lexeme
- 2. **Pointer forward**: scans until a pattern match is found
- Once the next lexeme is determined, forward is set to the character at its right end.
- Lexeme begin is set to the character immediately after the lexeme just found.

Input Buffering(Cont.)

```
if forward at end of first half then begin  
    reload second half;  
    forward := forward + 1  
end  
else if forward at end of second half then begin  
    reload first half;  
    move forward to beginning of first half  
end  
else forward := forward + 1;
```

Fig. 3.4. Code to advance forward pointer.

Sentinels

- For each character read we make two tests:
 - one for the end of the buffer
 - One to determine what character is read
- 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 can not be part of the source program -eof

- Eof is marked for the end of the entire input.
- Any eof that appears other than at the end of a buffer means that the input is at an end.

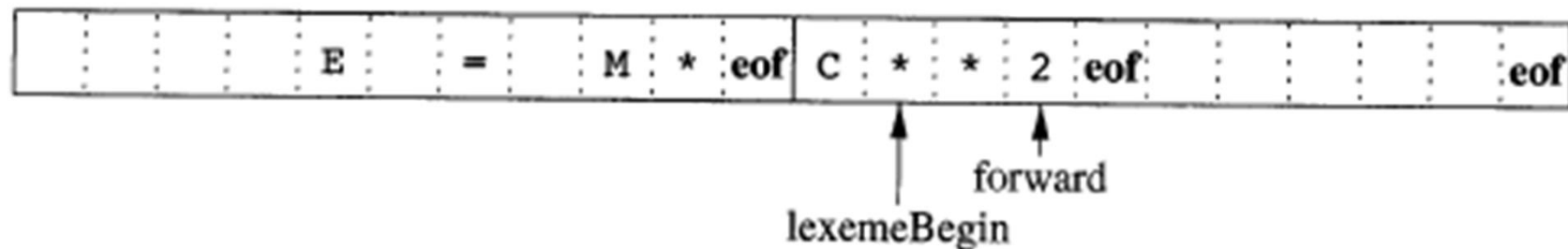


Figure 3.4: Sentinels at the end of each buffer

Sentinels

to Improving Input Buffering (Cont.)

```
forward := forward + 1;  
if forward↑ = eof then begin  
    if forward at end of first half then begin  
        reload second half;  
        forward := forward + 1  
    end  
    else if forward at end of second half then begin  
        reload first half;  
        move forward to beginning of first half  
    end  
    else /* eof within a buffer signifying end of input */  
        terminate lexical analysis  
end
```

Specification of Tokens

- In theory of compilation regular expressions are used to formalize the specification of tokens
- Regular expressions are means for specifying regular languages
 - Strings and Languages
 - Operations on Languages
 - Regular Expressions
 - Regular Definitions

Strings and Languages

Some Concepts:

- *symbol*: letters, digits, and punctuation
- *alphabet*: any finite set of symbols
e.g. $\{0,1\}$, ASCII, Unicode
- *string*: a finite sequence of symbols
 $|s|$: length of a string s
 ϵ : empty string
- *language*: any countable set of strings
e.g. Φ , $\{\epsilon\}$, C programs, English sentences

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 strings:

- *concatenation:* xy
e.g. 1) $x = \text{dog}$, $y = \text{house}$, $xy = \text{doghouse}$.
2) $\epsilon S = S \epsilon = S$
- *exponentiation:*

$$S^0 = \epsilon$$

$$S^i = S^{i-1}S$$

$$S^1 = S$$

$$S^2 = SS$$

$$S^3 = SSS$$

Operations on Languages

- *union*: $L \cup M = \{s \mid s \text{ is in } L \text{ or } s \text{ is in } M\}$
- *concatenation*: $LM = \{st \mid s \text{ is in } L \text{ and } t \text{ is in } M\}$
- *closure*:

a) Kleene closure:

b) Positive closure: $L^* = U_{i=0}^{\infty} L^i$

$$L^+ = U_{i=1}^{\infty} L^i$$

1. $L \cup D$ is the set of letters and digits — strictly speaking the language with 62 strings of length one, each of which strings is either one letter or one digit.
2. LD is the set of 520 strings of length two, each consisting of one letter followed by one digit.
3. L^4 is the set of all 4-letter strings.
4. L^* is the set of all strings of letters, including ϵ , the empty string.
5. $L(L \cup D)^*$ is the set of all strings of letters and digits beginning with a letter.
6. D^+ is the set of all strings of one or more digits.

Regular Expressions

- **Describing languages**

e.g. C identifiers: $letter_ (letter_ | digit)^*$

notice:

a) The regular expressions are built recursively out of smaller regular expressions

b) Each regular expression r denotes a language $L(r)$

- **BASIS: (two rules)**

1. ϵ is a regular expression, and $L(\epsilon)$ is $\{\epsilon\}$

2. If a is a symbol in Σ , then a is a regular expression, and $L(a) = \{a\}$

- **INDUCTION:**

1. $(r) | (s)$ is a regular expression denoting the language $L(r) \cup L(s)$
2. $(r)(s)$ is a regular expression denoting the language $L(r)L(s)$
3. $(r)^*$ is a regular expression denoting $(L(r))^*$
4. (r) is a regular expression denoting $L(r)$

- **Some conventions:**

1. $*$ has highest precedence and is left associative
2. Concatenation has second highest precedence and is left associative

3. | has lowest precedence and is left associative

e.g. $(a)|((b)^*(c)) = a|b^*c$

- *regular set:*

A language that can be defined by a regular expression

- *equivalent*

Two regular expressions r and s denote the same regular set, write $r=s$

- Algebraic laws for 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

Regular Definitions

- Regular Definition

A sequence of definitions of the form:

$d_1 \rightarrow r_1$

$d_2 \rightarrow r_2$

...

$d_n \rightarrow r_n$

where:

1. Each d_i is a new symbol
2. Each r_i is a regular expression

- Example:

C identifiers

letter_ -> A|B|...|Z|a|b| ...|z|_

digit->0|1|...|9

*id ->letter_(letter_ | digit)**

- The regular definition for **Unsigned numbers (integer or floating point)** such as 5280, 0.01234, 6.336E4, or 1.89E-4.
 - ❖ $digit \rightarrow 0|1|2|\dots|9$
 - ❖ $digits \rightarrow digit\,digit^*$
 - ❖ $optionalFraction \rightarrow .digits \mid \varepsilon$
 - ❖ $optionalExponent \rightarrow (E(+|-|\varepsilon)\,digits) \mid \varepsilon$
 - ❖ $number \rightarrow digits\,optionalFraction\,optionalExponent$
- More examples: integer constant, string constants, reserved words, operator, real constant.

Extensions of Regular Expressions

- *One or more instances: +*
 1. $(r)^+$ denotes the language $(L(r))^+$
 2. $r^* = r^+ | \epsilon$
 3. $r^+ = rr^* = r^*r$
- *Zero or one instance: ?*
 1. $r? = r | \epsilon$
 2. $L(r?) = L(r) \cup \{\epsilon\}$
- *Character classes:*
 1. $a_1 | a_2 | \dots | a_n = [a_1 a_2 \dots a_n]$.
 2. $a | b | \dots | z = [a-z]$

Recognition of Tokens

- Transition Diagrams
- Recognition of Reserved Words and Identifiers
- Completion of the Running Example
- Architecture of a Transition–Diagram-Based Lexical Analyzer

Recognition of Tokens

How to recognize tokens?

- *Reserved words: if, else, then...*
- *Id: letter*
- *Number: digit*
- *Relop: <, >, =, <=, >=, <>...*
- *Ws: blank, tab, newline...*

$$\begin{array}{lcl}
\textit{stmt} & \rightarrow & \textbf{if } \textit{expr} \textbf{ then } \textit{stmt} \\
& | & \textbf{if } \textit{expr} \textbf{ then } \textit{stmt} \textbf{ else } \textit{stmt} \\
& | & \epsilon \\
\textit{expr} & \rightarrow & \textit{term} \textbf{ relop } \textit{term} \\
& | & \textit{term} \\
\textit{term} & \rightarrow & \textbf{id} \\
& | & \textbf{number}
\end{array}$$

Figure 3.10: A grammar for branching statements

$$\begin{array}{lcl}
\textit{digit} & \rightarrow & [0-9] \\
\textit{digits} & \rightarrow & \textit{digit}^+ \\
\textit{number} & \rightarrow & \textit{digits} (. \textit{digits})? (\textbf{E} [+-]? \textit{digits})? \\
\textit{letter} & \rightarrow & [A-Za-z] \\
\textit{id} & \rightarrow & \textit{letter} (\textit{letter} | \textit{digit})^* \\
\textit{if} & \rightarrow & \textbf{if} \\
\textit{then} & \rightarrow & \textbf{then} \\
\textit{else} & \rightarrow & \textbf{else} \\
\textit{relop} & \rightarrow & < | > | <= | >= | = | <>
\end{array}$$

Figure 3.11: Patterns for tokens of Example 3.8

$$ws \rightarrow (\textbf{blank} | \textbf{tab} | \textbf{newline})^+$$

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

Figure 3.12: Tokens, their patterns, and attribute values

Transition Diagrams

- **States:** represents a condition
- **Edges:** directed from one state to another
- **Some Conventions:**
 1. Accepting or final states
 2. *: retract the forward pointer one position
 3. Start or initial state

Transition Diagrams for \geq

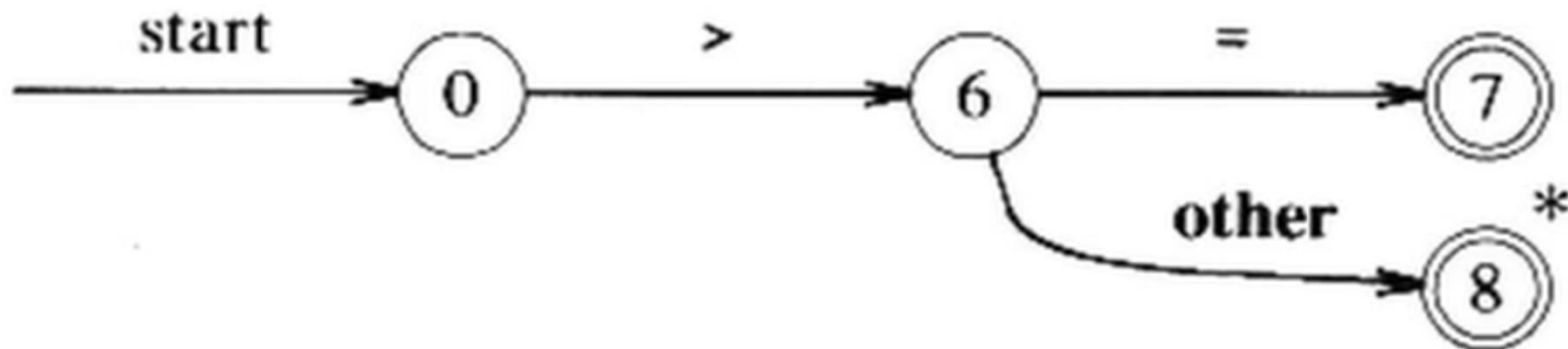


Fig. 3.11. Transition diagram for \geq .

- *start* state : state 0 in the above example
- If input character is $>$, go to state 6.
- **other** refers to any character that is not indicated by any of the other edges leaving s .

Transition Diagrams for Relational Operators

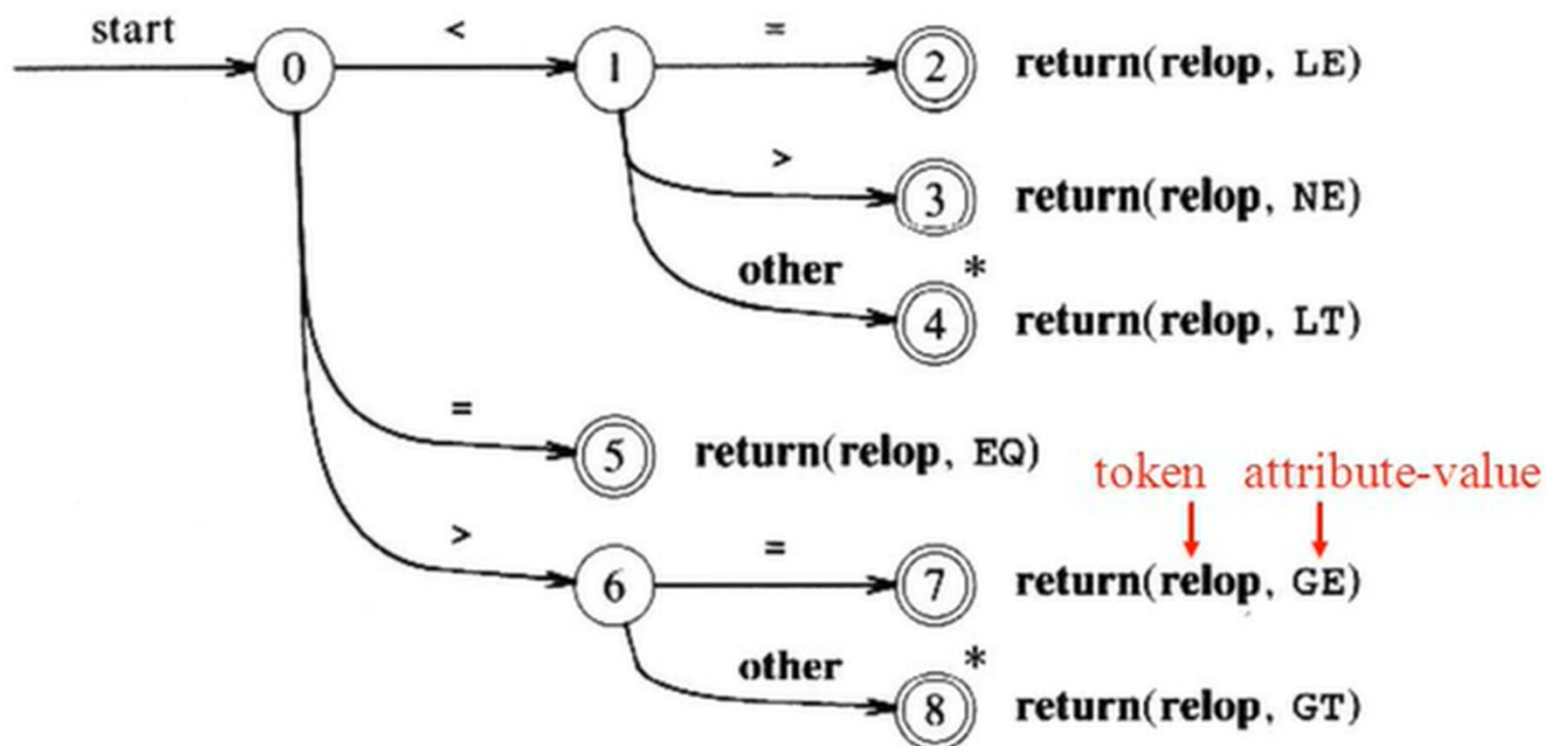


Fig. 3.12. Transition diagram for relational operators.

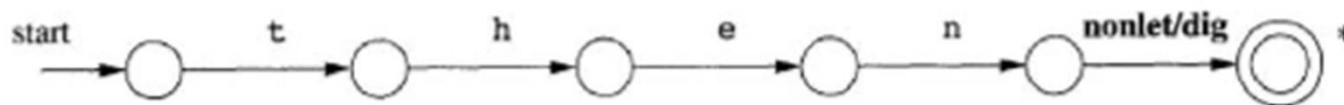
Recognition of Reserved Words and Identifiers

Two ways to handle reserved words:

- Install the reserved words in the symbol table initially

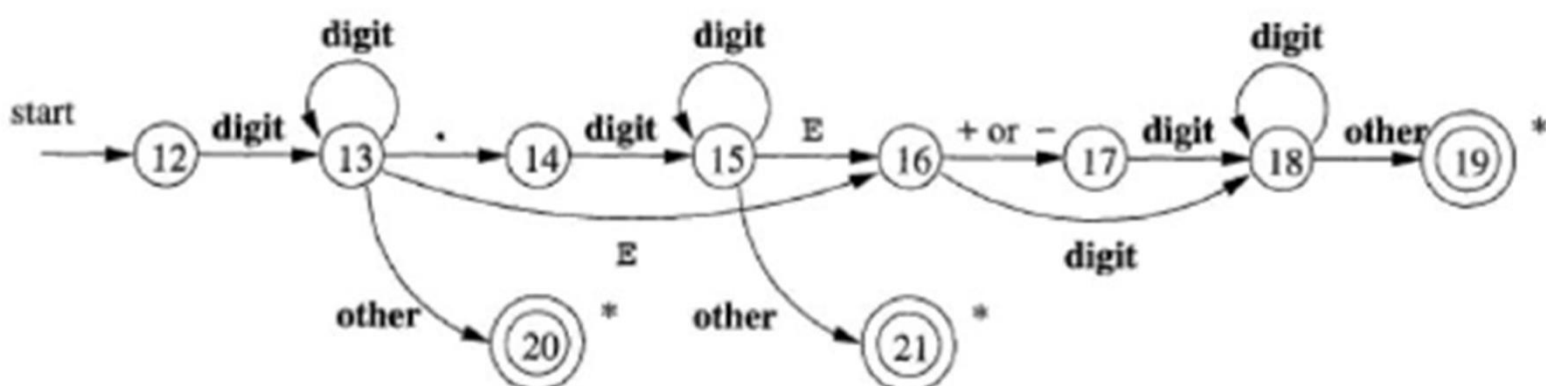


Create separate transition diagrams for each keyword

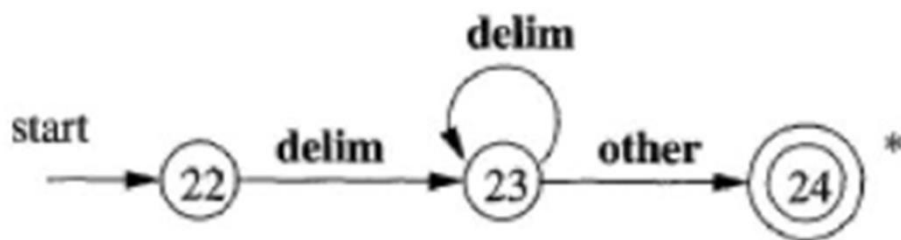


- `gettoken()`: return token (**id**, **if**, **then**,...) if it looks the symbol table
- `install_id()`: return 0 if keyword or a pointer to the symbol table entry if **id**

- Transition diagram for token number



- Transition diagram for whitespace



Implement a Transition Diagrams

- A sequence of transition diagrams can be converted into a program to look for tokens.
- Each state gets a segment of code.

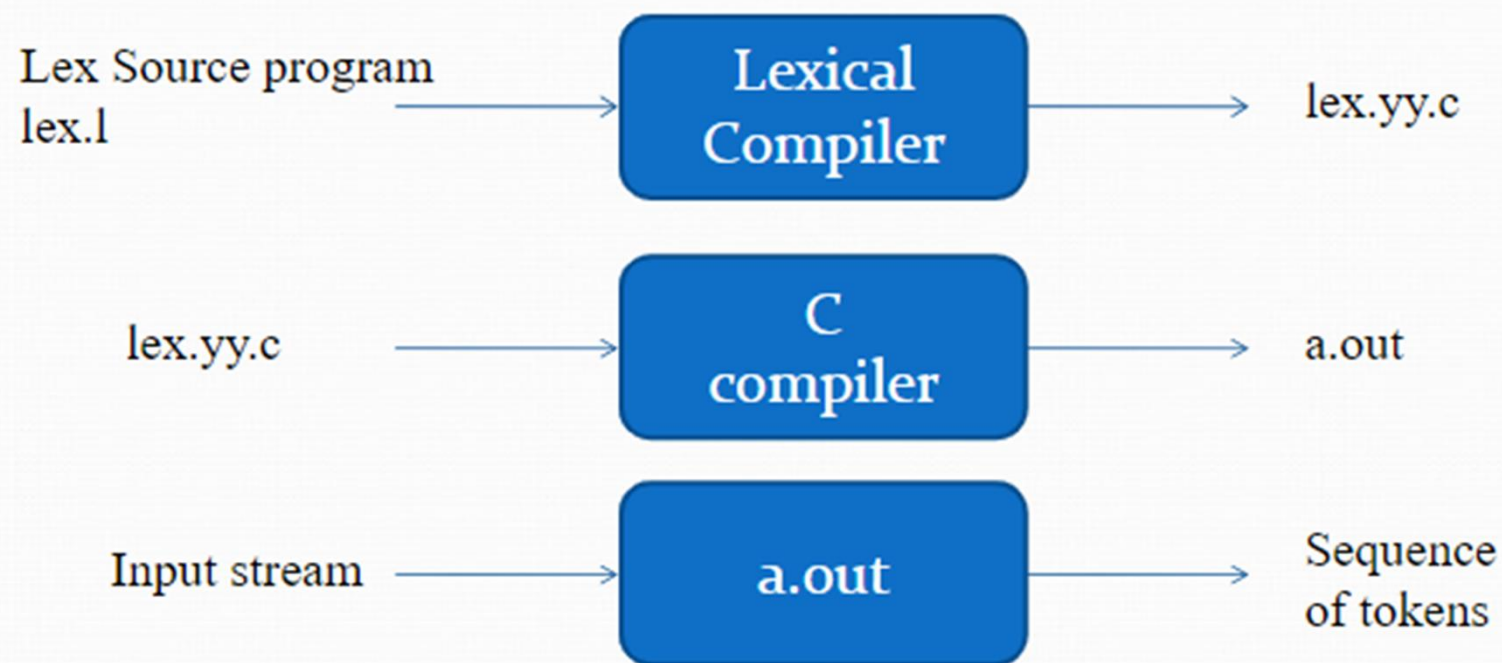
- `state` and `start` record the current state and the start state of current transition diagram.
- `lexical_value` is assigned the *pointer* returned by `install_id()` and `install_num()` when an identifier or number is found.
- When a diagram fails, the function `fail()` is used to *retract the forward pointer* to the position of the *lexeme beginning pointer* and to return the start state of the next diagram. If all diagrams fail the function `fail()` calls an error-recovery routine.


```
int state = 0, start = 0;
int lexical_value;
    /* to "return" second component of token */

int fail()
{
    forward = token_beginning;
    switch (start) {
        case 0:    start = 9; break;
        case 9:    start = 12; break;
        case 12:   start = 20; break;
        case 20:   start = 25; break;
        case 25:   recover(); break;
        default:   /* compiler error */
    }
    return start;
}
```

- Ways code fit into the entire lexical analyzer
 1. Arrange for the transition diagrams for each token to be tried sequentially
 2. Run the various transition diagrams "in parallel"
 3. Combine all the transition diagrams into one (preferred)

Lexical Analyzer Generator - Lex



- The lex tool is a widely used lexical analyzer generator.
- It helps in automatically generating a lexical analyzer (scanner) for processing input text based on user-defined patterns.
- The main purpose of lex is to simplify and automate the process of tokenizing input data, which is an essential first step in many programs like compilers, interpreters, and text processors.

- An input file, which we call **lex.1**, is written in the Lex language and describes the lexical analyzer to be generated.
- The Lex compiler transforms **lex.1 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 called **a. out**, as always.
- The C-compiler output is a working lexical analyzer that can take a stream of input characters and produce a stream of tokens.
- 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 Lex programs

declarations

%%

translation rules



Pattern {Action}

%%

auxiliary functions

- 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**.
- The translation rules each have the form
$$\text{Pattern}\{ \text{Action} \}$$
- Each **pattern is a regular expression**, which may use the regular definitions of the declaration section.
- The **actions are fragments of code**, typically written in C, although many variants of Lex using other languages have been created.
- The **third section holds whatever additional functions** are used in the actions.
- Alternatively, these functions can be compiled separately and loaded with the lexical analyzer.

- When called by the parser, the lexical analyzer begins reading its remaining input, one character at a time, until it finds the longest prefix of the input that matches one of the patterns P.
- It then executes the associated **action A**.
- **Typically, A**, will return to the parser, but if it does not (e.g., because P describes whitespace or comments), then the lexical analyzer proceeds to find additional lexemes, until one of the corresponding actions causes a return to the parser.
- The lexical analyzer returns a single value, the token name, to the parser, but uses the shared, integer variable `yylval` to pass additional information about the lexeme found, if needed.


```

%{
    /* 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); }

```

```
"="      {yyval = EQ; return(RELOP);}
"<>"    {yyval = NE; return(RELOP);}
">"      {yyval = GT; return(RELOP);}
">="    {yyval = 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 */
}
```

The action taken when *id* is matched is

1. Function `installID()` is called to place the lexeme found in the symbol table.
2. This function returns a pointer to the symbol table, which is placed in global variable `yylval`, where it can be used by the parser or a later component of the compiler. Note that `installID()` has available to it two variables that are set automatically by the lexical analyzer that **Lex** generates:
 - (a) `yytext` is a pointer to the beginning of the lexeme, analogous to `lexemeBegin` in Fig. 3.3.
 - (b) `yyleng` is the length of the lexeme found.
3. The token name `ID` is returned to the parser.

The action taken when a lexeme matching the pattern *number* is similar, using the auxiliary function `installNum()`. □

EXPRESSION	MATCHES	EXAMPLE
c	the one non-operator character c	a
$\backslash c$	character c literally	$\backslash *$
$"s"$	string s literally	$"**"$
$.$	any character but newline	$a.*b$
$^$	beginning of a line	abc
$\$$	end of a line	$abc\$$
$[s]$	any one of the characters in string s	$[abc]$
$[^s]$	any one character not in string s	$[^abc]$
r^*	zero or more strings matching r	a^*
r^+	one or more strings matching r	a^+
$r^?$	zero or one r	$a^?$
$r\{m,n\}$	between m and n occurrences of r	$a[1,5]$
r_1r_2	an r_1 followed by an r_2	ab
$r_1 \mid r_2$	an r_1 or an r_2	$a \mid b$
(r)	same as r	$(a \mid b)$
r_1/r_2	r_1 when followed by r_2	$abc/123$

Figure 3.8: Lex regular expressions