UNIT - I

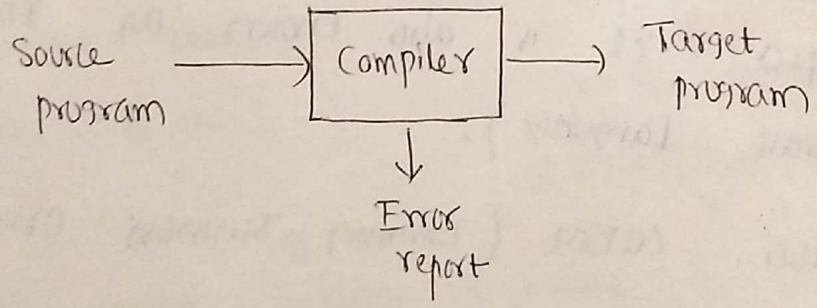
Introduction to Compiling

Compiler :

* A Compiler is a program that reads a source program written in one language (i.e) Source language (High level language) and translates it into an equivalent program in another language (i.e) Target language (Low level language)

(or)

A compiler is a piece of code that translates the high level language into machine language (or) low level language.



* After compilation, compiler generates an error report whenever an error occurs in the source program.

History of the compiler :-

* The term "compiler" was coined by "Grace Murray Hopper".
* In earlier days, computers have very limited memory

capacity and also created many technical problems when implementing a compiler.

* The first compiler was written by "Grace Murray Hopper" in 1952 for A-0 programming language (Arithmetic language version-0).
* In 1957, the FORTRAN (Formula Translating System) compiler was developed by IBM, lead by john Backus Team.
* The first compiler which was written by Grace Murray Hopper was "Incomplete Compiler".
* FORTRAN was the first complete compiler.
* In 1958, ALGOL 58 (Algorithmic Language) was developed. It is also known as IAL (International Algebraic Language).
* In 1960, COBOL (common Business Oriented language) compiler was developed.

Differences between Compiler and Interpreter :-

Compiler	Interpreter
(i) It takes the entire program as input.	(i) It takes a single instruction as input.

(ii) It generates an error report after compilation, whenever an error occurs.

(iii) By using compiler, control statements are executed faster.

(iv) It takes a large amount of time in analysing & processing the source program.

(v) Memory requirement is more.

Eg:- C Compiler, Java Compiler, C# etc.

(ii) It stops the translation after it gets the first error.

(iii) By using interpreter, control statements are executed slower.

(iv) It takes lesser time in the same process.

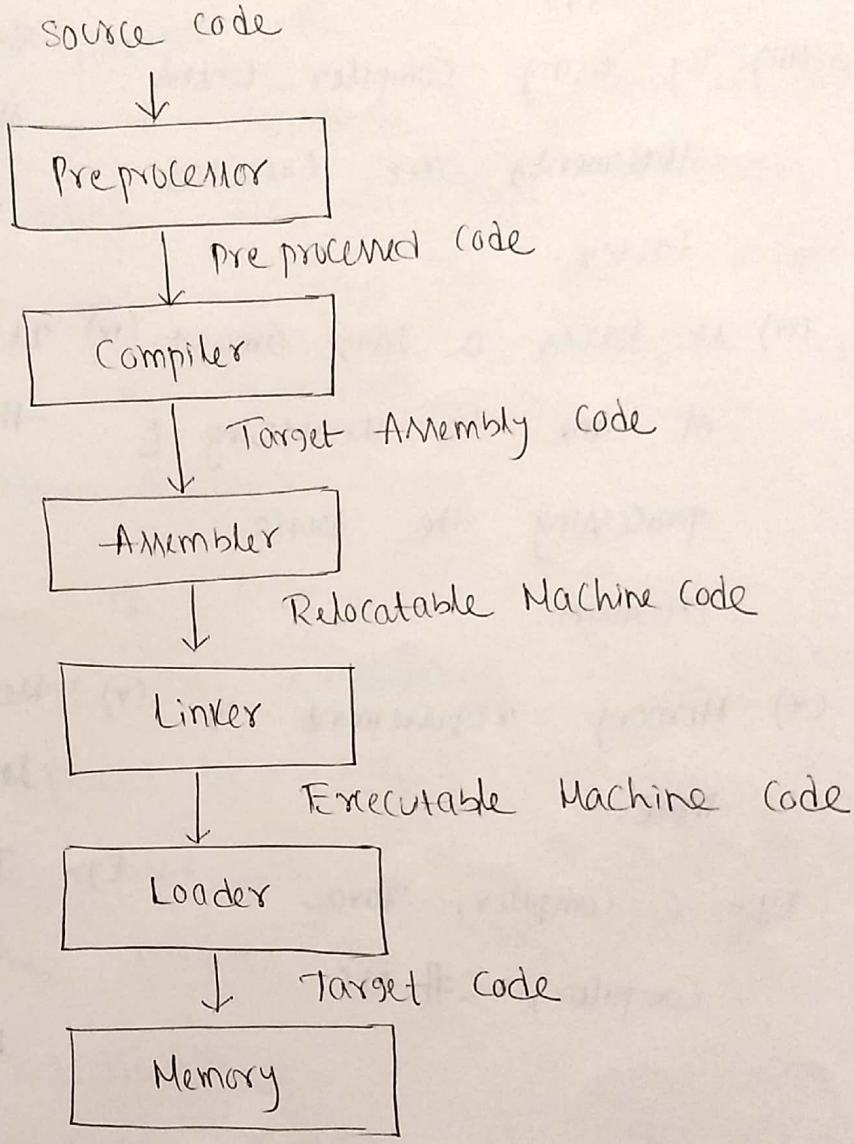
(v) Memory requirement is less.

Eg:- BASIC (Beginners All purpose Symbolic Instruction Code), LISP (List Processing), etc.

Language Processing System :- (Cousins of Compiler)

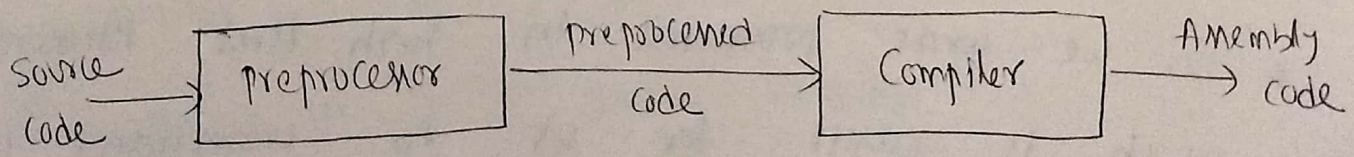
* Computer system is made of hardware and software.
* The hardware understands a language, which humans cannot understand.
* So we write programs in high-level language, which is easier for us to understand and remember.

* These programs are then fed into a series of tools and OS components to get the desired code that can be used by the machine.



(i) Preprocessor :-

* A preprocessor is considered as a part of compiler, that produces input for the compiler.



* The output of preprocessor given as input to the compiler.

* The tasks performed by the preprocessor are :-

(a) It allows the user to use "Macros" in the program. A Macro is a set of instructions which can be used repeatedly in the program. Whenever in the program, "Macro name" is identified then it is replaced by its "Macro definition".

Eg:- #define PI 3.14

whenever 'PI' is encountered in a program, it is replaced by '3.14'.

(b) It allows the user to use the "Header files", which may be required by the program.

Eg:- #include <stdio.h>

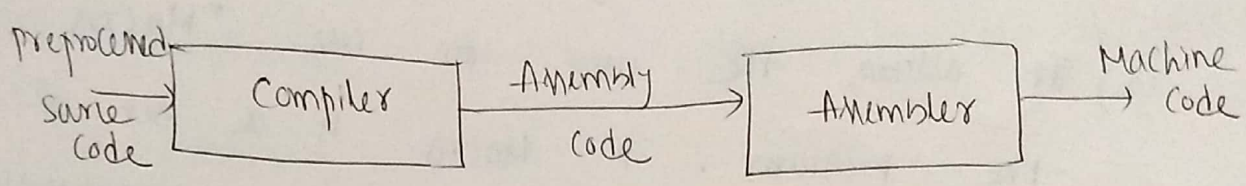
The header file contains different predefined functions. Those can be included by using preprocessor.

(ii) Compiler :-

A compiler is a program that converts high-level language in to assembly language.

(iii) Assembler :-

* An assembler translates the assembly language into machine code.



* Assembly code is a mnemonic version of machine code.
* The output of an assembler is an "Object file", which contains machine instructions as well as the data required to place the instructions in memory.

(iv) Linker :-

* Linker is a computer program that links and merges various object files together inorder to make an executable file.
* The major task of a linker is to search and locate referenced routines in a program and to determine the memory location where these codes will be loaded.

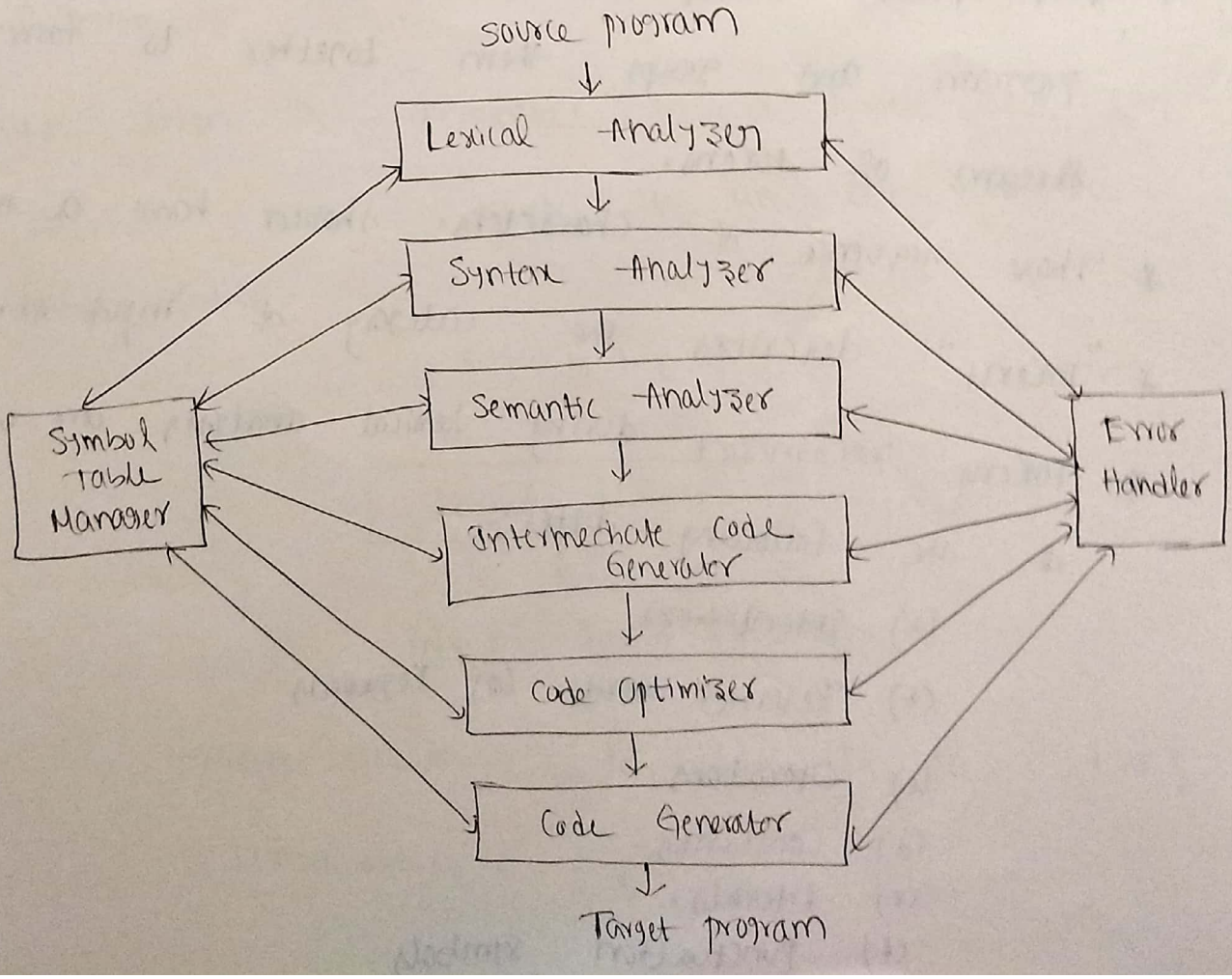
(v) Loader :-

* Loader is a part of operating system and it is responsible for loading executable files into memory and execute them.

* It calculates the size of program and creates memory space for it.

Phases of Compiler :-

* The compilation can be done in to 2 parts :-
  (i) Analysis.
  (ii) Synthesis.
* In "Analysis" part, the source program is read and broken down into pieces and the syntax & meaning of the source program is determined and then intermediate code is created.
* In "Synthesis" part, the intermediate code is taken and converted into target code.



* The analysis part consists of 3 phases:-

(i) Lexical Analysis

(ii) Syntax Analysis

(iii) Semantic Analysis

* The synthesis part consists of 3 phases:-

(i) Intermediate code Generator

(ii) Code optimizer.

(iii) Code Generator.

(i) Lexical Analysis :-

* This phase is also called as "Scanner" (or) "Tokenizer".

* This phase reads the characters in the source program and groups them together to form a stream of tokens.

* Those sequence of characters should have a meaning

* "Tokens" describes the category of input string.

* Tokens recognized during lexical analysis are usually of the following types:-

(a) Identifiers.

(b) Reserved words (or) keywords

(c) Operators

(d) Constants.

(e) Literals.

(f) Punctuation symbols.

Eg: int a = 100;

There are '5' tokens in the above statements.

* A "lexeme" is a sequence of characters that is matched by the pattern to represent a token.
* A "pattern" is a rule which describes a set of lexemes that can represent a token.

Eg:	Token	Lexeme	Pattern
	id	area, Pi, b2	letter followed by letters and digits

* The lexical analyzer removes white spaces (blank space, new line, tab) and comments from the source program.
* Each token is augmented by a "lexical value" (or) "Attribute value", which is an entry into the symbol table.

Eg:

$a := b + c * 60$

↓

Lexical Analyzer

↓

$id_1 := id_2 + id_3 * 60$

Token stream = { id_1, id_2, id_3, 60, =, +, * }

Lexical values = { 1, 2, 3 }

(ii) Syntax Analysis :-

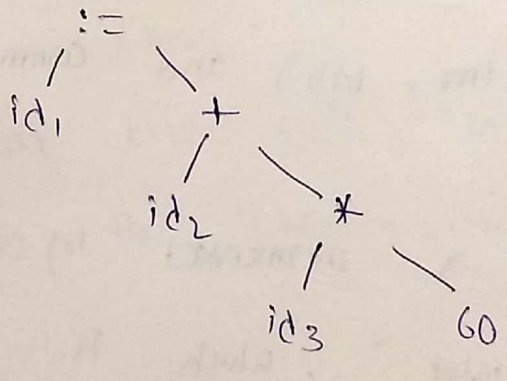
* This phase is also called as "parser".
* This phase takes the token stream as input and shows the structural representation of a program as output called "parse tree" (or) "syntax tree".
* This phase is responsible for identifying syntax errors in the source program.

$id_1 := id_2 + id_3 * 60$

↓

Syntax Analyzer

↓



* In the expression, "a = b + c * 60", multiplication is to be performed before addition because of "operator precedence".
* The hierarchical structure is formed by following Recursive rules (or) production rules.
* These rules are expressed by using CFG (Context Free Grammar)

* Content Free Grammar is a combination of

$G = (V, P, T, S)$ where,

V = Variables (or) Non-Terminals

P = Productions

T = Terminals

S = Start Symbol.

For the expression "a = b + c * 60", the CFG will be

$$S \rightarrow E = E \mid E + E \mid E * E$$

$$E \rightarrow id \mid num$$

* In the syntax tree, each "interior node" represents "Operators" and the "Children nodes" represents "Operands".

(iii) Semantic Analysis :-

* This phase uses the syntax tree produced by the syntax analyzer and checks the program consistency (i.e)

→ Correct no. of arguments to a function call.

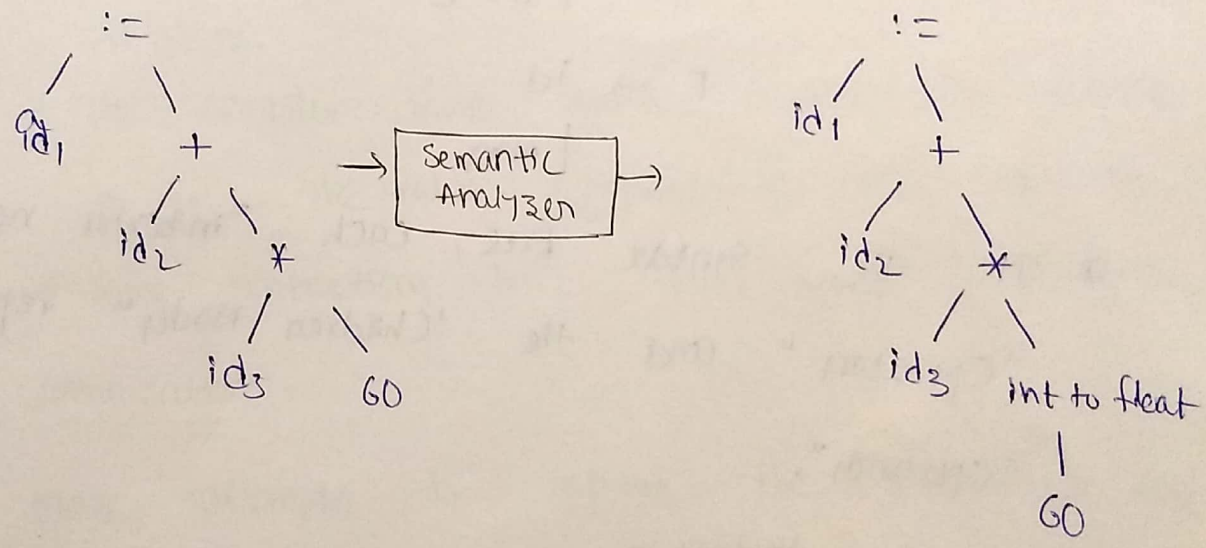
→ variables should be declared before used.

* This phase checks the source program for semantic errors and gathers information for next phases.

* "Type checking" is done by this phase.

Eg:- $a := b + c * 60 \Rightarrow id_1 := id_2 + id_3 * 60$

we assume that all identifiers are declared as "float" but '60' is an integer value. So, we need to convert the integer into float. This can be done by creating an extra node, convert integer to float value.



(iv) Intermediate Code Generator :-

* The intermediate code have 2 important properties:-
  - → It is easy to produce.
  - → Easy to translate into the target code.
* The intermediate code can be in different forms such as Three address code, quadruple, triple, posix etc.

* We consider an intermediate code as "Three address code", which is like assembly language.
* Every memory location act as a register.
* The "three address code" consists of instructions, each of which has atmost 3 operands.
* Properties of Three address code :-
  - → Each instruction has atmost one operator other than assignment operator.
  - → The compiler has to decide the order of operations.
  - → The compiler must generate a temporary variable to hold the value computed by each instruction.
  - → some instructions have fewer than 3 operands.

(v) Code Optimization :-

* This phase attempts to improve the intermediate code so that "Faster running machine code" will be produced.
* By optimizing the code, overall running time of the target program can be improved and less consumption of memory.

Ex:- $id_1 = id_2 + id_3 * 60$

The intermediate code for the above statements is

temp1 = int to float (60)

temp2 = id_3 * temp1

temp3 = id_2 + temp2

id_1 = temp3

temp1 = int to float (60)
temp2 = id_3 * temp1
temp3 = id_2 + temp2
id_1 = temp3

→ Code Optimizer →

temp1 = id_3 * 60.0
id_1 = id_2 + temp1

(vi) Code Generator:-

* This Phase contains machine code (or) Assembly code.
* Memory locations are selected for each variable used in the program.
* The intermediate code is translated into a sequence of machine instructions which performs the same task.

temp1 = id_3 * 60.0
id_1 = id_2 + temp1

→ Code Generator →

MOVF id_3, R_1
MULF #60.0, R_1
MOVF id_2, R_2
ADDF R_2, R_1
MOVF R_1, id_1

* The First and second operands in each instruction represents source & destination respectively.
* "F" in each instruction tells us those instructions deal with floating point numbers.
* The "#" indicates that "60.0" is treated as a constant value.

Symbol Table Management :-

* A symbol table is a data structure containing a record for each identifier and constant.
* this data structure allows us to find the record for each identifier quickly and to store (or) retrieve data from the record quickly.
* The lexical analyzer identifies the identifiers & constants in the source program and those information is entered into the symbol table.

Compiler Construction Tools :-

* Writing a compiler is tedious and time consuming task.
* There are some specialized tools which helps in implementation of various phases of a compiler.
* They are :-
  (i) Scanner generator.
  (ii) Parser generator.
  (iii) Syntax Directed translation engine.
  (iv) Automatic code generator.
  (v) Data flow engine.

(i) Scanner Generator :-

* This tool automatically generate lexical analyser phase from a specification based on "Regular Expression".
* A regular expression is a symbolic notation which describes a set of strings for a specific language over the input set.

Eg: Input set (Σ) = {0, 1}

A set of strings which ends with '10'

L = {10, 010, 110, 0010, 0110, ...}

Then, the regular expression for the above language is " $(0|1)^*10$

* UNIX operating system has the utility for generating a lexical analyzer called "LEX".

(ii) Parser Generator :-

* This tool generates "Syntax Analyzer" phase.
* The specification given to this tool is in the form of CFG (Context Free Grammar)
* A Context Free Grammar is :-
  - → a set of terminal symbols.
  - → a set of non-terminal symbols.
  - → a set of productions. non-terminal symbols are replaced with other non-terminal (or) terminal symbols.
  - → a start symbol, which is a non-terminal symbol.

Eg: Consider an expression

$$a+b*c-d$$

The CFG for the above expression is

$$S \rightarrow a \mid b \mid c \mid S+S \mid S-S \mid S*S$$

Check the grammar, whether the given grammar generates the given expression (or) not.

$$S \rightarrow S+\underline{S}$$
$$\rightarrow S+S*\underline{S}$$
$$\rightarrow S+S*S-\underline{S}$$
$$\rightarrow S+S*\underline{S}-d$$
$$\rightarrow S+\underline{S}*c-d$$

→ S + b * c - d

→ a + b * c - d

* UNIX operating system has the utility for generating the syntax analyser called "YACC" (Yet Another Compiler Compiler)

(iii) Syntax Directed Translation engine :-

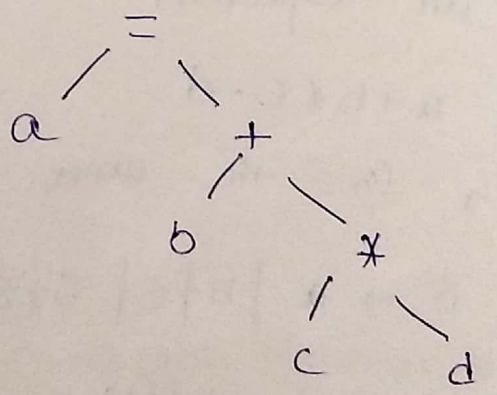
* By using this tool, the parse tree is scanned completly to generate "Intermediate code".

* The translation is done for each node of the tree.

Eg:- Consider a statement

$$a = b + c * d$$

The syntax tree for the above statement is



The intermediate code for the above syntax tree is

temp1 = c * d

temp2 = b + temp1

a = temp2

(iv) Automatic Code Generator:-

* This tool takes an intermediate code as input convert it into machine code
* In this tool, "Template Matching" technique is used.
* The intermediate code statements are replaced by the templates, which represents machine instructions.

(v) Data Flow Engine:-

* This tool is used for "Code optimization".
* To perform good code optimization, we need data flow analysis.
* This tool gathers information about how values are transmitted from one part of the program to the other.

Types of compiler:

(i) Incremental compiler:-

This type of compiler performs recompilation of only modified source code rather than compiling the whole source program.

(ii) Cross Compiler:

A compiler which runs on one machine and produces the target code for another machine.

A Simple One-pass Compiler

* In general, compiler is a program that reads a program written in one language, which is called the "source language" and translates it in to another language, which is called the "Target language".
* Traditionally, source language was a "High-level language" and target language was a "Low level language".
* "Pass" and "Phase" are 2 terms often used with Compilers.
* A "pass" refers to the traversal of a compiler through the entire program.
* No. of "passes" of a compiler is the no. of times it goes over the source program.
* A "phase" is a single-independent part of a Compiler.
* A standard way to classify compilers is by the no. of passes.
* Initially, computers did not have enough memory to hold such a program that did the complete job.
* Due to this limitation, Compilers were broken down into smaller sub programs that did its partial job over the source code.

* "One-pass Compilers" Compiler the entire source program in a single pass.
* It is easier to write a one-pass compiler and they perform faster than "Multi-pass Compilers".
* A Multi-pass compiler is made up of several stages. Each stage is considered as a "phase".
* The advantage of having different phases is that the development of compiler can be distributed.
* It improves the modularity and reuse by allowing phases to be replaced by improved ones (or) additional phases to the compiler.

Context Free Grammar (CFG):-

* A grammar is used to specify the syntax of a language.
* A grammar describes the hierarchical structure of any programming language.
* The Context Free Grammar has 4 components :-
  (a) A set of terminal symbols.
  (b) A set of non-terminal symbols.
  (c) A set of productions, where each production consists of :-

→ a non-terminal on left-hand side of a production.

→ an arrow and

→ a sequence of terminals (or) non-terminals on right-hand side of a production.

Eg: Consider a statement

a + b - c

The CFG for the above statement

E → E+E

| E-E

E → a | b | c

check whether the above statement is generated from the grammar or not.

E → E + E

→ E + E - E

→ E + E - c

→ E + b - c

→ a + b - c

Derivation :-

* It is the process of deriving an input string from the given grammar.
* A grammar derives strings by begining with the start symbol and repeatedly replacing a non-terminal by the body of a production.
* The geometrical representation of derivation is called "Derivation Tree".
* Derivations are of 2 types:-
  (i) Left-most derivation.
  (ii) Right-most derivation.

(i) Left-most derivation :-

The process of constructing a parse tree by expanding the left most non-terminal first in the derivation process is called "Left most derivation".

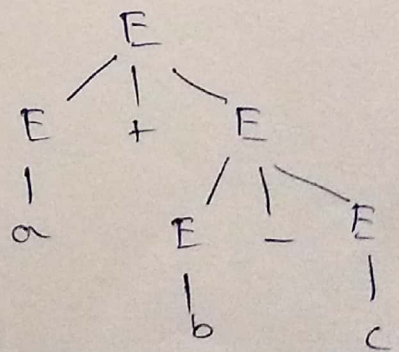
Eg:- CFG :- $E \rightarrow E+E \mid E-E \mid a \mid b \mid c$

input :- a+b-c

Derivation :-

$E \rightarrow \underline{E}+E$
$\rightarrow a+\underline{E}$
$\rightarrow a+\underline{E}-E$
$\rightarrow a+b-\underline{E}$
$\rightarrow a+b-c$

Parse tree :-



ii) Right-most Derivation :-

The process of constructing a parse tree by expanding the right-most non-terminal first in the derivation process is called "Right-most Derivation".

Eg: CFG: $E \to E+E \mid E-E \mid a \mid b \mid c$

input :- a+b-c

Derivation :-

$E \to E+\underline{E}$

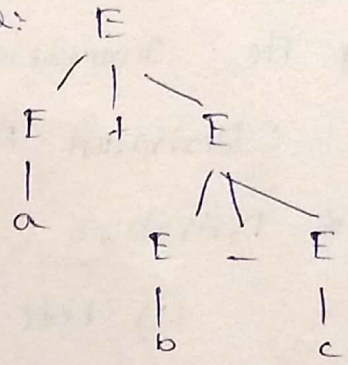
$\to E+E-\underline{E}$

$\to E+\underline{E}-c$

$\to \underline{E}+b-c$

$\to a+b-c$

Parse tree:



Eg: $E \to E+E \mid E*E \mid E-E \mid a \mid b \mid c \mid d$

input: a+b*c-d

RMD :- $E \to E+\underline{E}$

$\to E+E*\underline{E}$

$\to E+E*E-\underline{E}$

$\to E+E*\underline{E}-d$

$\to E+\underline{E}*c-d$

$\to \underline{E}+b*c-d$

$\to a+b*c-d$

LMD: $E+\underline{E}+E$

$\to a+\underline{E}$

$\to a+\underline{E}*E$

$\to a+b*\underline{E}$

$\to a+b*\underline{E}-E$

$\to a+b*c-\underline{E}$

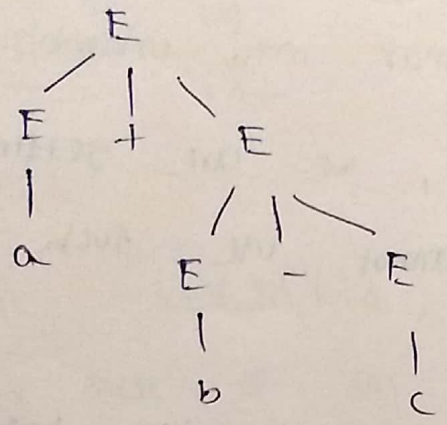
$\to a+b*c-d$

Ambiguous Grammar :-

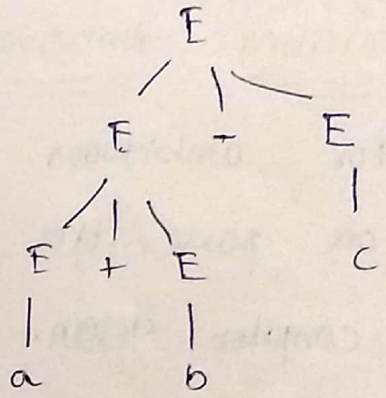
A grammar is said to be ambiguous, if it generates more than one parse tree.

Eg: $E \rightarrow E + E$
$\quad | \; E - E$
$\quad | \; a \; | \; b \; | \; c$

input :- a + b - c



(i)



(ii)

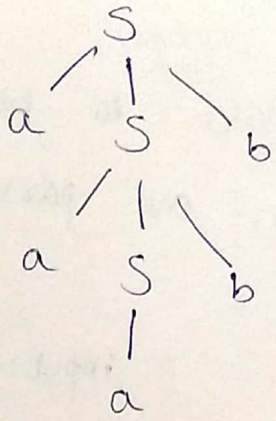
The above grammar generates 2 parse trees for the given input string. Hence, the given grammar is Ambiguous.

Unambiguous grammar :-

A grammar is said to be unambiguous, if it generates only one parse tree.

Eg: $S \rightarrow aSb$
$\quad | \; a$

input : aaabb



The above grammar generates only one parse tree. Hence, the given grammar is unambiguous.

Converting Ambiguous grammar into unambiguous grammar:-

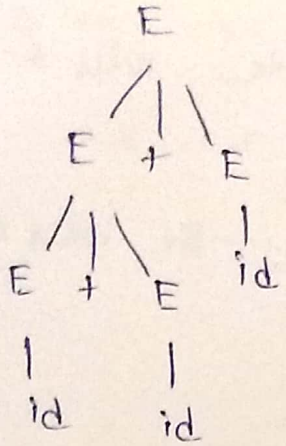
* For ambiguous grammar, we are getting more than one parse tree. We cannot use such grammar in compiler design.
* Ambiguous grammar can be converted into unambiguous grammar by following "Disambiguity Rule".
* If we see what are the reasons for ambiguity and if we work on that root cause we can have unambiguous grammar.

Eg: $E \rightarrow E + E$
$\mid E * E$
$\mid id$

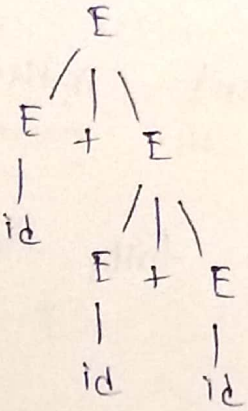
input 1 :- id + id + id

input 2 :- id + id * id

Input1 :- id + id + id

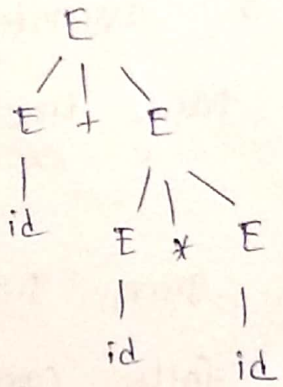


(i)
(✓)

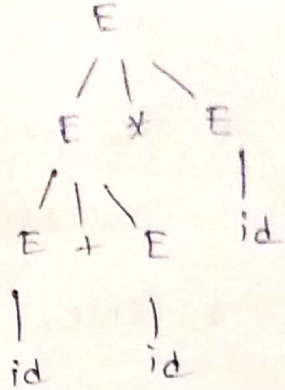


(ii)
(X)

Input2 :- id + id * id



(i)
(✓)



(ii)
(X)

For input1 :-

* we have "id + id + id", whenever there are 2 operators on either side of an operand, which operator should associate with the operand.

(id + (id) + id)

↙ Left associative

↘ Right associative

* In case of "+", the final output doesn't depend on what the actual associativity is.
* But, if we consider some other operators other than '+', which have equal precedence then it should be "left associative".
* In the first parse tree, the left most '+' is evaluated first. In the second parse tree, the right most '+' is evaluated first.

* The first parse tree is the correct parse tree based on rules of associativity.
* The second parse tree doesn't satisfy the rules of associativity.
* Hence, the given grammar fails because the rules of associativity fails completly.

For input2 :-

* we got 2 parse trees for the input "id+id*id".
* when we have different operators of different precedence present in an expression, the highest precedence operator should be evaluated first.
* In the first parse tree, the '*' is evaluated first and then '+' is evaluated
* In the second parse tree, the '+' is evaluated first and then '*' is evaluated.
* Among those 2 parse trees, the first parse tree is the correct parse tree which satisfies the "Operator precedence rules".
* Hence, the given grammar fails because the operator precedence is not taken care.
* If we take care of both operator precedence & associativity then the grammar is unambiguous.

Solution for Associativity problem :-

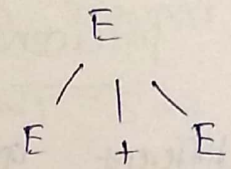
* Why we are getting both associativity ?

We are defining the grammar without any order.

$E \rightarrow E+E$
$\quad | E*E$
$\quad | id$

input :- id + id + id

* According to the grammar,

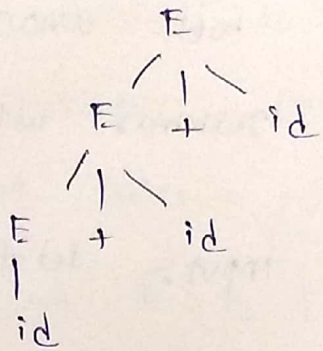


If we want one more '+', we could go either left side (or) right side.

* If we grow the parse tree in left direction, then it is left associativity.
* If we grow the parse tree in right direction, then it is right associativity.
* So, we need to restrict the grammar to grow the parse tree in only one direction (i.e) left
* Rewrite the grammar to have left associativity. Then,

$E \rightarrow E+id$
$\quad | id$

input :- id + id + id



* The given grammar is defined as left recursive, which grows the parse tree in left direction only.

Solution for operator precedence problem :-

* when there are different operators of different operator precedences, the highest precedence operator should be at bottom (or) least level. Then only it is evaluated first.

* To fix this precedence problem, we need to define the grammar to be of different levels.

$E \rightarrow E+E \mid E*E \mid id$

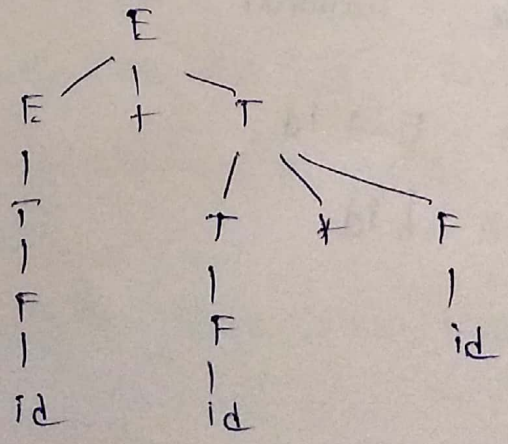
$\Rightarrow$

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow id$

input :- id + id * id



Recursive Grammar :-

A grammar is said to be recursive, if atleast one of the production contains same non-terminal symbols on both sides.

Eg:- (i) $S \rightarrow ASB \mid \varepsilon$

$A \rightarrow a$

$B \rightarrow b$

The given grammar is recursive because the production $S \rightarrow ASB$ contains same non-terminal symbol 'S' on both sides.

(ii) $S \rightarrow AB$

$A \rightarrow a$

$B \rightarrow b$

The given grammar is not recursive. Because, no production contains the same non-terminal symbols on both sides.

(iii) $S \rightarrow AB$

$A \rightarrow aB \mid a$

$B \rightarrow aA \mid b$

The given grammar is "Indirect recursive" because the production $A \rightarrow aB \rightarrow aaA$

derives the same non-terminal 'A' indirectly.

Non - recursive Grammar :-

A grammar is said to be non - recursive, if no production contains same non-terminal symbols on both sides

Eg:-

$S \rightarrow AB$

$A \rightarrow a$

$B \rightarrow b$

The given grammar is non - recursive because no production contains same non-terminal symbols on both sides.

Types of Recursive Grammar:-

Recursive Grammar are of 2 types:-

(i) Left recursive

(ii) Right recursive

(i) Left Recursive :-

A grammar is said to be left recursive, if the left most non-terminal symbol on right hand side of a production is equal to the non-terminal symbol on LHS of production.

Eg:- $S \rightarrow Sa$
$\quad | b$

The given grammar is left recursive

(ii) Right Recursive :-

A grammar is said to be right recursive, if the right most non-terminal symbol on RHS of a production is equal to the non-terminal symbol on LHS of a production.

Eg:- $S \rightarrow aS$
$\quad | \; a$

Left Recursion :-

* Because of left recursive grammar, the top down parser can enter in to "Infinite loop".

Eg:- $E \rightarrow E+T$

Every non-terminal have its own procedure which describes its production. In the above production, the procedure of 'E' calls its own procedure recursively. Then, infinite loop will be formed.

* A left recursive production can be eliminated by rewriting the production.

If the production is in the form

$$A \rightarrow A\alpha_1 \,|\, A\alpha_2 \,|\, \dots \,|\, A\alpha_m \,|\, \beta_1 \,|\, \beta_2 \,|\, \dots \,|\, \beta_n$$

where 'α' & 'β' are the sequence of terminals and non-terminals that doesn't start with 'A'.

We can eliminate the left recursion by rewriting the production rule as:-

$$A \to \beta_1 A' \mid \beta_2 A' \mid \dots \mid \beta_n A'$$

$$A' \to \alpha_1 A' \mid \alpha_2 A' \mid \dots \mid \alpha_m A' \mid \varepsilon$$

Eg:- $E \to E+T \mid T$

The production $E \to E+T$ is in the form of $A \to A\alpha \mid \beta$. where, $A = E$, $\alpha = +T$, $\beta = T$.

Then, the left recursion can be eliminated as follows:-

$$E \to E+T \mid T \quad \Rightarrow \quad E \to TE', \quad E' \to +TE' \mid \varepsilon$$

Eg:- $T \to T*F \mid F$

$A = T$, $\alpha = *F$, $\beta = F$

$$T \to T*F \mid F \quad \Rightarrow \quad T \to FT', \quad T' \to *FT' \mid \varepsilon$$

Eg:- $S \to S_0 S_1 S \mid 01$

$A = S$, $\alpha = 0S_1S$, $\beta = 01$

$$S \to S_0 S_1 S \mid 01 \quad \Rightarrow \quad S \to 01S', \quad S' \to 0S_1SS' \mid \varepsilon$$

Eg:-

$S \rightarrow (L) \mid x$

$L \rightarrow L, S \mid S$

$A = L \;,\; \alpha = , S \;,\; \beta = S$

$S \rightarrow (L) \mid x$

$L \rightarrow L, S \mid S$

$\Rightarrow$

$S \rightarrow (L) \mid x$

$L \rightarrow SL'$

$L' \rightarrow , SL' \mid \epsilon$

Non-Deterministic Grammar:-

A grammar is said to be non-deterministic, if a production having multiple forms which contains a common prefix.

$$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \dots \mid \alpha\beta_n$$

where α is the common prefix in multiple forms of a production

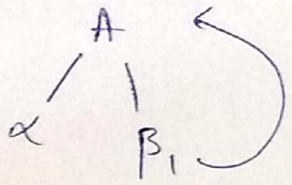
Problem of non-deterministic grammar:-

Consider an example

$$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \alpha\beta_3$$

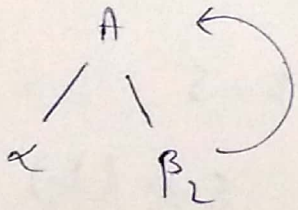
Input :- $\alpha\beta_3$

(i)



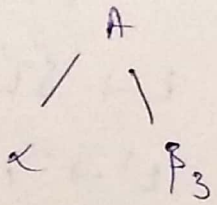
Backtracking

(ii)



Backtracking

(iii)



Matched.

* Backtracking is happening because of common prefix.
* Backtracking is happening because we are actually making a decision of which production to choose without seeing enough of the input
* Given input '$\alpha\beta_3$', we are making decision by seeing only 'α'.
* We have to postpone the decision making, this can be done by using "Left Factoring".

Left Factoring :-

* This is the process of converting non-deterministic grammar to deterministic grammar.
* Common prefixes are to be written separately.

* If the production is in the form :-

$$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \ldots \mid \alpha\beta_n$$

$$\Downarrow$$

$$A \rightarrow \alpha A'$$

$$A' \rightarrow \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$$

Eg:

$S \rightarrow iEtS$
$\mid iEtSeS$ where $\alpha = iEtS$
$\mid a$
$E \rightarrow b$

By applying left factoring the grammar is converted as :-

$S \rightarrow iEtSS'$
$\mid a$
$S' \rightarrow eS$
$\mid \epsilon$
$E \rightarrow b$

Eg:

$S \rightarrow aSSbS$
$\mid aSaSb$
$\mid abb$ Here $\alpha = a$
$\mid b$

After applying left factoring then

$S \rightarrow aS'$

$\quad | b$

$S' \rightarrow SSbS$ Here $\alpha = S$

$\quad | SaSb$

$\quad | bb$

Apply left Factoring again. Then,

$S \rightarrow aS'$

$\quad | b$

$S' \rightarrow SS''$

$\quad | bb$

$S'' \rightarrow SbS$

$\quad | aSb$

Eg: $S \rightarrow bSSaaS$

$| bSSaSb$ Here $\alpha = bS$, Then

$| bSb$

$| a$

$S \rightarrow bSS' | a$

$S' \rightarrow SaaS$ Here $\alpha = Sa$, Then

$\quad | SaSb$

$\quad | b$

$S \rightarrow bSS' | a$

$S' \rightarrow SaS'' | b$

$S'' \rightarrow aS | Sb$

Parsing:-

* It is the process of determining if a string of tokens can be generated by a grammar.
* The parser scans the input string from left to right and identifies that derivation is leftmost or rightmost.
* The parser use production rules for choosing the appropriate derivation. Finally a parse tree is constructed.
* When the parse tree can be constructed from root and expanded to leaves. Such type of parsing is called "Top down parsing".
* When the parse tree can be constructed from leaves to root, such type of parsing is called "Bottom parsing"

2.5 A TRANSLATOR FOR SIMPLE EXPRESSIONS

Using the techniques of the last three sections, we now construct a syntax-directed translator, in the form of a working C program, that translates arithmetic expressions into postfix form. To keep the initial program manageably small, we start off with expressions consisting of digits separated by plus and minus signs. The language is extended in the next two sections to include numbers, identifiers, and other operators. Since expressions appear as a construct in so many languages, it is worth studying their translation in detail.

A syntax-directed translation scheme can often serve as the specification for a translator. We use the scheme in Fig. 2.19 (repeated from Fig. 2.13) as the definition of the translation to be performed. As is often the case, the underlying grammar of a given scheme has to be modified before it can be parsed with a predictive parser. In particular, the grammar underlying the scheme in Fig. 2.19 is left-recursive, and as we saw in the last section, a predictive parser cannot handle a left-recursive grammar. By eliminating the left-recursion, we can obtain a grammar suitable for use in a predictive recursive-descent translator.

```
expr → expr + term    { print('+') }
expr → expr - term    { print('-') }
expr → term
term → 0              { print('0') }
term → 1              { print('1') }
  .
  .
  .
term → 9              { print('9') }
```

Fig. 2.19. Initial specification of infix-to-postfix translator.

Abstract and Concrete Syntax

A useful starting point for thinking about the translation of an input string is an *abstract syntax tree* in which each node represents an operator and the children of the node represent the operands. By contrast, a parse tree is called a *concrete syntax tree*, and the underlying grammar is called a *concrete syntax* for the language. Abstract syntax trees, or simply *syntax trees*, differ from parse trees because superficial distinctions of form, unimportant for translation, do not appear in syntax trees.

For example, the syntax tree for `9-5+2` is shown in Fig. 2.20. Since + and - have the same precedence level, and operators at the same precedence level are evaluated left to right, the tree shows `9-5` grouped as a subexpression. Comparing Fig. 2.20 with the corresponding parse tree of Fig. 2.2, we note that the syntax tree associates an operator with an interior node, rather than making the operator be one of the children.

It is desirable for a translation scheme to be based on a grammar whose parse trees are as close to syntax trees as possible. The grouping of subexpressions by the grammar in Fig. 2.19 is similar to their grouping in syntax trees. Unfortunately, the grammar of Fig. 2.19 is left-recursive, and hence not suitable for predictive

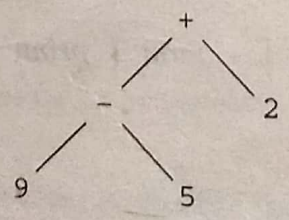


Fig. 2.20. Syntax tree for `9-5+2`.

Adapting the Translation Scheme

The left-recursion elimination technique sketched in Fig. 2.18 can also be applied to productions containing semantic actions. We extend the transformation in Section 5.5 to take synthesized attributes into account. The technique transforms the productions $A \rightarrow A\alpha \mid A\beta \mid \gamma$ into

$$A \rightarrow \gamma R$$
$$R \rightarrow \alpha R \mid \beta R \mid \epsilon$$

When semantic actions are embedded in the productions, we carry them along in the transformation. Here, if we let $A = expr$, $\alpha = +\ term\ \{\ print('+')\ \}$, $\beta = -\ term\ \{\ print('-')\ \}$, and $\gamma = term$, the transformation above produces the translation scheme (2.14). The *expr* productions in Fig. 2.19 have been transformed into the productions for *expr* and the new nonterminal *rest* in (2.14). The productions for *term* are repeated from Fig. 2.19. Notice that the underlying grammar is different from the one in Example 2.9 and the difference makes the desired translation possible.

$$\begin{aligned}
expr &\rightarrow term\ rest \\
rest &\rightarrow +\ term\ \{\ print('+')\ \}\ rest \mid -\ term\ \{\ print('-')\ \}\ rest \mid \epsilon \\
term &\rightarrow \texttt{0}\ \{\ print('\texttt{0}')\ \} \\
term &\rightarrow \texttt{1}\ \{\ print('\texttt{1}')\ \} \\
&\vdots \\
term &\rightarrow \texttt{9}\ \{\ print('\texttt{9}')\ \}
\end{aligned} \tag{2.14}$$

Figure 2.21 shows how `9-5+2` is translated using the above grammar.

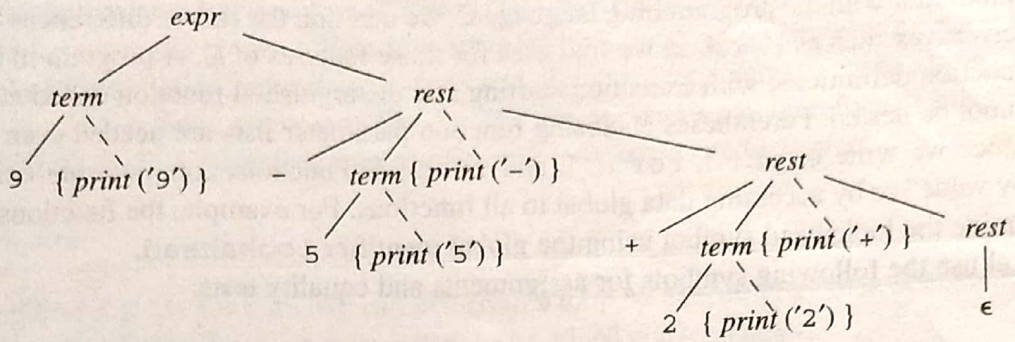


Fig. 2.21. Translation of `9-5+2` into `95-2+`.

Procedures for the Nonterminals *expr*, *term*, and *rest*

We now implement a translator in C using the syntax-directed translation scheme (2.14). The essence of the translator is the C code in Fig. 2.22 for the functions `expr`, `term`, and `rest`. These functions implement the corresponding nonterminals in (2.14).

```
expr()
{
    term(); rest();
}

rest()
{
    if (lookahead == '+') {
        match('+'); term(); putchar('+'); rest();
    }
    else if (lookahead == '-') {
        match('-'); term(); putchar('-'); rest();
    }
    else ;
}

term()
{
    if (isdigit(lookahead)) {
        putchar(lookahead); match(lookahead);
    }
    else error();
}
```

Fig. 2.22. Functions for the nonterminals *expr*, *rest*, and *term*.

The function `match`, presented later, is the C counterpart of the code in Fig. 2.17 to match a token with the lookahead symbol and advance through the input. Since each token is a single character in our language, `match` can be implemented by comparing and reading characters.

5. Lexical Analysis

The role of lexical analyzer :-

* lexical analyzer is the first phase of a compile
* It reads the input source program from left to right one character at a time and generates the sequence of tokens.
* Tokens recognized during lexical analysis are usually of the following types :-
  (i) Identifiers.
  (ii) keywords.
  (iii) constants.
  (iv) operators.
  (v) Symbols.
* The lexical analyzer interacts with the syntax analyzer and symbol table & Error report handler

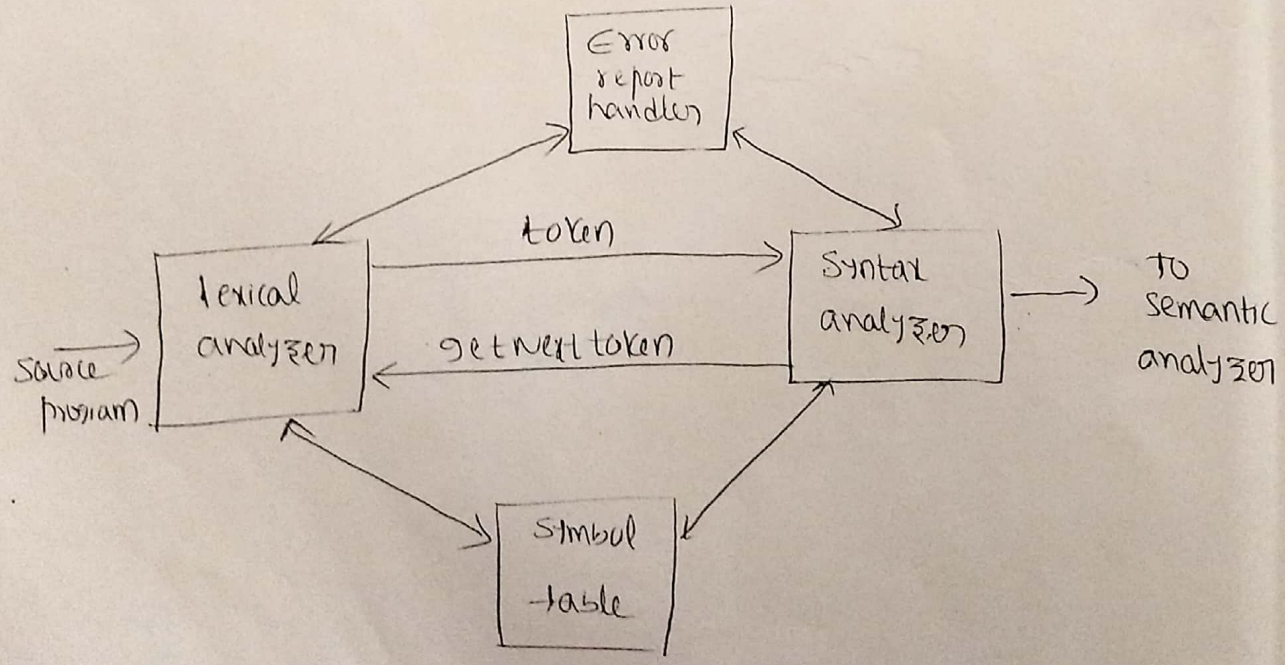


fig: lexical analyzer interaction with other phases

the stream of tokens is sent to the syntax analyzer.

A token is a pair consisting of token name & attribute value (or) lexical value, which describes the category of input string.

lexical value (or) attribute value is an entry into the symbol table.

The character sequence for forming a token is called the "lexeme" for a token.

lexical analyzer generates one token at a time & the remaining tokens gets generated whenever the syntax analyzer makes a request to the lexical analyzer.

"getnexttoken" command causes the lexical analyzer to generate next token which returns to the syntax analyzer.

The lexical analyzer is responsible for

→ removing comments and whitespaces.

→ produces stream of tokens.

Lexical analyzer keeps track of line numbers wherever a new line character (\n) is seen. this could be helpful in identifying the error in the program.

Eg:-

```
int max(int x, int y)
{
    return (x > y ? x : y);
}
```

No. of tokens = 22

lexeme	Token
int	Keyword
max	identifier
(	Symbol
int	Keyword
x	identifier
,	operator
int	Keyword
y	identifier
)	Symbol
{	Symbol
return	Keyword
(	Symbol
x	identifier
>	operator
y	identifier
?	operator
x	identifier
:	operator

lexeme	Token
y	identifier
)	Symbol
;	Symbol
}	Symbol

3.2 INPUT BUFFERING

This section covers some efficiency issues concerned with the buffering of input. We first mention a two-buffer input scheme that is useful when look-ahead on the input is necessary to identify tokens. Then we introduce some useful techniques for speeding up the lexical analyzer, such as the use of "sentinels" to mark the buffer end.

There are three general approaches to the implementation of a lexical analyzer.

1. Use a lexical-analyzer generator, such as the Lex compiler discussed in Section 3.5, to produce the lexical analyzer from a regular-expression-based specification. In this case, the generator provides routines for reading and buffering the input.
2. Write the lexical analyzer in a conventional systems-programming language, using the I/O facilities of that language to read the input.
3. Write the lexical analyzer in assembly language and explicitly manage the reading of input.

The three choices are listed in order of increasing difficulty for the implementor. Unfortunately, the harder-to-implement approaches often yield faster lexical analyzers. Since the lexical analyzer is the only phase of the compiler that reads the source program character-by-character, it is possible to spend a considerable amount of time in the lexical analysis phase, even though the later phases are conceptually more complex. Thus, the speed of lexical analysis is a concern in compiler design. While the bulk of the chapter is devoted to the first approach, the design and use of an automatic generator, we also consider techniques that are helpful in manual design. Section 3.4 discusses transition diagrams, which are a useful concept for the organization of a hand-designed lexical analyzer.

Buffer Pairs

For many source languages, there are times when the lexical analyzer needs to look ahead several characters beyond the lexeme for a pattern before a match can be announced. The lexical analyzers in Chapter 2 used a function `ungetc` to push lookahead characters back into the input stream. Because a large amount of time can be consumed moving characters, specialized buffering techniques have been developed to reduce the amount of overhead required to process an input character. Many buffering schemes can be used, but since the techniques are somewhat dependent on system parameters, we shall only outline the principles behind one class of schemes here.

We use a buffer divided into two N-character halves, as shown in Fig. 3.3. Typically, N is the number of characters on one disk block, e.g., 1024 or 4096.

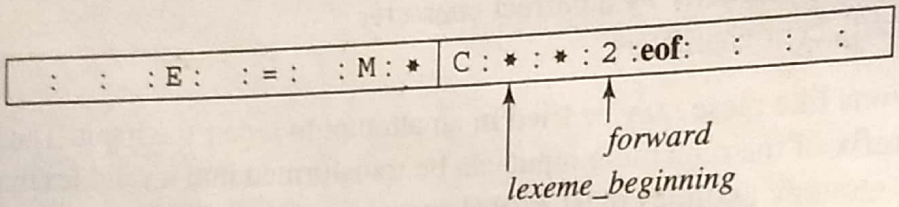


Fig. 3.3. An input buffer in two halves.

We read N input characters into each half of the buffer with one system read command, rather than invoking a read command for each input character. If fewer than N characters remain in the input, then a special character **eof** is read into the buffer after the input characters, as in Fig. 3.3. That is, **eof** marks the end of the source file and is different from any input character.

Two pointers to the input buffer are maintained. The string of characters between the two pointers is the current lexeme. Initially, both pointers point to the first character of the next lexeme to be found. One, called the forward pointer, scans ahead until a match for a pattern is found. Once the next lexeme is determined, the forward pointer is set to the character at its right end. After the lexeme is processed, both pointers are set to the character immediately past the lexeme. With this scheme, comments and white space can be treated as patterns that yield no token.

If the forward pointer is about to move past the halfway mark, the right half is filled with N new input characters. If the forward pointer is about to move past the right end of the buffer, the left half is filled with N new characters and the forward pointer wraps around to the beginning of the buffer.

This buffering scheme works quite well most of the time, but with it the amount of lookahead is limited, and this limited lookahead may make it impossible to recognize tokens in situations where the distance that the forward pointer must travel is more than the length of the buffer. For example, if we see

```
DECLARE ( ARG1, ARG2, ..., ARGn )
```

in a PL/I program, we cannot determine whether `DECLARE` is a keyword or an array name until we see the character that follows the right parenthesis. In either case, the lexeme ends at the second `E`, but the amount of lookahead needed is proportional to the number of arguments, which in principle is unbounded.

Sentinels

If we use the scheme of Fig. 3.3 exactly as shown, we must check each time we move the forward pointer that we have not moved off one half of the buffer; if we do, then we must reload the other half. That is, our code for advancing the forward pointer performs tests like those shown in Fig. 3.4.

```
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.

Except at the ends of the buffer halves, the code in Fig. 3.4 requires two tests for each advance of the forward pointer. We can reduce the two tests to one if we extend each buffer half to hold a *sentinel* character at the end. The sentinel is a special character that cannot be part of the source program. A natural choice is **eof**; Fig. 3.5 shows the same buffer arrangement as Fig. 3.3, with the sentinels added.

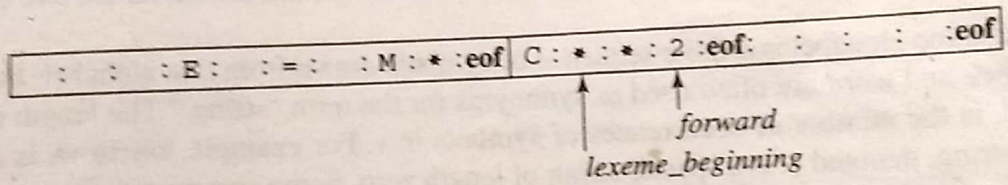


Fig. 3.5. Sentinels at end of each buffer half.

With the arrangement of Fig. 3.5, we can use the code shown in Fig. 3.6 to advance the forward pointer (and test for the end of the source file). Most of the time the code performs only one test to see whether *forward* points to an **eof**. Only when we reach the end of a buffer half or the end of the file do we perform more tests. Since N input characters are encountered between **eof**'s, the average number of tests per input character is very close to 1.

```
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
```

Fig. 3.6. Lookahead code with sentinels.

3.3 SPECIFICATION OF TOKENS

Regular expressions are an important notation for specifying patterns. Each pattern matches a set of strings, so regular expressions will serve as names for sets of strings. Section 3.5 extends this notation into a pattern-directed language for lexical analysis.

Strings and Languages

The term *alphabet* or *character class* denotes any finite set of symbols. Typical examples of symbols are letters and characters. The set {0, 1} is the *binary alphabet*. ASCII and EBCDIC are two examples of computer alphabets.

A *string* over some alphabet is a finite sequence of symbols drawn from that alphabet. In language theory, the terms *sentence* and *word* are often used as synonyms for the term "string." The length of a string s, usually written $|s|$, is the number of occurrences of symbols in s. For example, banana is a string of length six. The *empty* string, denoted ϵ, is a special string of length zero. Some common terms associated with parts of a string are summarized in Fig. 3.7.

The term *language* denotes any set of strings over some fixed alphabet. This definition is very broad. Abstract languages like $\emptyset$, the *empty* set, or $\{\epsilon\}$, the set containing only the empty string, are languages under this definition. So too are the set of all syntactically well-formed Pascal programs and the set of all grammatically correct English sentences, although the latter two sets are much more difficult to specify. Also note that this definition does not ascribe any meaning to the strings in a language. Methods for ascribing meanings to strings are discussed in Chapter 5.

TERM	DEFINITION
prefix of s	A string obtained by removing zero or more trailing symbols of string s; e.g., ban is a prefix of banana.
suffix of s	A string formed by deleting zero or more of the leading symbols of s; e.g., nana is a suffix of banana.
substring of s	A string obtained by deleting a prefix and a suffix from s; e.g., nan is a substring of banana. Every prefix and every suffix of s is a substring of s, but not every substring of s is a prefix or a suffix of s. For every string s, both s and ϵ are prefixes, suffixes, and substrings of s.
proper prefix, suffix, or substring of s	Any nonempty string x that is, respectively, a prefix, suffix, or substring of s such that $s \neq x$.
subsequence of s	Any string formed by deleting zero or more not necessarily contiguous symbols from s; e.g., baaa is a subsequence of banana.

Fig. 3.7. Terms for parts of a string.

If x and y are strings, then the *concatenation* of x and y, written xy, is the string formed by appending y to x. For example, if $x = \texttt{dog}$ and $y = \texttt{house}$, then $xy = \texttt{doghouse}$. The empty string is the identity element under concatenation. That is, $s\epsilon = \epsilon s = s$.

If we think of concatenation as a "product", we can define string "exponentiation" as follows. Define s^0 to be ϵ, and for $i > 0$ define s^i to be $s^{i-1}s$. Since ϵs is s itself, $s^1 = s$. Then, $s^2 = ss$, $s^3 = sss$, and so on.

Operations on Languages

There are several important operations that can be applied to languages. For lexical analysis, we are interested primarily in union, concatenation, and closure, which are defined in Fig. 3.8. We can also generalize the "exponentiation" operator to languages by defining L^0 to be $\{\epsilon\}$, and L^i to be $L^{i-1}L$. Thus, L^i is L concatenated with itself $i - 1$ times.

Example 3.2. Let L be the set $\{\texttt{A}, \texttt{B}, \ldots, \texttt{Z}, \texttt{a}, \texttt{b}, \ldots, \texttt{z}\}$ and D the set $\{\texttt{0}, \texttt{1}, \ldots, \texttt{9}\}$. We can think of L and D in two ways. We can think of L as the alphabet consisting of the set of upper and lower case letters, and D as the alphabet consisting of the set of the ten decimal digits. Alternatively, since a symbol can be regarded as a string of length one, the sets L and D are each finite languages. Here are some examples of new languages created from L and D by applying the operators defined in Fig. 3.8.

OPERATION	DEFINITION
union of L and M written $L \cup M$	$L \cup M = \{ s \mid s \text{ is in } L \text{ or } s \text{ is in } M \}$
concatenation of L and M written LM	$LM = \{ st \mid s \text{ is in } L \text{ and } t \text{ is in } M \}$
Kleene closure of L written L^*	$L^* = \bigcup_{i=0}^{\infty} L^i$ L^* denotes "zero or more concatenations of" L.
positive closure of L written L^+	$L^+ = \bigcup_{i=1}^{\infty} L^i$ L^+ denotes "one or more concatenations of" L.

Fig. 3.8. Definitions of operations on languages.

1. $L \cup D$ is the set of letters and digits.
2. LD is the set of strings consisting of a letter followed by a digit.
3. L^4 is the set of all four-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

In Pascal, an identifier is a letter followed by zero or more letters or digits; that is, an identifier is a member of the set defined in part (5) of Example 3.2. In this section, we present a notation, called regular expressions,

Regular Definitions

For notational convenience, we may wish to give names to regular expressions and to define regular expressions using these names as if they were symbols. If Σ is an alphabet of basic symbols, then a *regular definition* is a sequence of definitions of the form

$$
\begin{aligned}
d_1 &\rightarrow r_1 \\
d_2 &\rightarrow r_2 \\
&\cdots \\
d_n &\rightarrow r_n
\end{aligned}
$$

where each d_i is a distinct name, and each r_i is a regular expression over the symbols in $\Sigma \cup \{d_1, d_2, \ldots, d_{i-1}\}$, i.e., the basic symbols and the previously defined names. By restricting each r_i to symbols of Σ and the previously defined names, we can construct a regular expression over Σ for any r_i by repeatedly replacing regular-expression names by the expressions they denote. If r_i used d_j for some $j \geq i$, then r_i might be recursively defined, and this substitution process would not terminate.

To distinguish names from symbols, we print the names in regular definitions in boldface.

Example 3.4. As we have stated, the set of Pascal identifiers is the set of strings of letters and digits beginning with a letter. Here is a regular definition for this set.

letter → `A` | `B` | ··· | `Z` | `a` | `b` | ··· | `z`
digit → `0` | `1` | ··· | `9`
id → **letter** (**letter** | **digit**)*

Example 3.5. Unsigned numbers in Pascal are strings such as `5280`, `39.37`, `6.336E4`, or `1.894E-4`. The following regular definition provides a precise specification for this class of strings:

$$\begin{aligned}
\textbf{digit} &\rightarrow \texttt{0} \mid \texttt{1} \mid \cdots \mid \texttt{9} \\
\textbf{digits} &\rightarrow \textbf{digit}\ \textbf{digit}^* \\
\textbf{optional_fraction} &\rightarrow \texttt{.}\ \textbf{digits} \mid \epsilon \\
\textbf{optional_exponent} &\rightarrow (\ \texttt{E}\ (\ \texttt{+} \mid \texttt{-} \mid \epsilon\ )\ \textbf{digits}\ ) \mid \epsilon \\
\textbf{num} &\rightarrow \textbf{digits}\ \textbf{optional_fraction}\ \textbf{optional_exponent}
\end{aligned}$$

This definition says that an **optional_fraction** is either a decimal point followed by one or more digits, or it is missing (the empty string). An **optional_exponent**, if it is not missing, is an `E` followed by an optional `+` or `-` sign, followed by one or more digits. Note that at least one digit must follow the period, so **num** does not match `1.` but it does match `1.0`. □

Notational Shorthands

Certain constructs occur so frequently in regular expressions that it is convenient to introduce notational shorthands for them.

1. *One or more instances*. The unary postfix operator $^+$ means "one or more instances of." If r is a regular expression that denotes the language $L(r)$, then $(r)^+$ is a regular expression that denotes the language $(L(r))^+$. Thus, the regular expression a^+ denotes the set of all strings of one or more a's. The operator $^+$ has the same precedence and associativity as the operator $*$. The two algebraic identities $r^* = r^+ \mid \epsilon$ and $r^+ = rr^*$ relate the Kleene and positive closure operators.

2. *Zero or one instance*. The unary postfix operator ? means "zero or one instance of." The notation $r?$ is a shorthand for $r \mid \epsilon$. If r is a regular expression, then $(r)?$ is a regular expression that denotes the language $L(r) \cup \{\epsilon\}$. For example, using the $^+$ and ? operators, we can rewrite the regular definition for **num** in Example 3.5 as

$$\begin{aligned}
\textbf{digit} &\rightarrow \texttt{0} \mid \texttt{1} \mid \cdots \mid \texttt{9} \\
\textbf{digits} &\rightarrow \textbf{digit}^+ \\
\textbf{optional_fraction} &\rightarrow (\ \texttt{.}\ \textbf{digits}\ )? \\
\textbf{optional_exponent} &\rightarrow (\ \texttt{E}\ (\ \texttt{+} \mid \texttt{-}\ )?\ \textbf{digits}\ )? \\
\textbf{num} &\rightarrow \textbf{digits}\ \textbf{optional_fraction}\ \textbf{optional_exponent}
\end{aligned}$$

3. *Character classes*. The notation `[abc]` where `a`, `b`, and `c` are alphabet symbols denotes the regular expression `a | b | c`. An abbreviated character class such as `[a-z]` denotes the regular expression `a | b | ⋯ | z`. Using character classes, we can describe identifiers as being strings generated by the regular expression

```
[A-Za-z][A-Za-z0-9]*
```

3.4 RECOGNITION OF TOKENS

In the previous section, we considered the problem of how to specify tokens. In this section, we address the question of how to recognize them. Throughout this section, we use the language generated by the following grammar as a running example.

Example 3.6. Consider the following grammar fragment:

$$
\begin{aligned}
\textit{stmt} &\rightarrow \textbf{if}\ \textit{expr}\ \textbf{then}\ \textit{stmt} \\
&\quad \mid \textbf{if}\ \textit{expr}\ \textbf{then}\ \textit{stmt}\ \textbf{else}\ \textit{stmt} \\
&\quad \mid \epsilon \\
\textit{expr} &\rightarrow \textit{term}\ \textbf{relop}\ \textit{term} \\
&\quad \mid \textit{term} \\
\textit{term} &\rightarrow \textbf{id} \\
&\quad \mid \textbf{num}
\end{aligned}
$$

where the terminals **if**, **then**, **else**, **relop**, **id**, and **num** generate sets of strings given by the following regular definitions:

$$
\begin{aligned}
\textbf{if} &\rightarrow \texttt{if} \\
\textbf{then} &\rightarrow \texttt{then} \\
\textbf{else} &\rightarrow \texttt{else} \\
\textbf{relop} &\rightarrow \texttt{<} \mid \texttt{<=} \mid \texttt{=} \mid \texttt{<>} \mid \texttt{>} \mid \texttt{>=} \\
\textbf{id} &\rightarrow \textbf{letter}\ (\ \textbf{letter} \mid \textbf{digit}\ )^{*} \\
\textbf{num} &\rightarrow \textbf{digit}^{+}\ (\ \texttt{.}\ \textbf{digit}^{+}\ )?\ (\ \texttt{E(}\ \texttt{+} \mid \texttt{-}\ )?\ \textbf{digit}^{+}\ )?
\end{aligned}
$$

where **letter** and **digit** are as defined previously.

For this language fragment the lexical analyzer will recognize the keywords `if`, `then`, `else`, as well as the lexemes denoted by **relop**, **id**, and **num**. To simplify matters, we assume keywords are reserved; that is, they cannot be used as identifiers. As in Example 3.5, **num** represents the unsigned integer and real numbers of Pascal.

In addition, we assume lexemes are separated by white space, consisting of nonnull sequences of blanks, tabs, and newlines. Our lexical analyzer will strip out white space. It will do so by comparing a string against the regular definition **ws**, below.

$$
\begin{aligned}
\textbf{delim} &\rightarrow \textbf{blank} \mid \textbf{tab} \mid \textbf{newline} \\
\textbf{ws} &\rightarrow \textbf{delim}^{+}
\end{aligned}
$$

If a match for **ws** is found, the lexical analyzer does not return a token to the parser. Rather, it proceeds to find a token following the white space and returns that to the parser.

REGULAR EXPRESSION	TOKEN	ATTRIBUTE-VALUE
ws	-	-
`if`	**if**	-
`then`	**then**	-
`else`	**else**	-
id	**id**	pointer to table entry
num	**num**	pointer to table entry
`<`	**relop**	`LT`
`<=`	**relop**	`LE`
`=`	**relop**	`EQ`
`< >`	**relop**	`NE`
`>`	**relop**	`GT`
`>=`	**relop**	`GE`

Fig. 3.10. Regular-expression patterns for tokens.

Our goal is to construct a lexical analyzer that will isolate the lexeme for the next token in the input buffer and produce as output a pair consisting of the appropriate token and attribute-value, using the translation table given in Fig. 3.10. The attribute-values for the relational operators are given by the symbolic constants `LT`, `LE`, `EQ`, `NE`, `GT`, `GE`. □

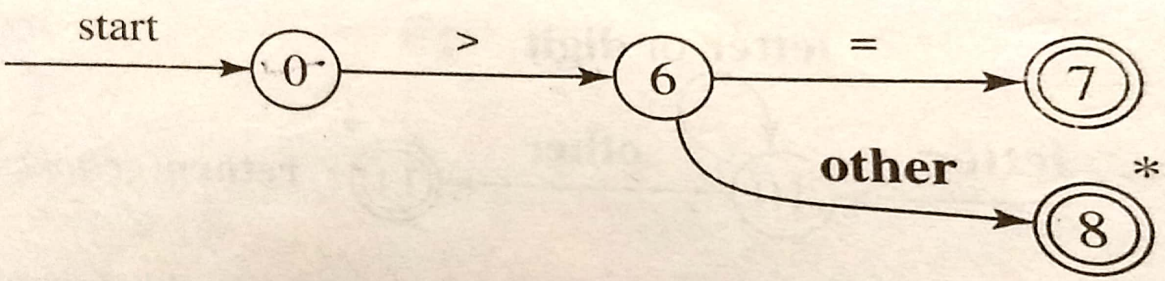


Fig. 3.11. Transition diagram for >=.

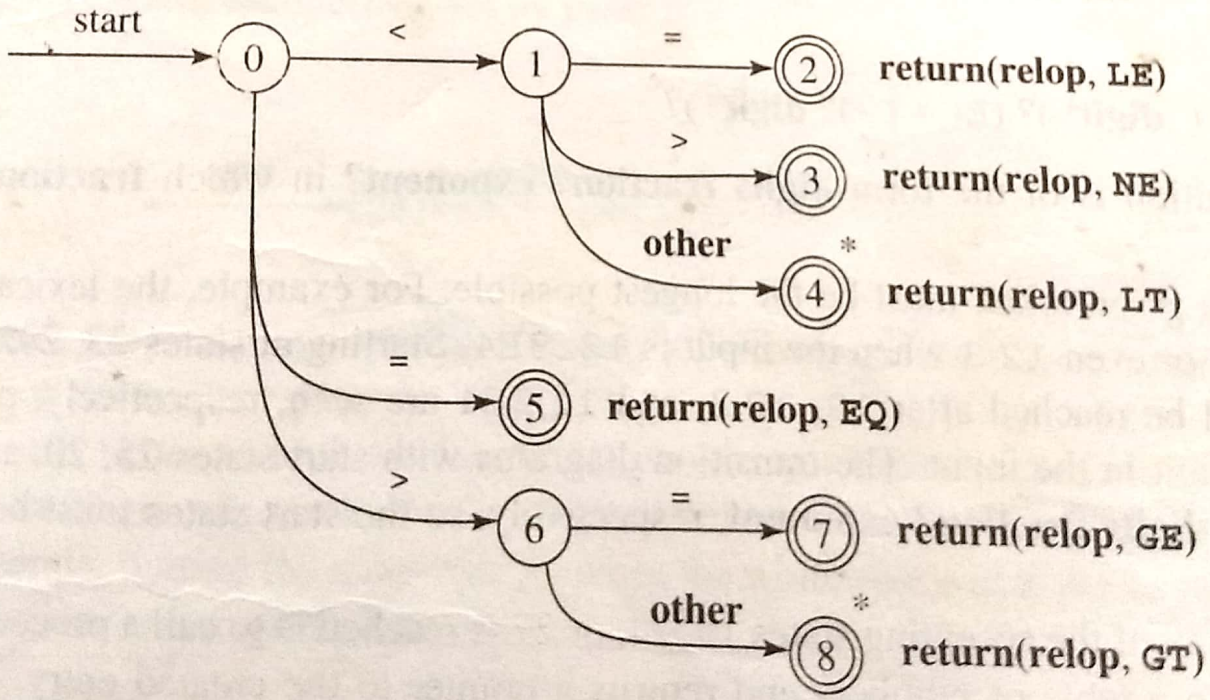


Fig. 3.12. Transition diagram for relational operators.

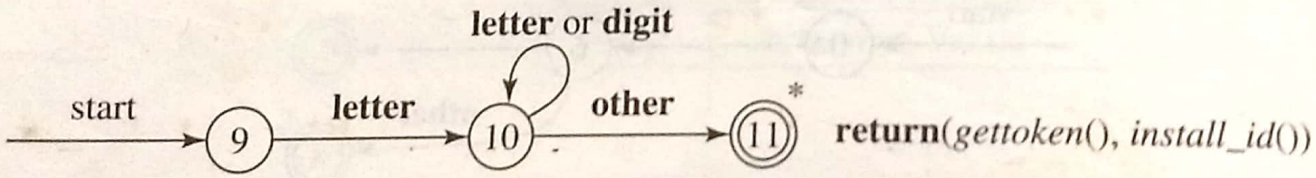


Fig. 3.13. Transition diagram for identifiers and keywords.

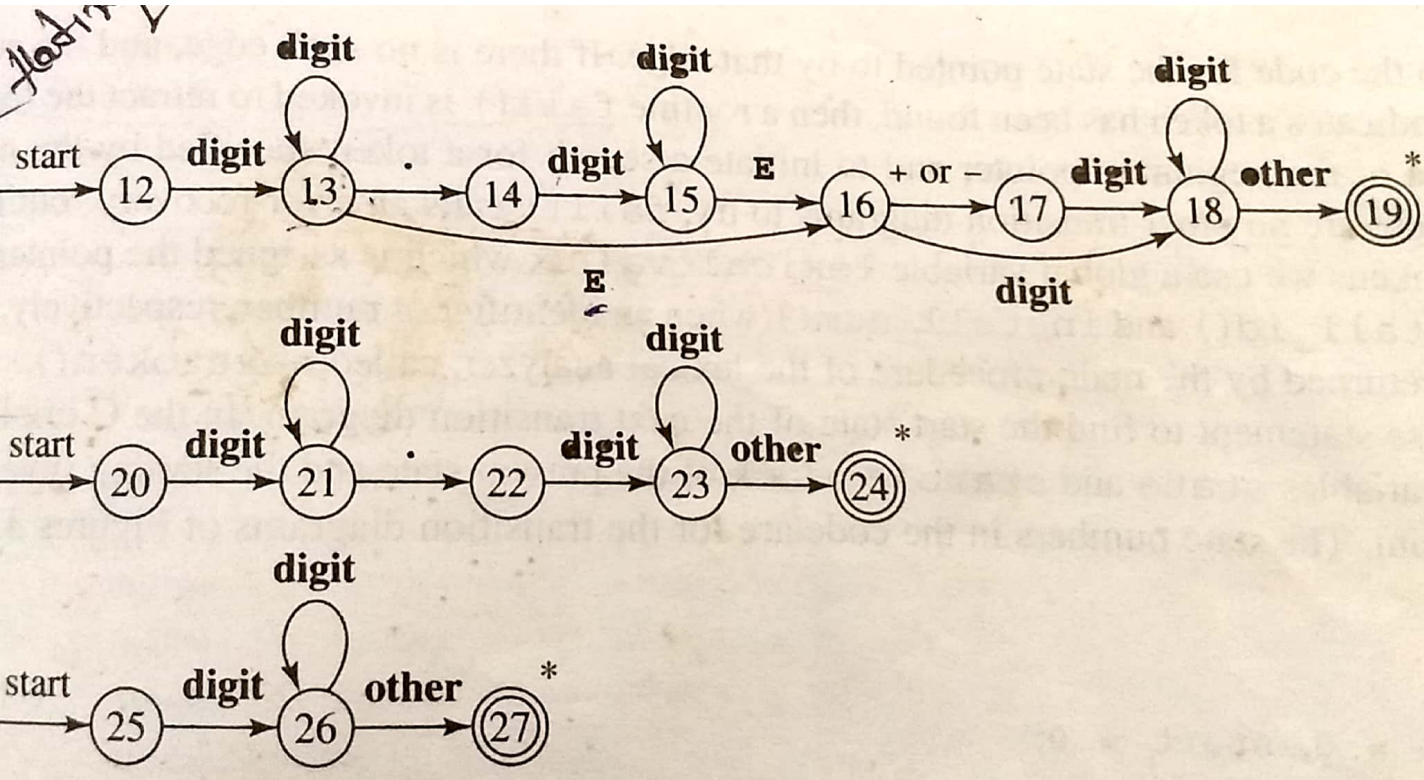
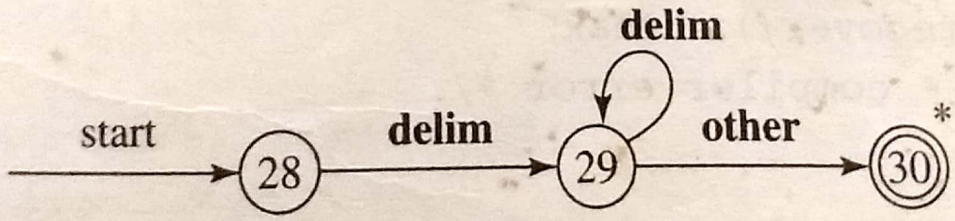


Fig. 3.14. Transition diagrams for unsigned numbers in Pascal.



```
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;
}
```

Fig. 3.15. C code to find next start state.

```
token nexttoken()
{   while(1) {
      switch (state) {
      case 0:   c = nextchar();
         /* c is lookahead character */
         if (c==blank || c==tab || c==newline) {
            state = 0;
            lexeme_beginning++;
               /* advance beginning of lexeme */
         }
         else if (c == '<') state = 1;
         else if (c == '=') state = 5;
         else if (c == '>') state = 6;
         else state = fail();
         break;
         ... /* cases 1-8 here */
      case 9:   c = nextchar();
         if (isletter(c)) state = 10;
         else state = fail();
         break;
      case 10:  c = nextchar();
         if (isletter(c)) state = 10;
         else if (isdigit(c)) state = 10;
         else state = 11;
         break;
      case 11: retract(1); install_id();
         return ( gettoken() );
         ... /* cases 12-24 here */
      case 25:  c = nextchar();
         if (isdigit(c)) state = 26;
         else state = fail();
         break;
      case 26:  c = nextchar();
         if (isdigit(c)) state = 26;
         else state = 27;
         break;
      case 27: retract(1); install_num();
         return ( NUM );
      }
   }
}
```

Fig. 3.16. C code for lexical analyzer.

3.5 A LANGUAGE FOR SPECIFYING LEXICAL ANALYZERS

Several tools have been built for constructing lexical analyzers from special-purpose notations based on regular expressions. We have already seen the use of regular expressions for specifying token patterns. Before we consider algorithms for compiling regular expressions into pattern-matching programs, we give an example of a tool that might use such an algorithm.

In this section, we describe a particular tool, called Lex, that has been widely used to specify lexical analyzers for a variety of languages. We refer to the tool as the *Lex compiler*, and to its input specification as the *Lex language*. Discussion of an existing tool will allow us to show how the specification of patterns using regular expressions can be combined with actions, e.g., making entries into a symbol table, that a lexical analyzer may be required to perform. Lex-like specifications can be used even if a Lex compiler is not available; the specifications can be manually transcribed into a working program using the transition diagram techniques of the previous section.

Lex is generally used in the manner depicted in Fig. 3.17. First, a specification of a lexical analyzer is prepared by creating a program `lex.l` in the Lex language. Then, `lex.l` is run through the Lex compiler to produce a C program `lex.yy.c`. The program `lex.yy.c` consists of a tabular representation of a transition diagram

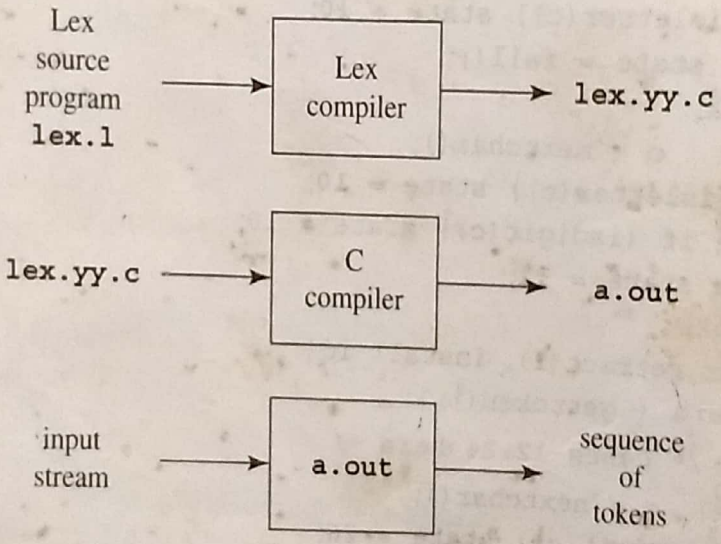


Fig. 3.17. Creating a lexical analyzer with Lex.

constructed from the regular expressions of `lex.l`, together with a standard routine that uses the table to recognize lexemes. The actions associated with regular expressions in `lex.l` are pieces of C code and are carried over directly to `lex.yy.c`. Finally, `lex.yy.c` is run through the C compiler to produce an object program `a.out`, which is the lexical analyzer that transforms an input stream into a sequence of tokens.

Lex Specifications

A Lex program consists of three parts:

declarations
%%
translation rules
%%
auxiliary procedures

The declarations section includes declarations of variables, manifest constants, and regular definitions. (A manifest constant is an identifier that is declared to represent a constant.) The regular definitions are statements similar to those given in Section 3.3 and are used as components of the regular expressions appearing in the translation rules.

The translation rules of a Lex program are statements of the form

$$\begin{array}{ll} p_1 & \{ action_1 \} \\ p_2 & \{ action_2 \} \\ \cdots & \cdots \\ p_n & \{ action_n \} \end{array}$$

where each p_i is a regular expression and each $action_i$ is a program fragment describing what action the lexical analyzer should take when pattern p_i matches a lexeme. In Lex, the actions are written in C; in general, however, they can be in any implementation language.

The third section holds whatever auxiliary procedures are needed by the actions. Alternatively, these procedures can be compiled separately and loaded with the lexical analyzer.

A lexical analyzer created by Lex behaves in concert with a parser in the following manner. When activated by the parser, the lexical analyzer begins reading its remaining input, one character at a time, until it has found the longest prefix of the input that is matched by one of the regular expressions p_i. Then, it executes $action_i$. Typically, $action_i$ will return control to the parser. However, if it does not, then the lexical analyzer proceeds to find more lexemes, until an action causes control to return to the parser. The repeated search for lexemes until an explicit return allows the lexical analyzer to process white space and comments conveniently.

The lexical analyzer returns a single quantity, the token, to the parser. To pass an attribute value with information about the lexeme, we can set a global variable called `yylval`.

Example 3.11. Figure 3.18 is a Lex program that recognizes the tokens of Fig. 3.10 and returns the token found. A few observations about the code will introduce us to many of the important features of Lex.

In the declarations section, we see (a place for) the declaration of certain manifest constants used by the translation rules.[4] These declarations are surrounded by the special brackets %{ and %}. Anything appearing between these brackets is copied directly into the lexical analyzer `lex.yy.c`, and is not treated as part of the regular definitions or the translation rules. Exactly the same treatment is accorded the auxiliary procedures in the third section. In Fig. 3.18, there are two procedures, `install_id` and `install_num`, that are used by the translation rules; these procedures will be copied into `lex.yy.c` verbatim.

Also included in the definitions section are some regular definitions. Each such definition consists of a name and a regular expression denoted by that name. For example, the first name defined is `delim`; it stands for the character class `[\t\n]`, that is, any of the three symbols blank, tab (represented by `\t`), or newline (represented by `\n`). The second definition is of white space, denoted by the name `ws`. White space is any sequence of one or more delimiter characters. Notice that the word `delim` must be surrounded by braces in Lex, to distinguish it from the pattern consisting of the five letters `delim`.

In the definition of `letter`, we see the use of a character class. The shorthand `[A-Za-z]` means any of the capital letters `A` through `Z` or the lowercase letters `a` through `z`. The fifth definition, of `id`, uses parentheses, which are metasymbols in Lex, with their natural meaning as groupers. Similarly, the vertical bar is a Lex metasymbol representing union.

```
%{
        /* 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 = install_id(); return(ID);}
{number}    {yylval = install_num(); 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);}

%%

install_id() {
        /* procedure 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 */
}

install_num() {
        /* similar procedure to install a lexeme that
        is a number */
}
```

Fig. 3.18. Lex program for the tokens of Fig. 3.10.