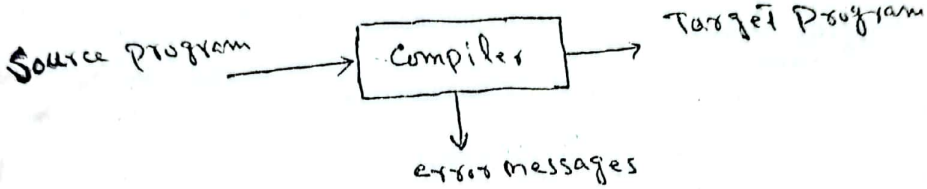
COMPILER THEORY

Compiler:- A compiler is a program that reads a program written in one language (i.e source program) and translates it into an equivalent program in another language (i.e the target language). Besides this, the compiler reports to its user the presence of errors in the source program.



There are thousands of source ~~program~~ languages, ranging from traditional programming languages such as fortran and Pascal to specialized languages that can be used in every area of computer application.

The target languages are equally as varied; a target language may be another programming language, or the machine language of any computer.

Compilers are sometimes classified as single-pass, multipass, debugging or optimizing, depending on how they have been constructed or on what function they are supposed to perform. Despite this apparent complexity, the basic functions that any compiler must perform are essentially the same.

The Analysis - Synthesis model of Compilation

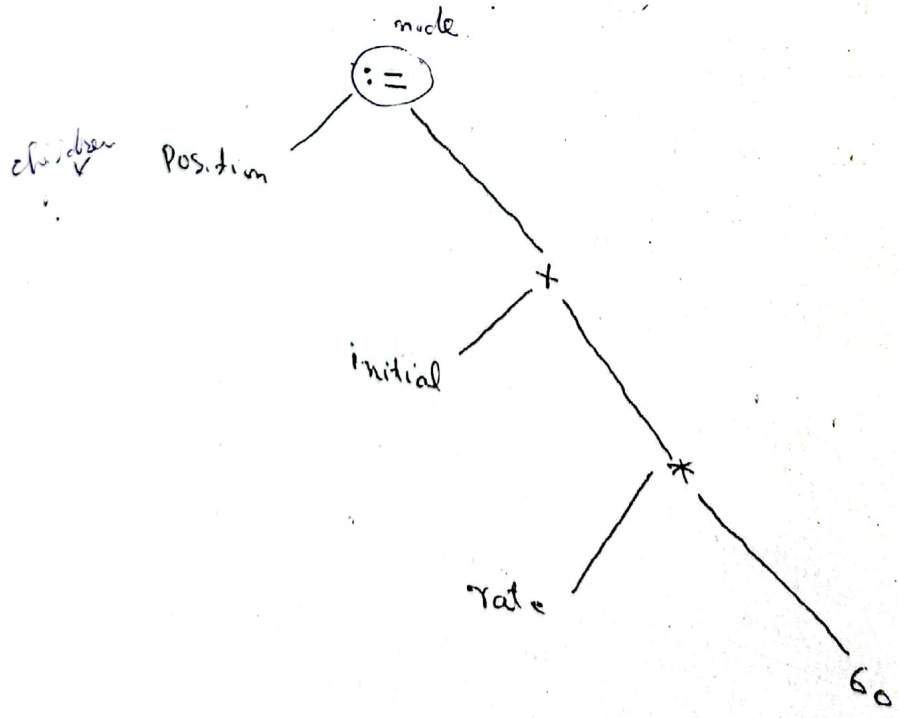
There are two parts of Compilation: analysis and Synthesis. The analysis part breaks up the source program into pieces and creates an intermediate representation of the source program.

The synthesis part constructs the desired target program from the intermediate representation.

During analysis, the operations implied by the source program are determined and recorded in a hierarchical structure called a tree. Often a special kind of tree called a syntax tree is used, in which each node represents an operation and the children of a node represent the arguments of the operation.

For example, a syntax tree for an assignment statement is as follows:

Position := initial + rate * 60



many software tools that manipulate source programs. first perform some kind of analysis.
Some examples of such tools include:

1- Structure Editors :- A structure editor takes as input a sequence of commands to build a source program. The structure editor not only performs the text-creation and modification functions of an ordinary text editor, but it also analyzes the program text, putting an appropriate hierarchical structure on the source program. Thus, the structure editor can perform additional tasks that are useful in the preparation of programs.
For example, it can check that the input is correctly formed, can supply keywords automatically (e.g when the user types while, the editor supplies the matching do and reminds the user that a Conditional must come between them), and can jump from a begin or left parenthesis to its matching end or right parenthesis.
Further, the output of such an editor is often similar to the output of the analysis phase of a Compiler.

2- Pretty printers :- A pretty printer analyzes a program and prints it in such a way that the structure of the program becomes clearly visible. For example, comments may appear in a special font, and statements may appear with an amount of identation proportional to the depth of their nesting in the hierarchical organization of the statements.

3- Static checkers :- A static checker reads a prog
analyzes it, and attempts to discover bugs wit
running the program. For example, a static chec
may detect the parts of the source program ca
never be executed, or that a certain variab
might be used before being defined.

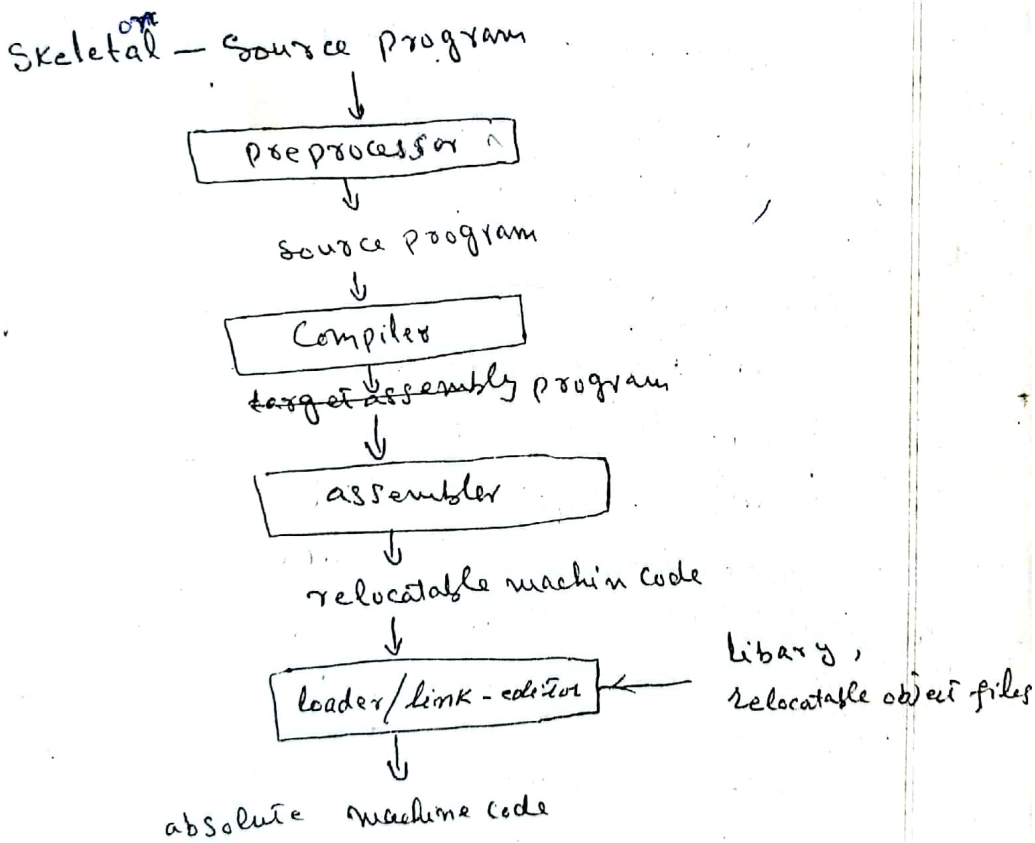
4. Interpreters :- Instead of producing a targe
program as a translation, an interpreter
performs the operations implied by the source prog
For an assignment statement, for example, an
interpreter might build a tree and then carry out the
operations at the nodes as it walks the tree.
consider the tree on page 2.
At the root it would discover it (interpreter) had
an assignment to perform, so it would call a routin
to evaluate the expression on the right, and then
store the resulting value in the location associated w
the identifier position. At the right side of the root
the routine would discover it had to compute the sum
two expressions. It would call itself recursive
to compute the value of the expression rate * 60
It would then add that value to the value of the
variable initial.

Interpreters are frequently used to
execute command languages. Since each operator
executed in a command language is usually an
invocation of a complex routine such as a
compiler.

The Context of a Compiler:

In addition to a compiler, several other programs may be required to create an executable target program. A source program may be divided in modules stored in separate files. The task of collecting the source program is sometimes given to a distinct program, called a preprocessor. The preprocessor may also expand shorthands called macros, into source language statements.

The following fig shows a typical "Compilation".



The target program created by the compiler may require further processing before it can be run. The compiler creates assembly code that is translated by an assembler into machine code and then linked together with some library routines into the code that actually runs on the machine.

Analysis of the Source program:

On Compiling, analysis consists of three phas

1- linear analysis, in which the stream of characters mak the source program is read from left to right and grou into tokens that are sequences of characters have a meaning. Coll

2- Hierarchical analysis, in which characters or toke are grouped hierarchically into nested collections wi collective meaning.

3- Semantic analysis, in which certain checks are performed to ensure that the components of a progr fit together meaningfully.

Lexical Analysis:

On a Compiler, linear analysis is called lexical analysis or scanning. For example, in lexic analysis the characters in the assignment statement

position := initial + rate * 60

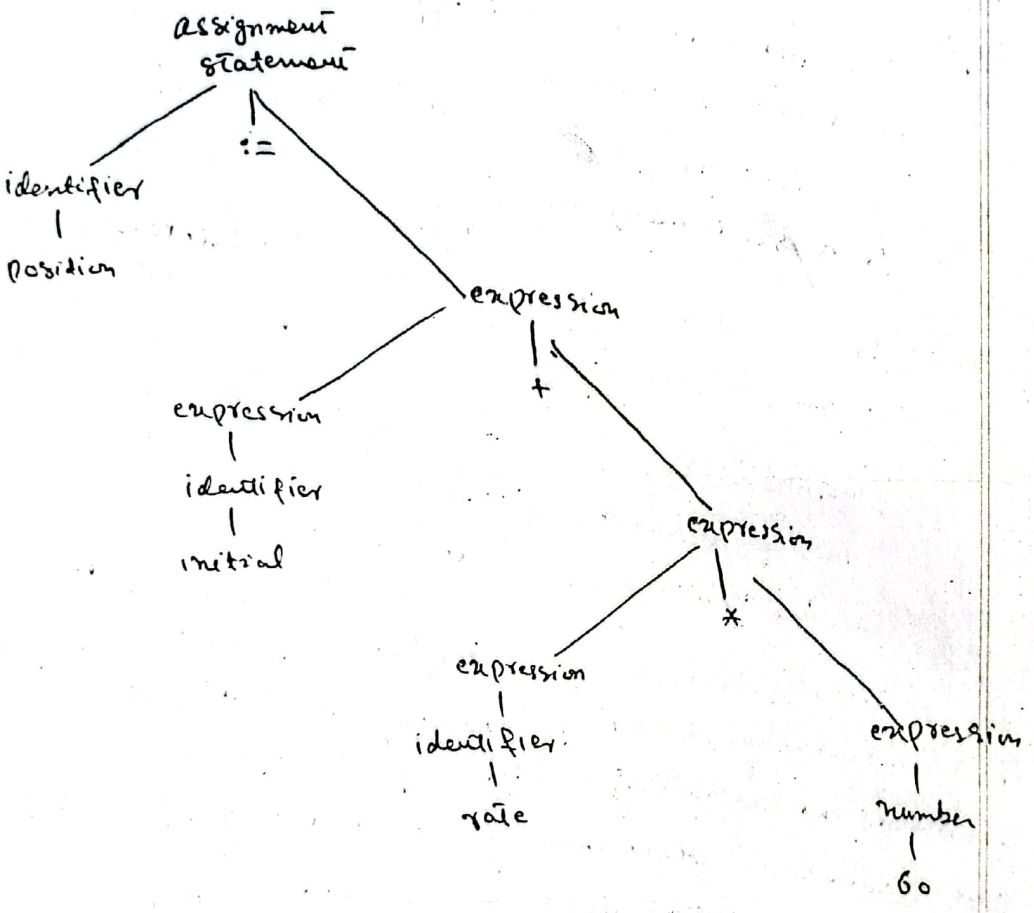
would be grouped into the following tokens:

- The identifier position
- The assignment symbol :=
- The identifier initial
- The plus sign
- The identifier rate
- The multiplication sign
- The number 60

The blanks separating the characters of these tokens would normally be eliminated during lexical analys

Syntax Analysis.

Hierarchical analysis is called parsing or syntax analysis. It involves grouping the tokens of the source program into grammatical phrases that are used by the compiler to synthesize the output. Usually, the grammatical phrase of the source program are represented by a parse tree such as in following fig; the parse tree of position := initial + rate * 60 is,



The hierarchical structure of a program is usually expressed by recursive rules. For example, we might have the following rules as part of the definition of expressions:

1 - Any identifier is an expression
2 - Any number is an expression
3 - If expression$_1$ and expression$_2$ are expressions, then so are
    expression$_1$ + expression$_2$
    expression$_1$ * expression$_2$
    (expression$_1$)

Rules (1) and (2) are (non-recursive) basis [illegible]
defines expressions in terms of operators applied to other exp
then by rule (1), initial and rate are expressions. By
60 is an expression, while by rule (3), we can first infer
rate * 60 is an expression and finally that initial + rate * 6
is an expression.

Similarly, many languages define statements recursively by rules such as:

1- If identifier, is an identifier, and expression$_2$ is an expression, then

identifier$_1$:= expression$_2$

is a statement

2- If expression$_1$ is an expression and statement$_2$ is a statement, then:

while (expression$_1$) do statement$_2$

if (expression$_1$) then statement$_2$

are statements.

The division between lexical and syntactic analysis is somewhat arbitrary - we usually ~~do~~ choose a division that simplifies the overall task of analysis. One factor in determining the division is whether source language construct is inherently recursive or not. Lexical constructs do not require recursion while syntactic constructs often do.

1-1-1994

A Syntax tree is a compressed representation of the parse tree in which the operators appear as interior nodes, and the operands of an operator are children of the node for that operator.

e.g.

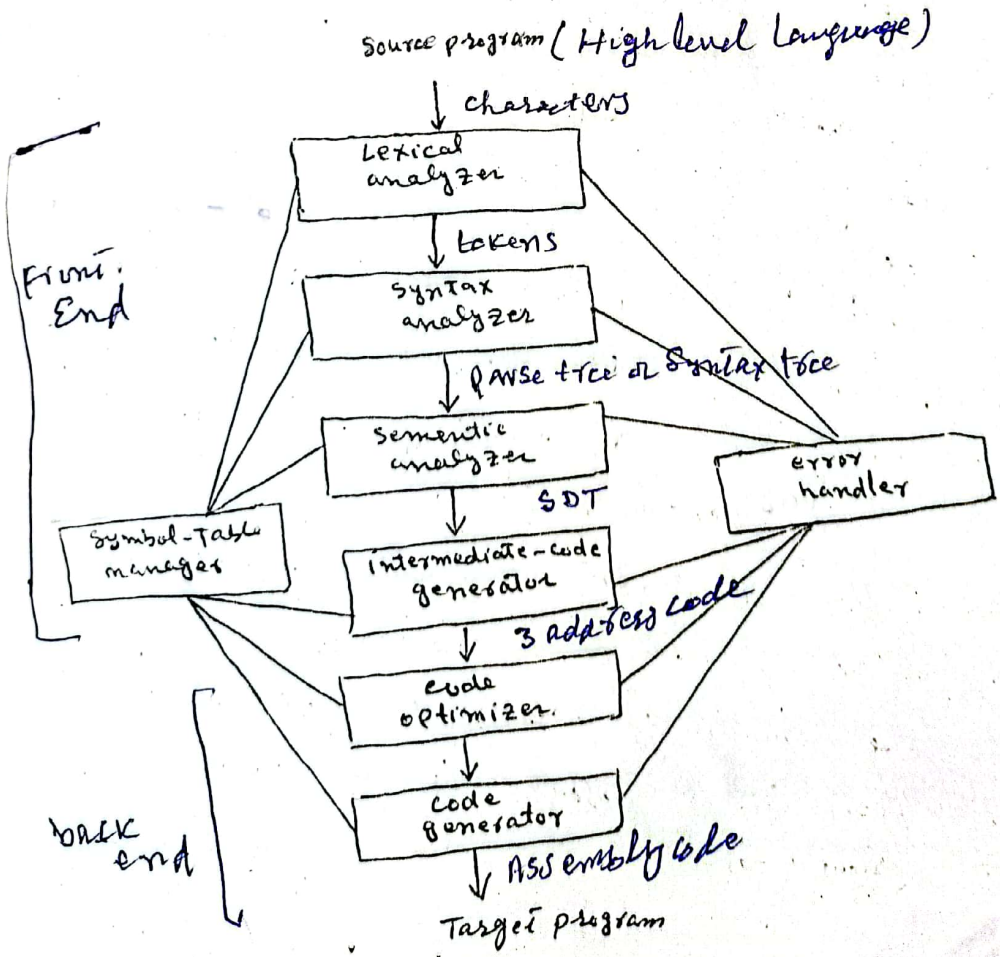
Semantic Analysis: (It find logical Errors)

The semantic analysis phase checks the source program for semantic errors and gathers type information for the subsequent code-generation phase. It uses the hierarchical structure determined by the syntax analysis phase to identify the operators and operands of expressions and statements.

An important component of semantic analysis is type checking. Here the compiler checks that each operator has operands that are permitted by the source language specification. For example, many programming language definitions require a compiler to report an error every time a real number is used to index an array.

The phases of a Compiler :

A compiler operates in phases, each of which transforms the source program from one representation to another. A typical decomposition of a compiler is:



The first three phases, forming the bulk of the analysis part
Two other activities, symbol-table management and error han
are shown interacting with the six phases of lexical analy
syntax analysis, Semantic analysis, intermediate-code gen
code optimization, and code generation.

Symbol-Table management :-

An essential function of a compiler is
record the identifiers used in the source program and co
information about the various attributes of each identifier.
These attributes may provide information about the storage al
for an identifier, its type, its scope (where in the pro
it is valid), and in the case of procedure names,
such things as the number and types of its arguments

A symbol table is a data structure containing
record for each identifier, with fields for the attributes
the identifier. The data structure allows us to find the
record for each identifier quickly and to store or retrieve
data from that record quickly.

When an identifier in the source program is detected
the lexical analyzer, the identifier is entered into the
symbol table. However, the attributes of an identifier
cannot normally be determined during lexical analy
For example, in a pascal declaration like

var position, initial, rate : real ;

The type real is not known when position, initial, an
rate are seen by the lexical analyzer.

The remaining phases enter information about
identifiers into the symbol table and then use this infor
in various ways.

Error Detection and Reporting:

Each phase can encounter errors. However, 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.

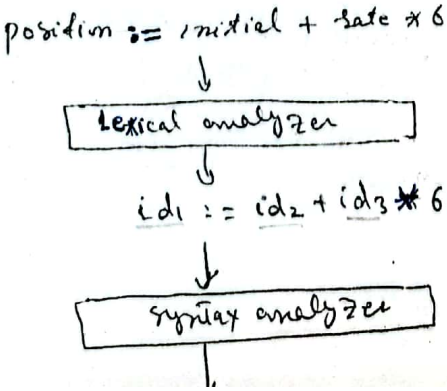
The syntax and semantic analysis phases usually handle a large fraction of the errors detectable by the compiler. The lexical phase can detect errors where the characters remaining in the input do not form any token of the language. Errors where the token stream violates the structure rules (syntax) of the language are determined by the syntax analysis phase. During semantic analysis the compiler tries to detect constructs that have the right syntactic structure but no meaning to the operation involved, e.g: if we try to add two ~~structure~~ identifiers, one of which is the name of an array, and the other the name of a procedure.

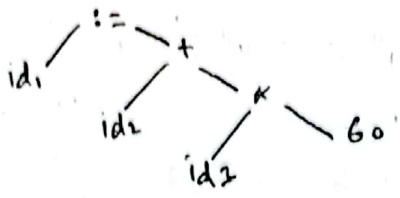
The Analysis phases:

As translation progresses, the compiler's internal representation of the source program changes. We show these representations by considering the translation of the statement.

position := initial + rate * 60 —— (1)

The following fig shows the representation of this statement after each phase.

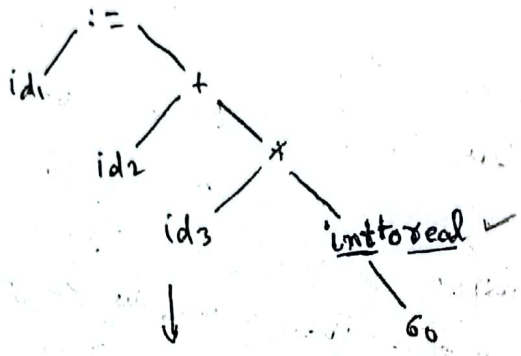




↓

semantic analyzer

↓



Symbol table

1	position	...
2	initial	...
3	rate	...
4		

↓

intermediate code generator

↓

```
temp1 := inttoreal (60)
temp2 := id3 * temp1
temp3 := id2 + temp2
id1   := temp3
```

↓

code optimizer

↓

```
temp1 := id3 * 60.0
id1   := id2 + temp1
```

↓

code generator

↓

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

left to right to

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

m & tariq.
m & tariq.

The lexical analysis phase reads the characters in the source program and groups them into a stream of tokens in which each token represents a logically sequence of characters, such as an identifier, a keyword (if, while etc), a punctuation character, or a multi-character operator like :=. The character sequence forming a token is called the lexeme for the token

Certain tokens will be augmented by a "lexical value". For example, when an identifier like rate is found, the lexical analyzer not only generates a token, say id, but also enters the lexeme rate into the symbol table. The lexical value associated with occurrence of id points to the symbol-table entry for rate.

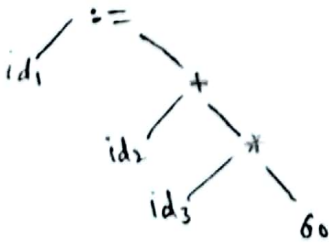
We shall use id_1, id_2 and id_3 for position, initial, and rate respectively, to emphasize that the internal representation of an identifier is different from the character sequence forming the identifier. The representation of (1) after lexical analysis is

Therefore is

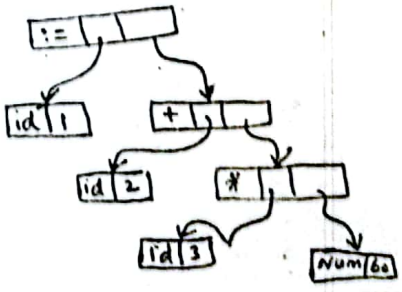
$$id1 := id_2 + id_3 * 60 \qquad \text{---- (2)}$$

Syntax analysis imposes a hierarchical structure on the token stream, which we shall show by syntax trees in fig (a).

A typical data structure for the tree is shown in fig (b) in which an interior node is a record with a field for the operator and two fields containing pointers to the records for the left and right children. A leaf is a record with two or more fields, one to identify the token at the leaf, and the others to record information about the token.



(a)



(b)

The data structure in (b) for the syntax tree in (a)

Intermediate Code Generation:

After syntax and semantic analysis, some compilers g an explicit intermediate representation of the source program. This intermediate representation should have two importan properties; it should be easy to produce, and easy to translate into the target program.

We consider an intermediate form called "three-address c which is like the assembly language for a machine in which ev memory location can act like a register. Three-address code consists of a sequence of instructions, each of which has at mo three operands. The source program in (1) in three-address code is:

```
temp1 := intToReal(60)
temp2 := id3 * temp1
temp3 := id2 + temp2
id1   := temp3
```

——— (3)

This intermediate form has several properties:-

(i) Each three-address instruction has at most on operator in addition to the assignment. Thus when generating these instructions, the compiler has to decide the order in which operations are to be done; the multiplication precedes the addition in source program (1)

(ii) The compiler must generate a temporary name to hold the value computed by each instruction.

(iii) Some "three address" instructions have fewer than three operands, e.g. the first and las instructions in (3)

id2, temp2 etc.

Code optimization:

The code optimization phase attempts to improve the intermediate code, so that faster-running machine code will result. Some optimizations are trivial. For example, a natural algorithm generates the intermediate code (3), using an instruction for each operator in the tree representation after semantic analysis, even though there is a better way to perform the same calculation, using the two instructions

```
temp1 := id3 * 60.0
id1   := id2 + temp1
```

} ——— (4)

Code Generation:

The final phase of the compiler is the generation of target code, consisting normally of relocatable m/c code or assembly code. Memory locations are selected for each of the variables used by the program. A crucial aspect is the assignment of variables to registers.

For example, using registers 1 and 2, the translation of the code in (4) will become

8086

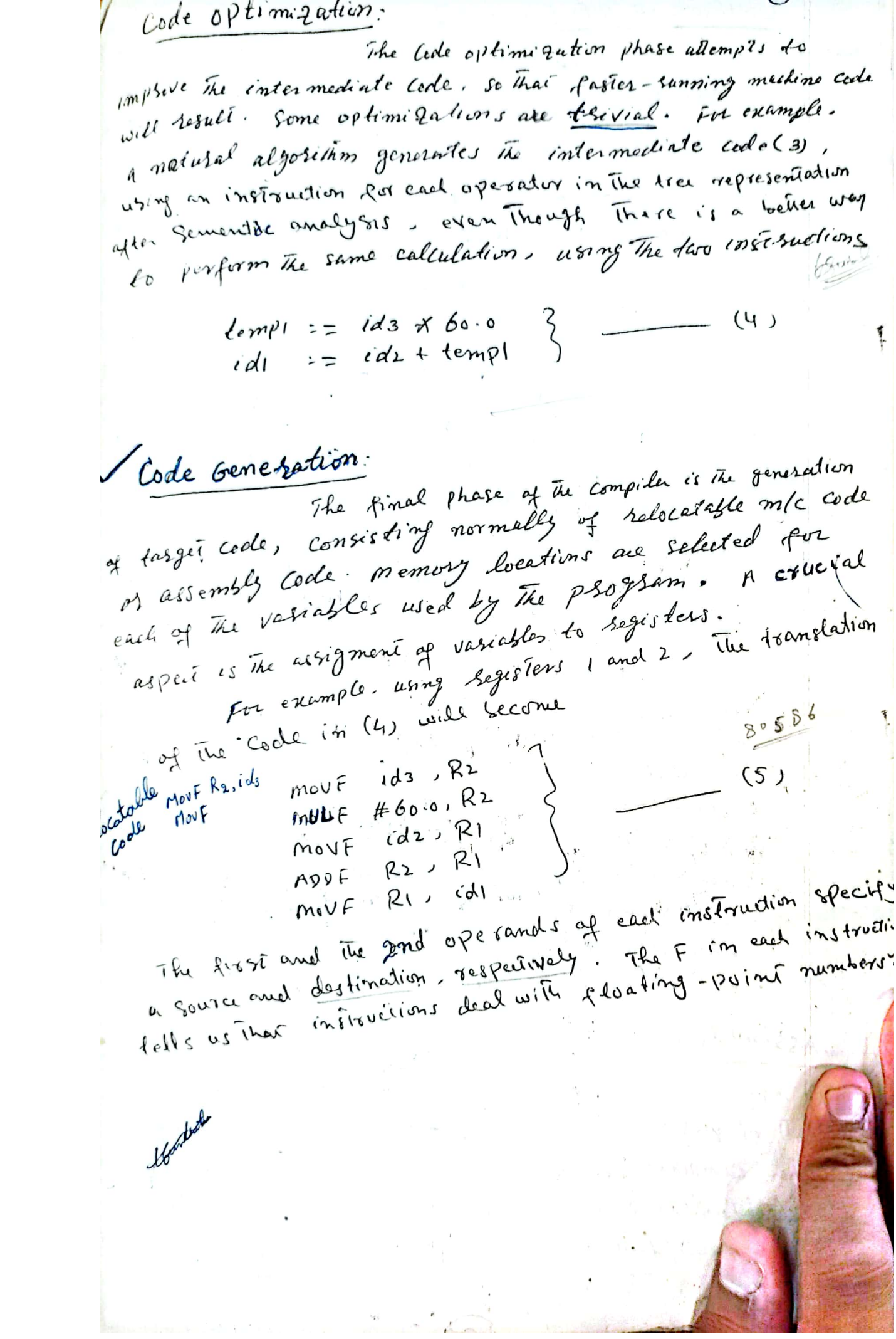
relocatable code

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

} ——— (5)

MOVF R2, id3
MOVF

The first and the 2nd operands of each instruction specify a source and destination, respectively. The F in each instruction tells us that instructions deal with floating-point numbers.



COUSINS OF THE COMPILER :-

Preprocessors :- preprocessors produce input to compilers. They may perform the following functions :-

1- **macro processing**: A preprocessor may allow a user to define macros that are shorthands for longer constructs.

2- **File inclusion**: A preprocessor may include header files into the program text.

3- **Rational preprocessors** :- These processors augment older languages with more modern flow-of-control and data-structuring facilities. For example, such a preprocessor might provide the user with built-in macros for constructs like while-statements or if-statements, where none exist in the programming language itself.

4- **language extensions**: These processors attempt to add capabilities to the language by what amounts to built in macros. For example, the language Equel is a database query language embedded in C. statements beginning with # are taken by the preprocessor to be database-access statements, unrelated to C, and are translated into procedure calls on routines that perform the database access.

Assemblers :-

Some compilers produce assembly code as in (a) that is passed to an assembler for further processing. Other compilers perform the job of the assembler, producing relocatable m/c code that can be passed directly to the loader / link-editor.

Assembly Code is a mnemonic version of m/c code, in which names are used instead of binary codes for operations and names are also given to memory addresses. A typical sequence of assembly instructions might be

```
MOV a, R1
ADD #2, R1
MOV R1, b
```

(6)

This code moves the contents of the address a into register 1. Then adds the constant 2 to it and finally stores the result in the location named b. Thus, it computes

b := a + 2.

TWO-PASS Assembler

The simplest form of assembler makes two passes over the input, where a pass consists of reading an input file once. In the 1st pass, all the identifiers that denote storage locations are found and stored in a symbol table. Identifiers are assigned storage locations as they are encountered for the first time, so after reading (6), for example, the symbol table might contain the entries as follows

IDENTIFIER	ADDRESS
a	0
b	4

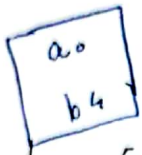
Here, we have assumed that a word, consisting of four bytes, is set for each identifier, and that addresses are assigned starting from byte 0.

In the second pass, the assembler scans the input again. This time, it translates each operation code into the sequence of bits representing that operation in m/c language, and it translates each identifier representing a location into the address given for that identifier in the symbol table.

The output of the 2nd pass is usually relocatable m/c code, meaning that it can be loaded starting at

any location L in memory i.e., if L is
to all addresses in the code, then all references will
correct. Thus, the output of the assembler must dif
Those positions of instructions that refer to addres
that can be relocated.

Example:



Loaders and Link-Editors:-

Usually a program called a loader performs
two functions of loading and link editing. The process of loa
consists of taking relocatable m/c code, altering the
relocatable addresses and placing the altered instructi
and data in memory at the proper locations.

The link-editor allows us to make a single p
from several files of relocatable m/c code. These f
may have been the result of several different compilati
and one or more may be library files of routines
provided by the system and available to any progra
that needs them.

THE GROUPING OF PHASES:-

In an implementation, activities from more than
one phase are often grouped together.

Front and Back Ends:-

Often, the phases are collected into a front end
and a back end. The front end consists of those phases,
depend primarily on the source program (or source language)
and largely independent of the target m/c. These normall
include lexical and syntactic analysis, the creation of
symbolic table, semantic analysis, and the generation
of intermediate code.

A certain amount of code optimization can be done by front end as well. The front end also includes the error handling that goes along with each of these phases.

The back end includes those portions of compiler that depend on the target m/c, and generally those portions do not depend on the source language. In the back end, we find aspects of the code optimization phase, and we find code generation, along with the necessary error handling and symbol-table operations.

PASSES :- Several phases of compilation are usually implemented in a single pass consisting of reading an input file and writing an output file.

It is common for several phases to be grouped into one pass, and for the activity of these phases to be interleaved during the pass. For example, lexical analysis, syntax analysis, semantic analysis and intermediate code generation might be grouped into one pass. If so, the token stream after lexical analysis may be translated directly into intermediate code.

In an implementation of a compiler, portions of one or more phases are combined into a module called a pass. A pass reads the source program or output of the previous pass, makes the transformations specified by its phases, and writes output into an intermediate file, which may then be read by a subsequent pass. If several phases are grouped into one pass, then the operation of the phases may be interleaved, with control alternating among several phases.

A multi-pass compiler can be made can be made use less space than a single-pass compiler, since the occupied by the compiler programs for one pass can be by the following pass. A multipass compiler is slower than a single-pass compiler, because each reads and writes an intermediate file. Thus, running on computers with small memory would require use several passes while, on a computer with a random memory, a compiler with fewer passes would be possible.

Compiler-Construction Tools:

A number of tools have been developed specifically to help construct compilers. These range from scanner and parser generators to complete systems, variously called compile-compilers, compile-generators or translator-writing systems which produce a compiler from some form of specification of a source language and target. The input specification for these systems may contain

- a description of the lexical and syntactic structure of the source language.
- a description of what output is to be generated for each source language construct.
- a description of the target m/c (language)

The following is the list of some useful compiler-construction tools or

The principal aids provided by existing compilers are:

1- Parser generator:- These produce syntax analyzers, normally from input that is based on a context-free grammar.

2- Scanner generator:- These automatically generate lexical analyzers, normally from a specification based on regular expressions.

3- Syntax-directed translation engines:

These produce collections of routines that walk the parse tree, generating intermediate code.

4- Automatic code generators:

such a tool takes a collection of rules that define the translation of each operation of the intermediate language into the m/c language for the target machine. The rules must include sufficient detail that we can handle the different possible access methods for data: e.g., variables may be in registers or a fixed (static) location in memory.

5- Data-flow engines:

Much of the information needed to perform d.f.a good code optimization involves "data-flow analysis", the gathering of information about how values are transmitted from one part of the program to each other part.

A SIMPLE ONE-PASS COMPILER

Overview :- A programming language can be defined by describing what its programs looks like (the syntax of the language) and what its programs mean (the semantics of the language). For specifying the syntax of a language, we use a notation called context-free grammars or BNF (Backus-Naur Form).

A grammar-oriented compiling technique, known as syntax-directed translation, is very helpful for organizing compiler front end (before machine code)

In Compiler, the lexical analyzer converts the stream of input characters into a stream of tokens that becomes the input to the following phase as in fig.

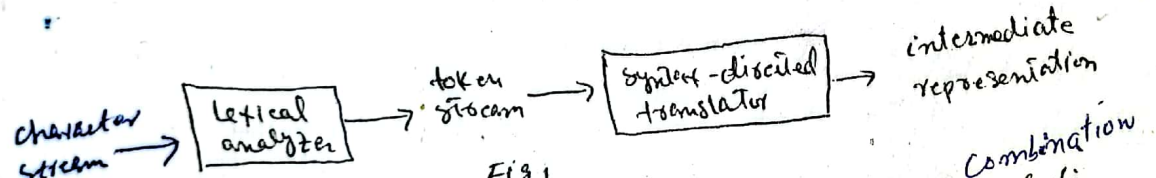


Fig 1

The "syntax-directed translator" in the fig is the ~~Compilation~~ combination of a syntax analyzer and an intermediate code generator.

Syntax Definition:

we introduce a notation, called a context-free grammar (or grammar), for specifying the syntax of a language.

A grammar naturally describes the hierarchical structure of many programming language constructs. For example, an if-else statement in C has the form

if (expression) statement else statement

· That is, the statement is the [illegible] of the keyword
an opening parenthesis, an expression, a closing Parenthesis
a statement, the keyword else, and another statement.
Using the variable expr to denote an expression and the
variable stmt to denote a statement, this conditional
rule can be expressed as

left side ↓ ↓ right side
stmt → if (expr) stmt else stmt

in which the arrow may be read as "can have the form". Such
rule is called a production. In a production lexical
elements like the keyword if and the parenthesis are called
tokens. Variables like expr and stmt represent sequences of
tokens and called nonterminals.

A context-free grammar has four components:

1- A set of tokens, known as terminal symbols.
2- A set of nonterminals
3- A set of productions where each production consists of
a nonterminal, called the left side of the production,
an arrow, and a sequence of tokens and/or nonterminals,
called the right side of the production.
4- A designation of one of the nonterminals as the
start symbol

We follow the convention of specifying grammars by listing
their productions, with the productions for the start symbol
listed first. We assume that digits, signs such as <= ,
and underline strings such as while are called terminals.

Example 1 :- We use expressions consisting of digits and plus and minus signs, e.g. 9-5+2, 3-1 and 7. Since a plus or minus sign must appear between two digits, we refer to such expressions as "lists of digits separated by plus or minus signs". The following grammar describes the syntax of these expressions. The productions are:

list → list + digit ——— (1)
list → list - digit ——— (2)
list → digit ——— (3)
digit → 0|1|2|3|4|5|6|7|8|9 ——— (4)

The right sides of the productions with nonterminal list on the left side can equivalently be grouped:

list → list + digit | list - digit | digit

* The tokens of the grammar are the symbols

+ - 0 1 2 3 4 5 6 7 8 9

The nonterminals are the list and digit, with list being the starting nonterminal because its productions are given first.

We say a production is for a nonterminal if the nonterminal appears on the left side of the production. A string of tokens is a sequence of zero or more tokens. The string containing zero tokens, written as ε, is called the empty string.

A grammar derives strings by beginning with the start symbol and repeatedly replacing a nonterminal by the right side of a production for that nonterminal. The token strings that can be derived from the start symbol form the language defined by the grammar.

Example 2 :- The language defined by the grammar of example 1, consists of lists of digits separated by plus and minus signs.

We can deduce that 9-5+2 is a list as follows:

a) 9 is a list by production (3), since 9 is a digit.
b) 9-5 is a list by production (2), since 9 is a list and 5 is a digit.
c) 9-5+2 is a list by production (1), since 9-5 is a list and 2 is a digit.

list = list + digit
list - list - digit
list - digit

This reasoning is shown by the tree. Each node in the tree is labeled by a grammar symbol. An interior node and its children correspond to a production; the interior node corresponds to the left side of the production, the children to the right side. Such trees are called parse trees.

list = list + digit
terminal node children.

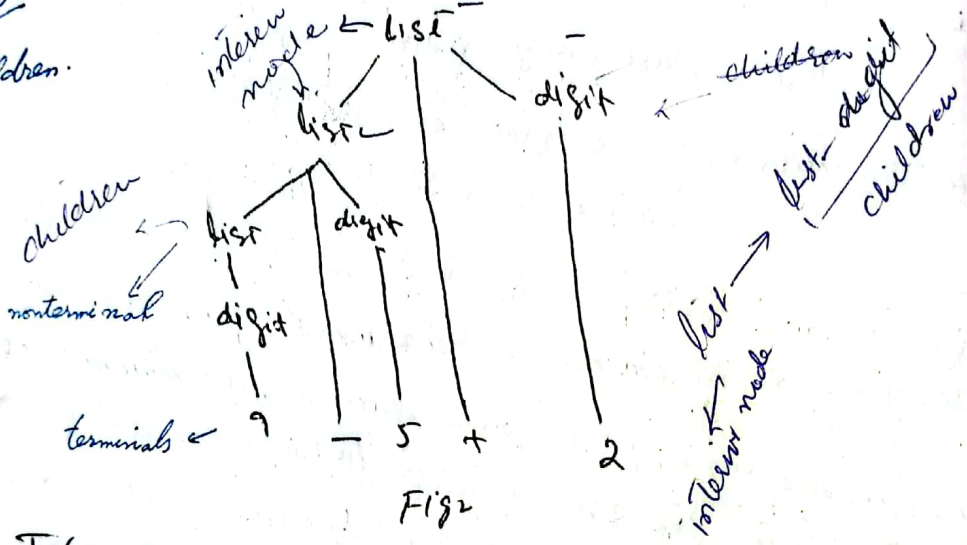


Fig 2

Parse Trees :- A parse tree ~~that~~ shows how the start symbol of a grammar derives a string in the language of nonterminal A has a production $A \to XYZ$. Then a parse tree may have an interior node labeled A with three children labeled X, Y and Z, from left to right.

Formally, given a context free grammar, a parse tree is a tree with the following properties:-

1. The root is labeled by the start symbol
2. Each leaf is labeled by a token or by ε
3. Each interior node is labeled by a nonterminal.
4. If A is the nonterminal labeling some interior node and $X_1, X_2, \ldots, X_n$ are the labels of the children of that node from left to right, then $A \to X_1 X_2 \cdots X_n$ is a production. Here $X_1, X_2 \cdots X_n$ stand for a symbol that is either a terminal or a nonterminal. As a special case, if $A \to \epsilon$ then a node labeled A may have a single child labeled ε.

list & digits are nonterminals, +, - are terminals

Example 3 :- In fig 2, the root is labeled list, the start symbol in example(1). The children of the root are labeled from left to right, list, +, and digit.

The leaves of a parse tree read from left to right form the **yield** of the tree, which is the string generated from the nonterminal at the root of the parse tree. In fig 2, the generated string is 9-5+2. In that fig, all the leaves are shown at the bottom level.

(1 Another definition of the language generated by a grammar is as "the set of strings that can be generated by some parse tree". The process of finding a parse tree for a given string of tokens is called **parsing** that string. //#

Ambiguity :- If a grammar can have more than one parse tree generating a given string of tokens, then such a grammar is said to be **ambiguous**.

To show that a grammar is ambiguous, all we need

to do is find a token string that has more than one parse tree. Since a string with more than one parse tree usually has more than one meaning, ~~a string with more than one parse tree usually~~ for compiling applications we need to design unambig grammars.

EXAMPLE 4:- Suppose we did not distinguish betw digits and lists as in example(1). We could have written the grammar

string → string + string | string - string | 0|1|2|3|4|5|6|7|8|9

However, fig(3) shows that an expression like 9-5+2 now has more than one parse tree. The two trees for 9-5+2 correspond to the two ways of parenthesizing the expression: (9-5)+2 and ~~(9-(~~ 9-(

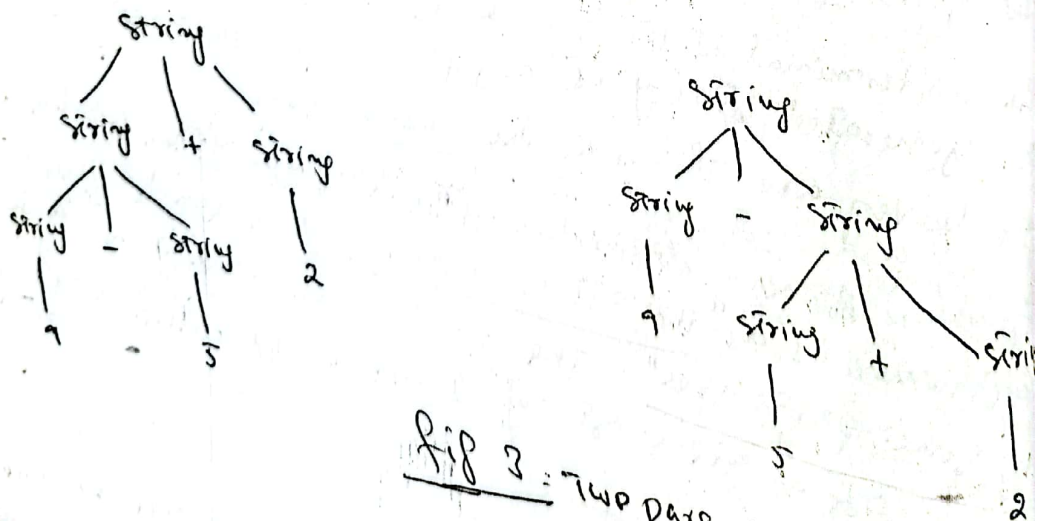


fig 3:- Two parse trees for 9-5+2.

Associativity of operators:

left associative

By Rule, 9+5+2 is equivalent to (9+5)+2 and 9-5-2 is equivalent to (9-5)-2. When an operand like 5 has operators to its left and right, conventions are needed for deciding which operator takes that operand. We say that the operator + associates to the left because an operand with + signs on both sides of it is taken by the operator to its left. In most programming languages the four arithmetic operators addition, subtraction, multiplication and division are left associative.

Some common operators such as exponentiation are right associative. As another example, the assignment operator = in C is right associative in C, the expression a=b=c is treated in the same way as the expression a=(b=c).

Right associative

Strings like a=b=c with a right associative operator are generated by the following grammar:

right → letter = right | letter

letter → a | b | c | ..... | z

The contrast between a parse tree for a left-associative operator like − and a parse tree for a right-associative operator like = is shown in fig 4. Note that the parse tree for 9-5-2 grows down towards the left, whereas the parse tree for a=b=c grows down towards the right.

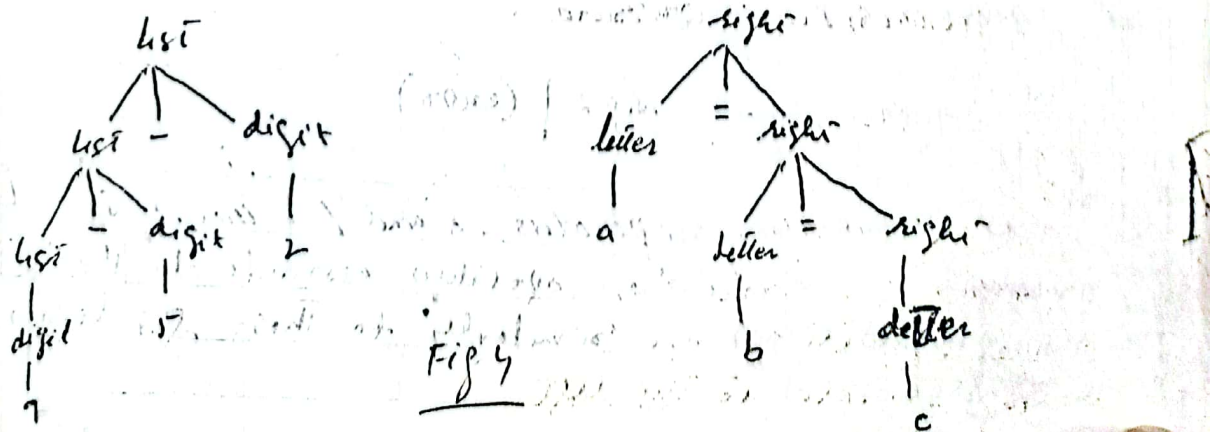


Fig 4

Precedence of operators:

Consider the expression 9+5*2. There are two possible interpretations of this expression: (9+5)*2 or 9+(5*2). The associativity of + and * don't resolve this ambiguity. For this reason, we need to know the precedence of operators.

We say that * has higher precedence than + if * takes its operands before + does. In ordinary arithmetic multiplication and division have higher precedence than addition and subtraction. Therefore, 5 is taken by * in both 9+5*2 and 9*5+2; i.e., the expressions are equivalent to 9+(5*2) and (9*5)+2, respectively.

Syntax of expressions:

A grammar for arithmetic expressions can be constructed from a table showing the associativity and precedence of operators. We start with the four common arithmetic operators and a precedence table, showing the operators in order of increasing precedence

left associative: + -

left associative: * /

We create two nonterminals expr and term for the two level of precedence, and an extra nonterminal factor for generating basic units in expressions. The basic units in expressions are presently digits and parenthesized expressions.

factor → digit | (expr)

Now consider the operators, * and /, that have the high precedence. Since these operators associate to the left, the productions are similarly to those for lists that associate to the left.

term → term * factor
| term / factor
| factor

Similarly, expr generates lists of terms separated by the additive operators.

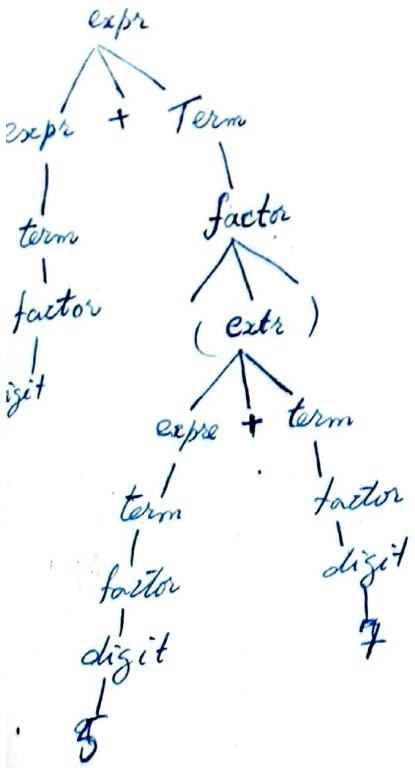
expr → expr + term
| expr − term
| term

The resulting grammar is therefore,

expr → expr + term | expr − term | term
term → term * factor | term / factor | factor
factor → digit | (expr)

This grammar treats an expression as a list of terms separated by either + or − signs, and a term as a list of factors separated by * or / signs.

9+(5+7).



SYNTAX-DIRECTED TRANSLATION

To translate a programming languages, a compiler may need keep track of many quantities besides the code generated for the construct. For example, the compiler may need to know the type of construct or location of the 1st instruction in its target code or the number instructions generated. We therefore talk about attributes associated with constructs. An attribute may represent any quantity, e.g.: a string, a type, a memory location etc.

A syntax directed definition specifies the translation of a construct in terms of attributes associated with its syntactic components.

postfix notation:

9+5
95+ (postfix)
9+5
+95 → Infix

The postfix notation of an expression E can be defined as follows:

1- If E is a variable or constant, then the postfix notation for E is E itself.

2- If E is an expression of the form E_1 op E_2, where op is any binary operation, then the postfix notation for E is $E_1' E_2'$ op, where E_1' and E_2' are the postfix notations of E_1 and E_2, respectively.

3- If E is an expression of the form (E_1), then the postfix notation for E_1 is also the postfix notation for E.

No parentheses are needed in postfix notation because the position and arity (number of arguments) of the operators permits only one decoding of a postfix notation. For example, the postfix notation for (9-5)+2 is 95-2+ and the postfix notation for 9-(5+2) is 952+-.

Syntax-Directed Definitions:

A syntax-directed definition uses a context-free grammar to specify the syntactic structure of

The input. With each grammar symbol, it associates a set of attributes, and with each production, a set of semantic rules for computing values of the attributes associated with the symbols appearing in that production. The grammar and the set of semantic rules constitute the syntax-directed definition.

A translation is an input-output mapping. The output of each input x is specified in the following manner. First, construct a parse tree for x. Suppose a node n in the parse tree is labeled by the grammar symbol X. We write $X.a$ to denote the value of attribute a of X at that node. The value of $X.a$ at n is computed using the semantic rule for attribute a associated with the X-production used at node n. A parse tree showing the attribute values at each node is called an annotated parse tree.

Synthesized Attributes:

An attribute is said to be synthesized if its value at a parse-tree node is determined from attribute values at the children of the node. Synthesized attributes have the desirable property that they can be evaluated during a single bottom-up traversal of the parse tree.

Example 5:

A syntax-directed definition for translating expressions consisting of digits separated by plus or minus signs into postfix notation is shown in fig 5.1. Associated with non each nonterminal is a string-valued attribute t that represents the postfix notation for the expression generated by that nonterminal in a parse tree.

The value of attribute is a string.

production	Semantic Rule
$expr \rightarrow expr_1 + term$	$expr.t := expr_1.t \,\|\|\, term.t \,\|\|\, '+'$
$expr \rightarrow expr_1 - term$	$expr.t := expr_1.t \,\|\|\, term.t \,\|\|\, '-'$
$expr \rightarrow term$	$expr.t := term.t$
$term \rightarrow 0$	$term.t := '0'$
$term \rightarrow 1$	$term.t := '1'$
⋮	⋮
$term \rightarrow 9$	$term.t := '9'$

Fig 5 :- syntax-directed definition for infix to postfix translation

The postfix form of a digit is the digit itself. e.g; the Semantic rule associated with the production term → 9 defines term.t to be 9 whenever this production is used at a node in a parse tree. when the production expr → term is applied, the value of term.t becomes the value of expr.t.

The production $expr \rightarrow expr_1 + term$ derives an expression containing a plus operator. The left operand of the plus operator is given by $expr_1$ and the right operand by term. The Semantic rule

$$expr.t := expr_1.t \,\|\, term.t \,\|\, '+'$$

associated with this production defines the value of attribute expr.t by concatenating the postfix forms $expr_1.t$ and term.t of the left and right operands, respectively and then appending the plus sign. The operator || in Semantic rules represents string concatenation.

Fig 6 contains the annotated parse tree corresponding to the tree of Fig 2. The value of the t-attribute at each node has been computed using the Semantic rule associated with the production used at that node. The value of the attribute at the root is the postfix notation for the string generated by the parse tree.

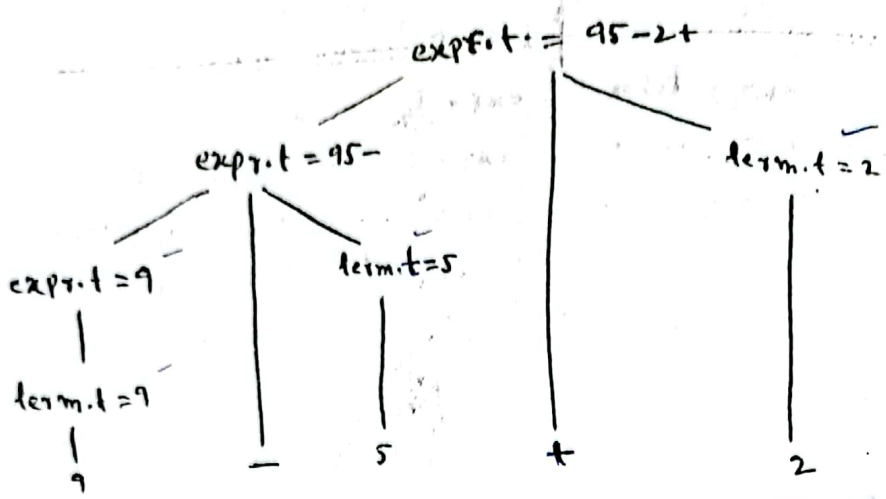


Fig 6 : Attribute values at nodes in a parse tree

Depth-First Traversals

A traversal of a tree starts at the root in some order. Here, Semantic rules will be evaluated using the depth-first traversal defined in Fig 7. It starts at the root and recursively visits the children of each node in left-to-right order. The Semantic rules at a given node are evaluated once all descendants of that node have been visited. It is called "depth-first" because it visits an unvisited child of a node whenever it can, so it tries to visit nodes as far away from the root as quickly as it can.


```
procedure visit (n: node);
begin
    for each child m of n, from left to right do
        visit (m);
    evaluate Semantic rules at node n
end
```

Fig 7 :- A depth-first traversal of a tree.

Translation Schemes

A translation scheme is a context free grammar in which program fragments called semantic actions are embedded within the right sides of productions. A translation scheme is like a syntax - directed definition, except that the order of evaluation of the semantic rules is explicitly shown. The position at which an action is to be executed is shown by enclosing it between braces and writing it within the right side of a production, as in

rest → + term { print('+') } rest1

A translation scheme generates an output for each sentence x generated by the underlying grammar by executing the actions in the order they appear during a depth-first traversal of a parse tree for x. For example, consider a parse tree with a node labeled rest representing a production. The action { print('+') } will be performed after the subtree for term is traversed but before the child for rest1 is visited.

5+2
5-2+

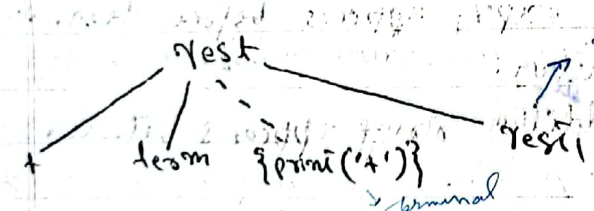


Fig 8: An extra leaf is constructed for a semantic action

When drawing a parse tree for a translation scheme we indicate an action by constructing for it an extra child, connected by a dashed line to the node for its production. For example, the portion of the parse tree for the above production and action is drawn in the above fig 8. The node for a semantic action has no children, so the action is performed when that node is first seen.

Emitting a Translation:

The Semantic action in translation schemes write the output of a translation into a string. when the output is created in this fashion, the order in which the characters are printed is important.

The string representing the translation of the nonterminal on the left side of each production is the concatenation of the translations of the nonterminals on the right, in the same order as in the production, with some additional strings (perhaps none) interleaved. A Syntax-directed definition with this property is deemed simple. For example, consider the first production and semantic rule from the syntax-directed definition of fig 5.

PRODUCTION	SEMANTIC RULE
$expr \rightarrow expr_1 + term$	$expr.t := expr_1.t \,\|\|\, term.t \,\|\|\, '+'$ ——— (5)

Here the translation $expr.t$ is the concatenation of the translations of $expr_1$ and $term$, followed by the symbol $+$. Notice that $expr_1$ appears before $term$ on the right side of the production.

An additional string appears between $term.t$ and $rest_1.t$ in

PRODUCTION	SEMANTIC RULE
$rest \rightarrow + term\ rest_1$	$rest_1.t := term.t \,\|\|\, '+' \,\|\|\, rest_1.t$ ——— (6)

but again, the nonterminal $term$ appears before $rest_1$ on the right side.

Simple syntax-directed definition can be implemented with translation schemes in which actions print the additional strings in the order they appear in the definition. The actions in the following productions print the additional strings in (5) and (6), respectively:

expr → expr + term { print ('+') }
rest → + term { print ('+') } rest$_1$

Example 6 :- Fig 5 contained a simple definition of translating expressions into postfix form. A translation scheme derived from this definition is given in Fig 9 and a parse tree with actions for 9-5+2 is shown in fig 10. Note that although Figures 6 and 10 represent the same input-output mapping, the translation in the two cases is constructed differently; Fig 6 attaches the output to the root of the parse tree, while fig 10 prints the output incrementally.

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

Fig 9:- Actions translating expressions into postfix notation.

The root of fig 10 represents the first production in fig 9. On a depth-first traversal, we first perform all the actions in the subtree for the left operand expr when we traverse the leftmost subtree of the root. we then visit the leaf + at which there is not action. we next perform the actions in the subtree for the right operand term and finally, the semantic action { print ('+') } at the extra node.

Since the productions for term have only a digit on the right side, that digit is printed by the

actions for the productions. No output is necessary for the production expr → term, and only the operator needs to be printed in the action for the first three productions. When executed during a depth-first traversal of the parse tree, the actions in fig 10 print 95-2+.

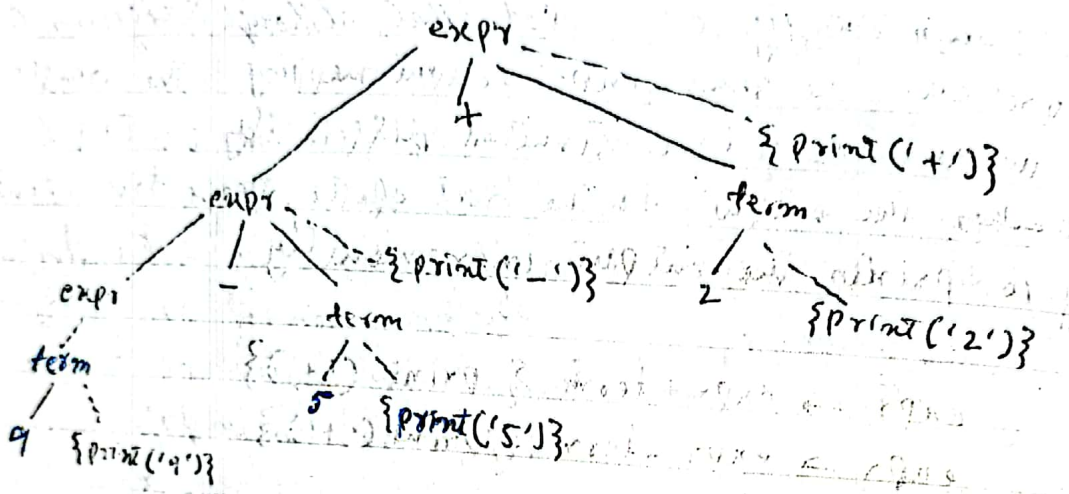


Fig 10:- Actions translating 9-5+2 into 95-2+.

As a general rule, most parsing methods process their input from left to right in a "greedy" fashion; that is they construct as much of a parse tree as possible before reading the next input token. In a simple translation scheme, actions are also done in a left to right order.

PARSING :-

Passing is the process of determining if a string of tokens can be generated by a grammar.

Most passing methods have two classes, called the top-down and bottom-up methods. These terms refer to the order in which nodes in the parse tree are constructed. In the former, construction starts at the root and proceeds towards the leaves, while in the latter, construction starts at the leaves and proceeds towards the root.

Top-Down Passing :-

The following grammar generates a subset of the types of Pascal ; we use the token dotdot for "..".

type → simple
| ↑ id
| array [simple] of type

simple → integer
| char
| num dotdot num

(simple is a nonterminal)

The top-down construction of a parse tree is done by starting with the root, labeled with the starting non-terminal, and repeatedly performing the following two steps

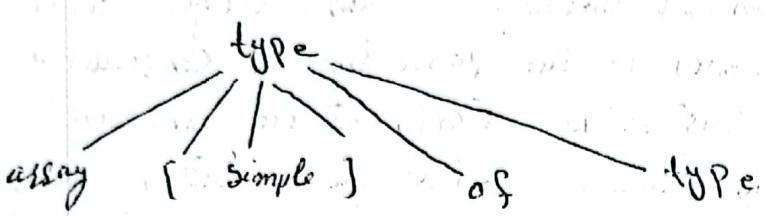
1- At node n, labeled with nonterminal A, select one of the productions for A and construct children at n for the symbols on the right side of the production.

2- Find the next node at which a subtree is to be constructed.

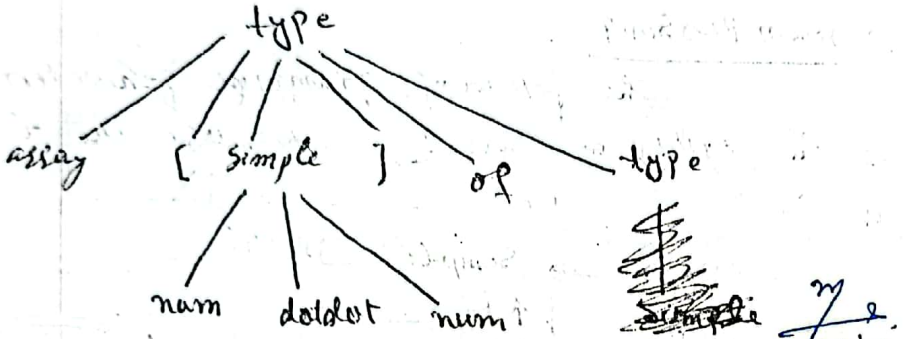
The top-down construction of the parse tree for the above grammar is:

(a) type

(b)



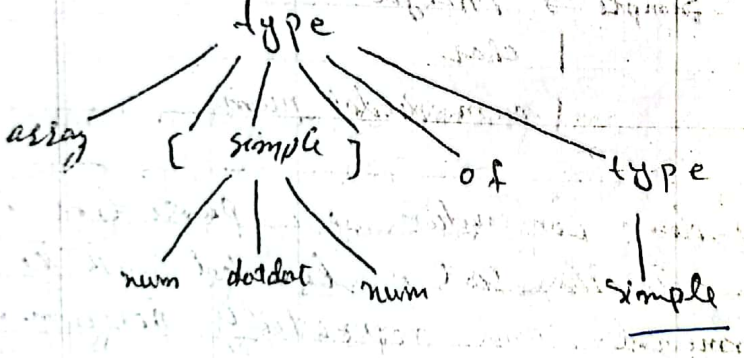
(c)



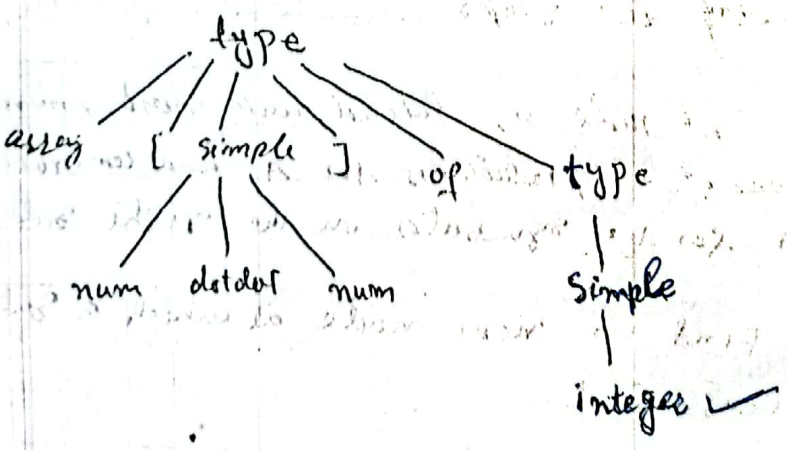
m & tariq.

m & tariq.

(d)



(e)



Integer

LEXICAL ANALYSIS

The Role of the LEXICAL ANALYZER:

The lexical analyzer is the first phase of a compiler. Its main task is to read the input characters and produce an output as sequence of tokens that the parser the parser uses for syntax analysis. The interaction of the lexical analyzer with parser is shown in fig 1.

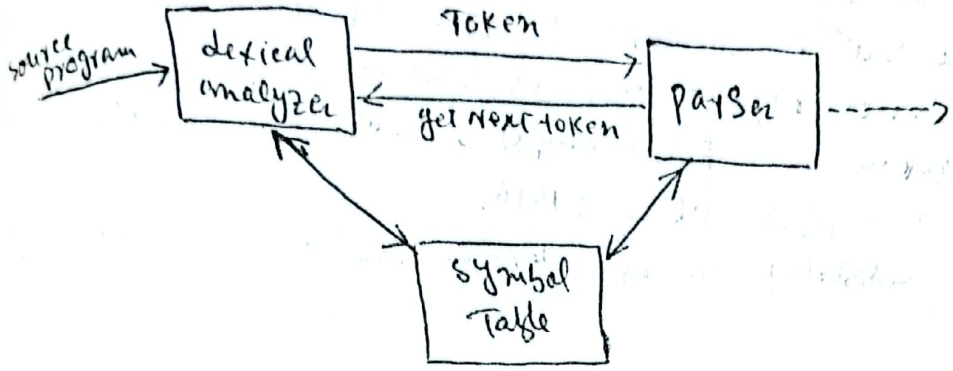


Fig 1:- Interaction of lexical Analyzer with parser

Since the lexical analyzer is the part of the compiler that reads the source text, it may also perform certain secondary tasks. one such task is stripping out from the source program comments and white space in the form of blank, tab, and newline characters. Another is correlating error messages from the compiler with the source program. For example, the lexical analyzer may keep track of the number of newline characters seen, so that a line number can be associated with an error message.

Sometimes, lexical analyzers are divided into two phases, the first called "scanning" and the 2nd "lexical analysis". The scanning is responsible for doing simple tasks, while the lexical analyzer does the more complex operations. For example, a Fortran compiler might use a scanner to eliminate blanks from the input.

Tokens, Patterns, Lexemes:

When talking about lexical analysis, we use the terms token, pattern and lexeme with specific meaning. Example of their use are shown in fig 2.

In general, there is a set of strings in the input for which the same token is produced as output. This set of strings is described by a rule called a **Pattern** associated with the token. The pattern is said to match each string in the set. A lexeme is a sequence of characters in the source program that is matched by the pattern for a token. For example, in the Pascal statement

```
Const pi = 3.1416;
```

The substring pi is a lexeme for the token "identifier".

Token	SAMPLE LEXIMES	INFORMAL DESCRIPTION OF PATTERN
Const	Const	Const
if	if	if
relation	<, <=, =, <>, >, >=	< or <= or = or <> or >= or >
id	pi, count, D2	letter followed by letters and digits
num	3.1416, 0.6, 6.02E23	any numeric constant
literal	"Aglam"	any character between " and "

Fig 2:- Examples of tokens.

We treat tokens as terminal symbols in the grammar for the source language, using underline names to represent tokens. (The lexemes matched by the pattern for the token represent strings of characters in the source program that can be treated together as a lexical unit.)

In most programming languages, the following constructs treated as tokens: keywords, operators, identifiers, constants, literal strings, and punctuation symbols such as parentheses, commas, and semicolons. In the example above, when the character sequence pi appears in a source program, a token representing an identifier is returned to the parser.

A pattern is a rule describing the set of lexemes that can represent a particular token in source programs.

Attributes for Tokens:

The lexical analyzer collects information about tokens into their associated attributes. As a practical matter, a token has usually only a single attribute — a pointer to the symbol-table entry in which the information about the token is kept; the pointer becomes the attribute for the token.

Example 1 :- The tokens and associated attribute-values for the FORTRAN statement

E = M * C ** 2

Pointer is used for identifiers.

are written below as a sequence of pairs:

< id, pointer to symbol-table entry for E >
< assign_op, >
< id, pointer to symbol-table entry for M >
< mult_op, >
< id, pointer to symbol-table entry for C >
< exp_op, >
< num, integer value 2 >

Lexical Errors - Of the string fi is encountered in a C program for the first time in the context

fi (a == f(x)) ----

a lexical analyzer cannot tell whether fi is a misspelling of the keyword if or an undeclared function identifier. Since fi is valid identifier, the lexical analyzer must return the token for an identifier and let some other phase of the compiler handle any error.

But suppose a situation does arise in which the lexical is unable to proceed because none of the patterns for tokens matches a prefix of the remaining input. The simplest recovery strategy is "panic mode" recovery. We delete the successive characters from the remaining input until the lexical analyzer can find a well-formed token.

other possible error-recovery actions are :

1- deleting an extraneous character
2 - inserting a missing character
3 - replacing an incorrect character by a correct character
4 - transposing two adjacent characters

INPUT BUFFERING :

Buffer Pairs :-

For many source languages, the lexical analyzer needs to look ahead several characters beyond the lexeme for a pattern before a match can be announced.

We use a buffer divided into two N-character halves as shown in fig 3.

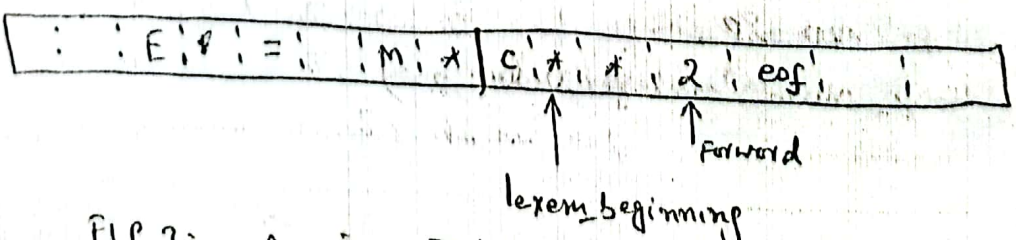


Fig 3 :- An input buffer in two halves

we read N input characters into each half of the buffer with the 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. That is, eof marks the end of the source file.

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, a 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.

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 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.

Sentinels :-

If we use the scheme of fig 3, 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 shown in fig 4.6.

```
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 the beginning of first half
end
else forward := forward + 1 ;
```

Fig 4:- Code to advance forward pointer.

Except at the ends of the buffer halves, the code in fig 4 required 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 5 shows the same buffer arrangement as fig 3 with the sentinels added

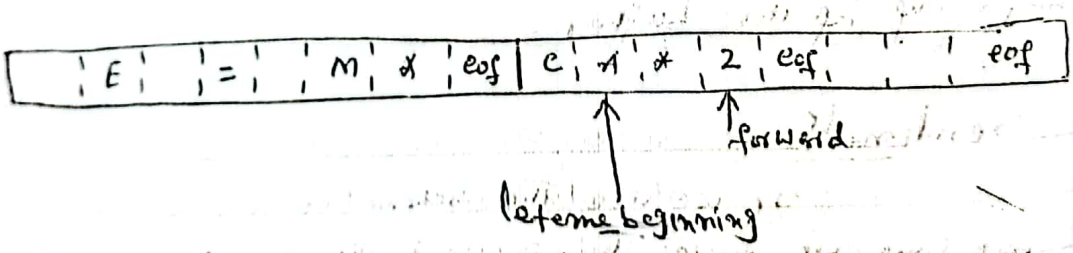


Fig 5:- Sentinels at end of each buffer half.

With the arrangement of Fig 5, we can use the code shown in fig 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.

```
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
        terminate lexical analysis
end
```

Fig:- Lookhead code with Sentinels

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 a set of strings.

String 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.

A string over some alphabet is a finite sequence of symbols drawn from that alphabet. In language theory, the terms sentence and word often denote "string".

The length of a string s, usually written |s|, is the number of occurrence 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.

The term language denotes any set of strings over some fixed alphabet. Languages like ϕ, the empty set or $\{\epsilon\}$, the set containing only the empty string, are languages under this definition.

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 = dog and y = house, then xy = 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.

Some common terms associated with parts of string are shown in fig 7.

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 string 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

...r prefix, suffix, ... substring of s	Any nonempty string x that is, respectively, a prefix, suffix, or substring of s such that $s \neq x$.
...equence 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 8:- Terms for parts of a string

Operations and Languages :-

There are several important notations that can be applied to languages, which are union, concatenation, and closure as show in fig 8. we can also generalize the "exponentiation" operator to languages by defining L^0 to be $\{\epsilon\}$ and L^i to be $L^{i-1}L$:-

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^*	$L^* = \bigcup_{i=1}^{\infty} L^i$ concatenation is not zero. L^* denotes "zero or more concatenations of L"
positive closure of L written L^+	$L^+ = \bigcup_{i=1}^{\infty} L^i$ start = 1 L^+ denotes "one or more concatenations of L"

Fig 8:- Definitions of operations on Languages

Example 2: Let L be the set {A, B, ... Z, a, b, c, ... z} and D the set {0, 1, ... 9}.

The new languages created from L and D by applying the above operations are as follows:

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.
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. But ~~there~~, here, with a notation, called regular expression, we might define Pascal identifiers as

letter (letter | digit)* (| = or; concatenation of ...)

The vertical bar here means "or" the parentheses are used to group subexpressions, the star means "zero or more instances of" the parenthesized expressions.

Each regular expression r denotes a language L(r). The defining rules specify how L(r) is formed by combining in various ways the languages denoted by the subexpressions of r.

Here are the rules that define the regular expression over alphabet Σ. Associated with each rule is a specification of the language denoted by the regular expression being defined.

ϵ is a regular expression that denotes $\{\epsilon\}$, i.e. the set containing the empty string.

If a is a symbol in Σ, then a is a regular expression that denotes $\{a\}$ i.e. the set containing the string a.

Suppose r and s are regular expressions denoting the languages $L(r)$ and $L(s)$. Then,

a) $(r)|(s)$ is a regular expression denoting $L(r) \cup L(s)$

b) $(r)(s)$ is a regular expression denoting $L(r)L(s)$

c) $(r)^*$ is a regular expression denoting $(L(r))^*$

d) $(r)^+$ is a regular expression denoting $L(r)^+$

Language denoted by a regular expression is said to be regular set.

Unnecessary parentheses can be avoided in regular expressions if we adopt the conventions that:

1. The unary operator * has the highest precedence.
2. Concatenation has the second highest precedence.
3. | has the lowest precedence.

Under these conventions, $(a)|((b)^*(c))$ is equivalent to $a|b^*c$. Both expressions denote the set of strings that are like a single a or zero or more b's followed by one c.

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

1. The regular expression $a|b$ denotes the set $\{a, b\}$
2. The regular expression $(a|b)(a|b)$ denotes $\{aa, ab, ba, bb\}$, The set of all strings of a's and b's of length two
3. The regular expression a^* denotes the set of all strings of zero or more a's i.e. $\{\epsilon, a, aa, aaa, \ldots\}$

4- The regular expression $(a|b)^*$ denotes the set of all strings containing zero or more instances of an a or b, i.e; the set of all strings of a's and b's. Another regular expression for this set is $(a^*b^*)^*$.

5- The regular expression $a|a^*b$ denotes the set containing the string a and all strings consisting of zero or more a's followed by a b.

Tariq

If two regular expressions r and s denote the same language, we say r and s are equivalent and write $r = s$. For example, $(a|b) = (b|a)$

Fig 9 shows the some algebraic laws that hold for regular expressions r, s, and t.

AXIOM	DESCRIPTION
$r\|s = s\|r$	| is Commutative
$r\|(s\|t) = (r\|s)\|t$	| is associative.
$(rs)t = r(st)$	Concatenation is associative
$r(s\|t) = rs\|rt$ $(s\|t)r = sr\|tr$	Concatenation distributes over |
$\epsilon r = r$ $r\epsilon = r$	ϵ is the identity element for concatenation
$r^* = (r\|\epsilon)^*$	relation between * and ϵ
$r^{**} = r^*$	* is idempotent

Fig 9: Algebraic properties of Regular expressions.

gular Definitions :-

If Σ is an alphbet of basic symbols, a regular definition is a sequence of definitions of form

$$d_1 \longrightarrow r_1$$
$$d_2 \longrightarrow r_2$$
$$\vdots$$
$$d_n \longrightarrow r_n$$

re each d_i is a distinct name over the symbols in $\cup\{d_1, d_2, \ldots, d_{i-1}\}$ i.e., the basic symbols and the viously defined names.

AMPLE 4 :- The set of Pascal identifiers is the set of strings of letters and digits beginning with a letter. The regular definition for this set is

letter $\longrightarrow$ A | B | ---- | Z | a | b | ---- | z
digit $\longrightarrow$ 0 | 1 | 2 | ---- | 9
id $\longrightarrow$ letter (letter | digit)*

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

EXAMPLE 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 :-

digit $\longrightarrow$ 0 | 1 | ---- | 9
digits $\longrightarrow$ digit digit*
optional fraction $\longrightarrow$. digits | ϵ
optional exponent $\longrightarrow$ (E (+ | - | ϵ) digits) | ϵ
num $\longrightarrow$ digits optional fraction optional exponent

Notational shorthands:-

Notational shorthands for components in Regular expressions are instances

$r^+ = r r^*$

1- one or more instances: The unary operator + means "one or more instances of". If r is a regular expression that denote 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 two algebraic identities $r^* = r^+ | \epsilon$ and $r^+ = r r^*$ relate the Kleene and positive closure operators.

2- Zero or one instance:- The unary operator ? means "zero or one instance of". The notation r? is a shorthand for $r | \epsilon$. 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 5 as

digit → 0 | 1 | - - - | 9

digits → digit+ → 1 or more instance not zero.

optional_fraction → (. digits)?

optional_exponent → (E (+|-)? digits)?

num → digits optional_fraction optional_exponent

3- 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] denote the regular expression a|b|c|- - - - |z.

RECOGNITION OF TOKENS:

EXAMPLE 6: Consider the following grammar fragment:

stmt → if expr then stmt
| if expr then stmt else stmt
| ∈

expr → term relop term
| term

term → id | num

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

if → if
then → then
else → else
relop → < | <= | = | <> | > | >=
id → letter (letter | digit)*
num → digit+ (. digit+)? (E (+|-)? digit+)?

where letter and digit are as defined in example 4.

For this language, the lexical analyzer will recognize the keywords if, then, else as well as the lexemes denoted by relop, id, and num. For simplicity we assume keywords are reserved; that is, they cannot be used as identifiers.

In addition, we assume lexemes are separated by white space, consisting of 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 expression ws below

delim → blank | Tab | newline

WS → delim

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.

The lexical analyzer isolate the lexeme for the next token in the input buffer and produce as output a pair consisting of its appropriate token and attribute-value using the translation Table in Fig 10. The attribute-values for the relational operators are given by the symbolic constants LT, LE, EQ, NE, GT, GE.

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

token

Fig 10:- Regular-expression Patterns for Tokens

Transition Diagrams: As an intermediate step in the construction of a lexical analyzer, we first produce a stylized flowchart, called a transition diagram. Transition diagrams depict the actions that take place when a lexical analyzer is called by the parser to get the next token. Suppose that the input buffer is as in Fig 3 and the lexeme_beginning pointer points to the character following the last lexeme found. We use a transition diagram to keep track of information about characters that are seen as forward pointer scans the input. We do so by moving from position to position in diagram as characters are read.

Positions in a transition diagram are drawn as circles and are called states. The states are connected by arrows, called edges. Edges leaving state s have labels indicating the input characters that can next appear after the transition diagram has reached state s. The label other refers to any character that is not indicated by any of the other edges leaving s.

One state is labeled the start state: it is the initial state of the transition diagram where control resides when we begin to ~~regogn~~ recognize a token. Upon entering a state we read the next input character. If there is an edge from the current state whose label matches this input character, we then go to the state pointed to by the edge. Otherwise, we indicate failure.

Fig 11 shows a transition diagram for the patters >= and >. The transition diagram works as follows:-

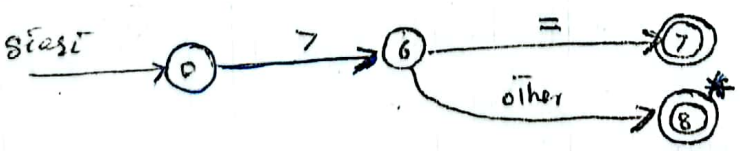


Fig:- 11 Transition diagram for >=

its start state is state 0. In state 0, we read the next input character. The edge labeled > from state 0 is to be followed to state 6 if this input character is >. Otherwise we have failed to ~~rego~~ recognize either > or >=. In state 6 we read the next input character. The edge labeled = from state 6 is to be followed to state 7 if this input character is =. Otherwise, the edge labeled other indicates that we are to go to state 8. The double circle on state 7 indicates that it is an accepting state, a state in which the token >= has been found.

Notice that the character > and another extra character are read as we follow the sequence of edges from the start state to the accepting state 8. Since the extra character is not a part of the relational operator >, we must retract the forward pointer one character. We use a * to indicate states on which this input retraction must take place.

In general, there may be several transition diagrams, each specifying a group of tokens. If failure occurs while we are following one transition diagram, then we retract the forward pointer to where it was in the next state of this diagram, and activate the next transition diagram. If failure occurs in all transition diagrams, then a lexical error has been detected.

The End of The News.