Programming Language

In computer programming, a programming Language serves as a means of communication between the person with a problem and the computer used to help solve it. An effective programming language enhances both the development and the expression of computer programs. It must bridge the gap between the often unstructured nature of human thought and precision required for computer execution.

A hierarchy of programming languages based on increasing machine independence includes the following:

1. **Machine – Level Languages.**
2. **Assembly Languages.**
3. **High – level Language.**
4. **Problem – Oriented Languages.**

1) **Machine – Level Languages:-** is the lowest form of computer language. Each instruction in a program is represented by a numeric code, and numerical addresses are used throughout the program to refer to memory locations in the computer's memory.

2) **Assembly Languages:-** is essentially a symbolic version of a machine-level language. Each operation code is given a symbolic code such as ADD for addition and MUL for multiplication.

Assembly-language systems offer certain diagnostic and debugging assistance that is normally not available at the machine level.

3) **High level language:-** such as FORTRAN, PASCAL, C++, …,etc. it offers most of the features of an assembly language. While some facilities for accessing system-level features may not be provided, a high-level language offers a more enriched set of language features such as structured control constructs, nested statements, blocks, and procedures.

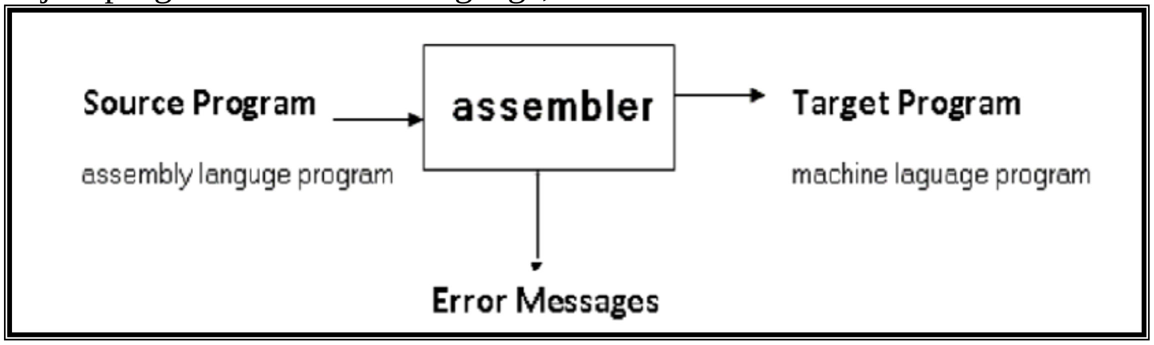
4) **A problem-oriented language:-** treating problems in a specific application or problem area. Such as (SQL) for database retrieval applications and COGO for civil engineering applications.

Programming language (high level language) can be depicted as notations for describing computation to people and to machine. The world as we know it depends on programming languages, because all the software running on all the computers was written in some programming language. But, before a program can be run, it first must be translated into a form in which it can be executed by a computer. The software to do this translation is called ***Compiler***.

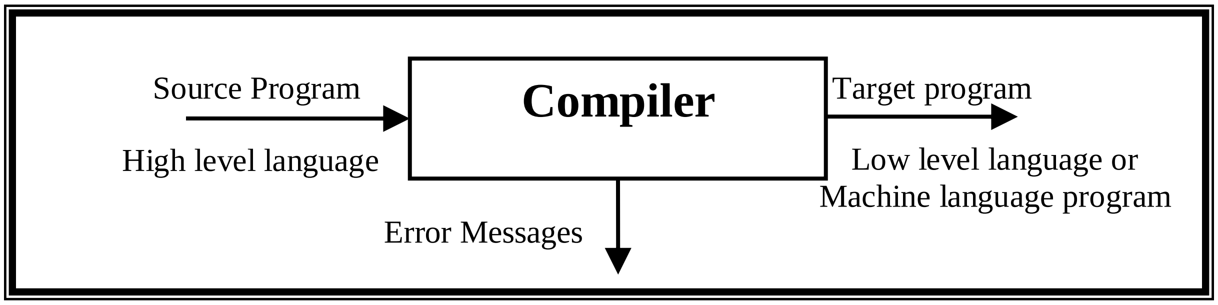
Translator

A translator is program that takes as input a program written in a given programming language (the source program) and produce as output program in another language (the object or target program). As an important part of this translation process, the compiler reports to its user the presence of errors in the source program.

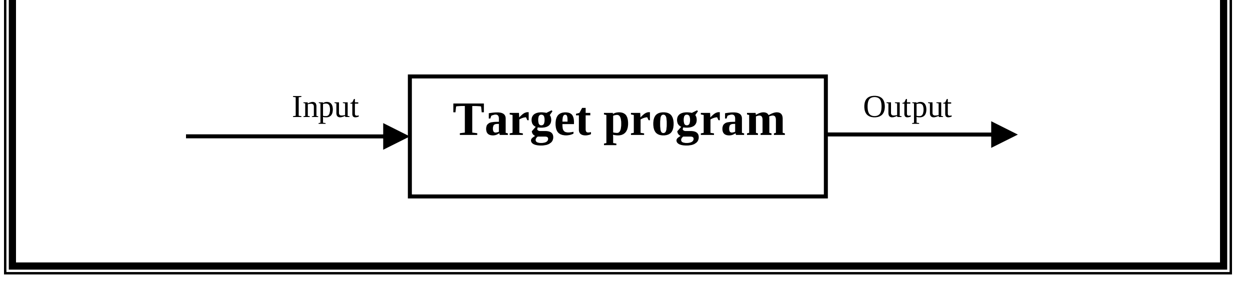
If the source language being translated is assembly language, and the object program is machine language, the translator is called **Assembler**.



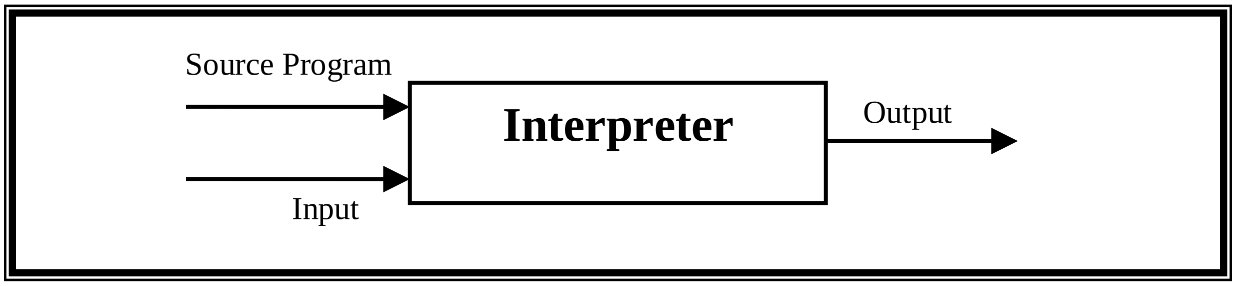
A translator, which transforms a high level language such as C in to a particular computers machine or assembly language, called **Compiler**. A **compiler** translates (or *compiles*) a program written in a high-level programming language that is suitable for human programmers into the low-level machine language that is required by computers. During this process, the compiler will also attempt to spot and report obvious programmer mistakes.



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

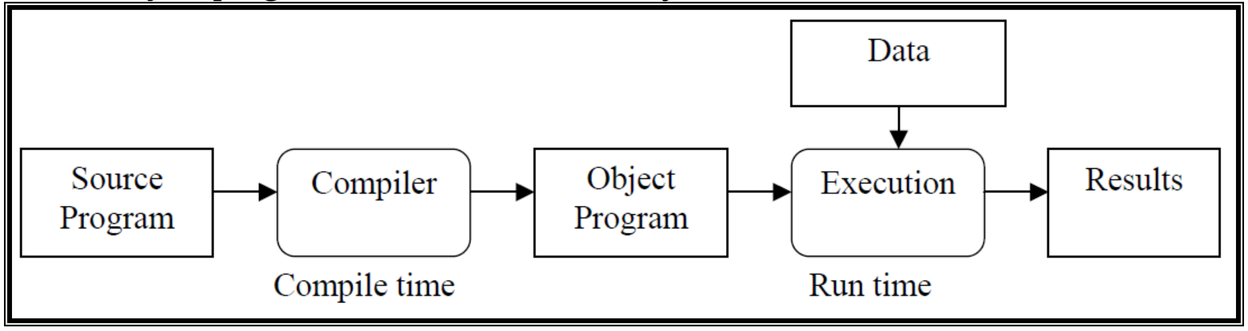


Another common kind of translator (language processors) called an **Interpreter,** in this kind, instead of producing a target program as a translation, an interpreter appears to directly execute the operations specified in the source program on inputs supplied by the user, Figure below shows the role of interpreter



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

Note :- The execution of a program written in a high level language is basically take two steps:-
1- The source program must first be compiled (translated into the object program)
2- The object program is loaded into memory to be executed.

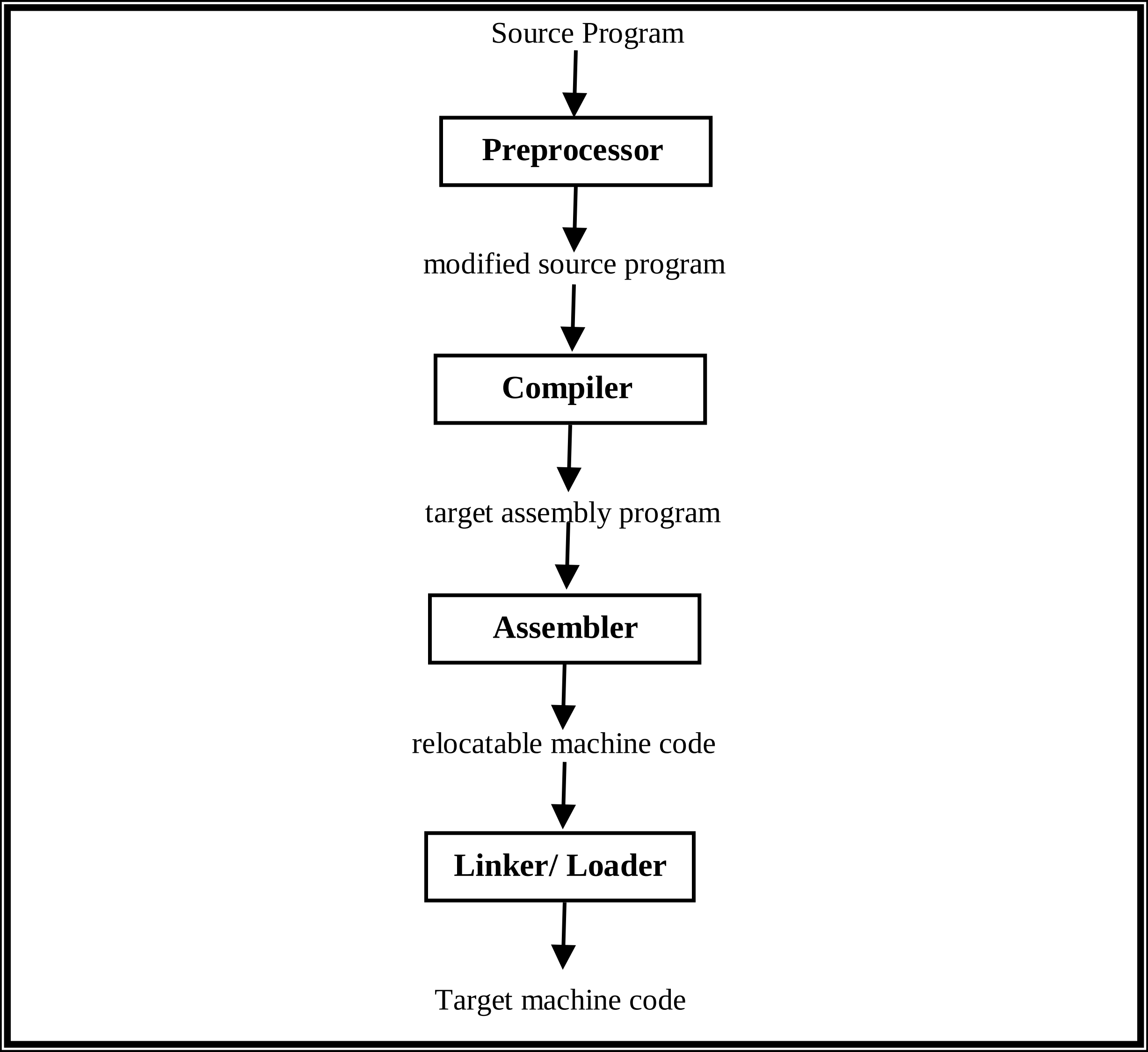


The time used to compile the program called ***Compile time*** and time that used to run the object program is called ***Run time***.

In addition to a compiler, several other programs may be required to create an executable target program, as shown in figure below. A source program may be divided into modules stored in separate files. The task of collecting the source program is sometimes entrusted to a separate program, called a ***preprocessor***.

The modified source program is then fed to a compiler. The compiler may produce an assembly-language program as its output, because assembly language is easier to produce as output and is easier to debug. The assembly language is then processed by a program called an *assembler* that produces relocatable machine code as its output.

Large programs are often compiled in pieces, so the relocatable machine code may have to be linked together with other relocatable object files and library files into the code that actually runs on the machine. The *linker* resolves external memory addresses, where the code in one file may refer to a location in another file. The *loader* then puts together all of the executable object files into memory for execution.



The Structure of Compiler

The Compiler consists of two parts:

1. ***Analysis***.
2. ***Synthesis***.

The ***analysis*** part breaks up the source program into constituent pieces and imposes a grammatical structure on them. It then uses this structure to create an intermediate representation of the source program. If the analysis part detects that the source program is either syntactically ill formed or semantically unsound, then it must provide informative messages, so the user can take corrective action. The analysis part also collects information about the source program and stores it in a data structure called a ***symbol table***, which is passed along with the intermediate representation to the synthesis part.

The ***synthesis*** part constructs the desired target program from the intermediate representation and the information in the symbol table. The analysis part is often called the ***front end*** of the compiler; the synthesis part is the ***back end***. If we examine the compilation process in more detail, we see that it operates as a sequence of ***phases***, each of which transforms one representation of the source program to another. A typical decomposition of a compiler into phases is shown in the following Figure. In practice, several phases may be grouped together, and the intermediate representations between the grouped phases need not be constructed explicitly. The symbol table, which stores information about the entire source program, **is used by all phases of the compiler**. Some compilers have a **machine-independent optimization phase** between the front end and the back end. The purpose of this optimization phase is to perform transformations on the intermediate representation, so that the back end can produce a better target program than it would have otherwise produced from an unoptimized intermediate representation. Since optimization is optional, one or the other of the two optimization phases shown in Fig. 1.6 may be missing.

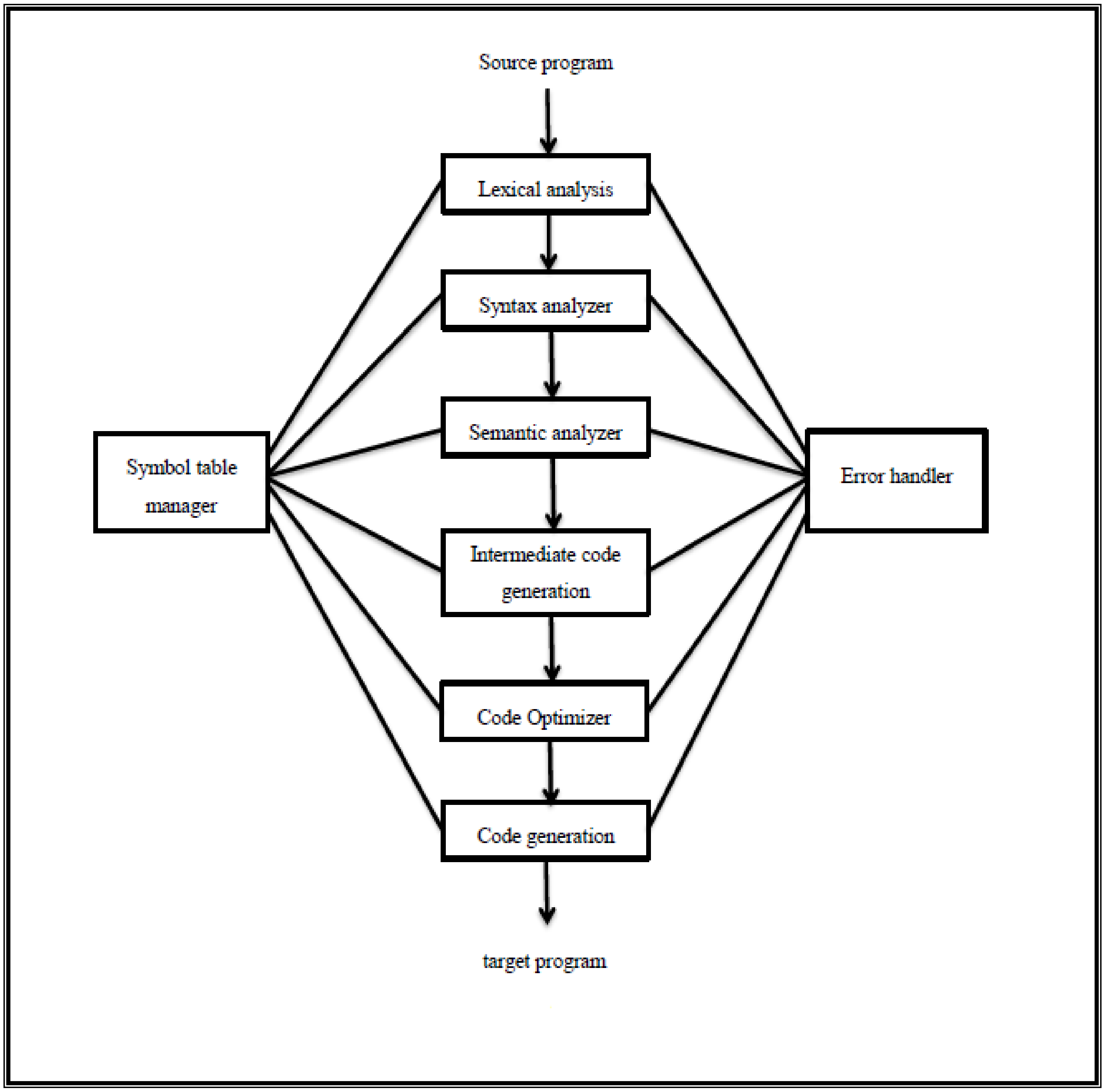


Figure () shows the phases of compilers

The Compiler Phases

1. Lexical analysis

The first phase of a compiler is called ***lexical analysis*** or ***Scanning*** or ***Lexer***. The lexical analyzer reads the stream of characters making up the source program and groups the characters into meaningful sequences called ***lexemes***. From each lexeme, the lexical analyzer produces as output a token of the form

<token – name, attribute – value>

that it passes on to the subsequent phase, syntax analysis. In the token, the first component token-name is an abstract symbol that is used during syntax analysis, and the second component ***attribute-value*** points to an entry in the symbol table for this token. Information from the symbol-table entry Is needed for semantic analysis and code generation.

For example, suppose a source program contains the assignment statement

position = initial + rate * 60

The characters in this assignment could be grouped into the following lexemes and mapped into the following tokens passed on to the syntax analyzer:

id is an abstract symbol standing for identifier

Lexems	Token
position	<id,1>
=	<=>
initial	<id,2>
+	<+>
rate	<id,3>
*	<*>
60	<60>

symbol-table entry for initial

Note) Blanks separating the lexemes would be discarded by the lexical analyzer.

2. Syntax Analysis

The syntax analyzer groups the tokens together into syntactic structures. This phase is called ***syntax analysis*** or ***parsing***. The parser uses the first components of the tokens produced by the lexical analyzer to create a **tree-like intermediate representation that depicts the grammatical structure of the token stream**. A typical representation is a ***syntax tree*** in which each interior node represents an operation and the children of the node represent the arguments of the operation.

The subsequent phases of the compiler use the grammatical structure to help analyze the source program and generate the target program.

3. .Semantic Analysis

The semantic analyzer uses the syntax tree and the information in the symbol table to check the source program for semantic consistency with the language definition. It also gathers type information and saves it in either the syntax tree or the symbol table, for subsequent use during intermediate-code generation.

An important part of semantic analysis is ***type checking***, where the compiler checks that each operator has matching operands. For example, many programming language definitions require an array index to be an integer; the compiler must report an error if a floating-point number is used to index an array. The language specification may permit some type conversions called coercions. For example, a binary arithmetic operator may be applied to either a pair of integers or to a pair of floating-point numbers. If the operator is applied to a floating-point number and an integer, the compiler may convert or coerce the integer into a floating-point number

4. Intermediate Code Generation

In this process of translating a source program into target code, a compiler may construct one or more intermediate representations, which can have a variety of forms. Syntax trees are a form of intermediate representation; they are commonly used during syntax and semantic analysis.

After syntax and semantic analysis of the source program, many compilers generate an explicit low – level or machine – like intermediate representation, which we can think of as a program for an abstract machine. This intermediate representation should have two important properties: it should be easy to produce and it should be easy to translate into the target machine.

5. Code optimization

This is optional phase designed to improve the intermediate code so that better target code will result. Usually better means faster, but other objectives may be desired, such as shorter code, or target code that consumes less power. For example, a straightforward algorithm generates the intermediate code Figure using an instruction for each operator in the tree representation that comes from the semantic analyzer. A simple intermediate code generation algorithm followed by code optimization is a reasonable way to generate good target code.

6. Code Generation

The code generator takes as input an intermediate representation of the source and maps it into the target language. If the target language is machine code, registers or memory locations are selected for each of the variables used by the program. Then, the intermediate instructions are translated into sequences of machine instructions that perform the same task. A crucial aspect of code generation is the judicious assignment of registers to hold variables. Designing a code generator that produces truly efficient object programs is one of the most difficult parts of compiler design.

Error Handling

One of the most important functions of a compiler is the detection and reporting of errors in the source program.

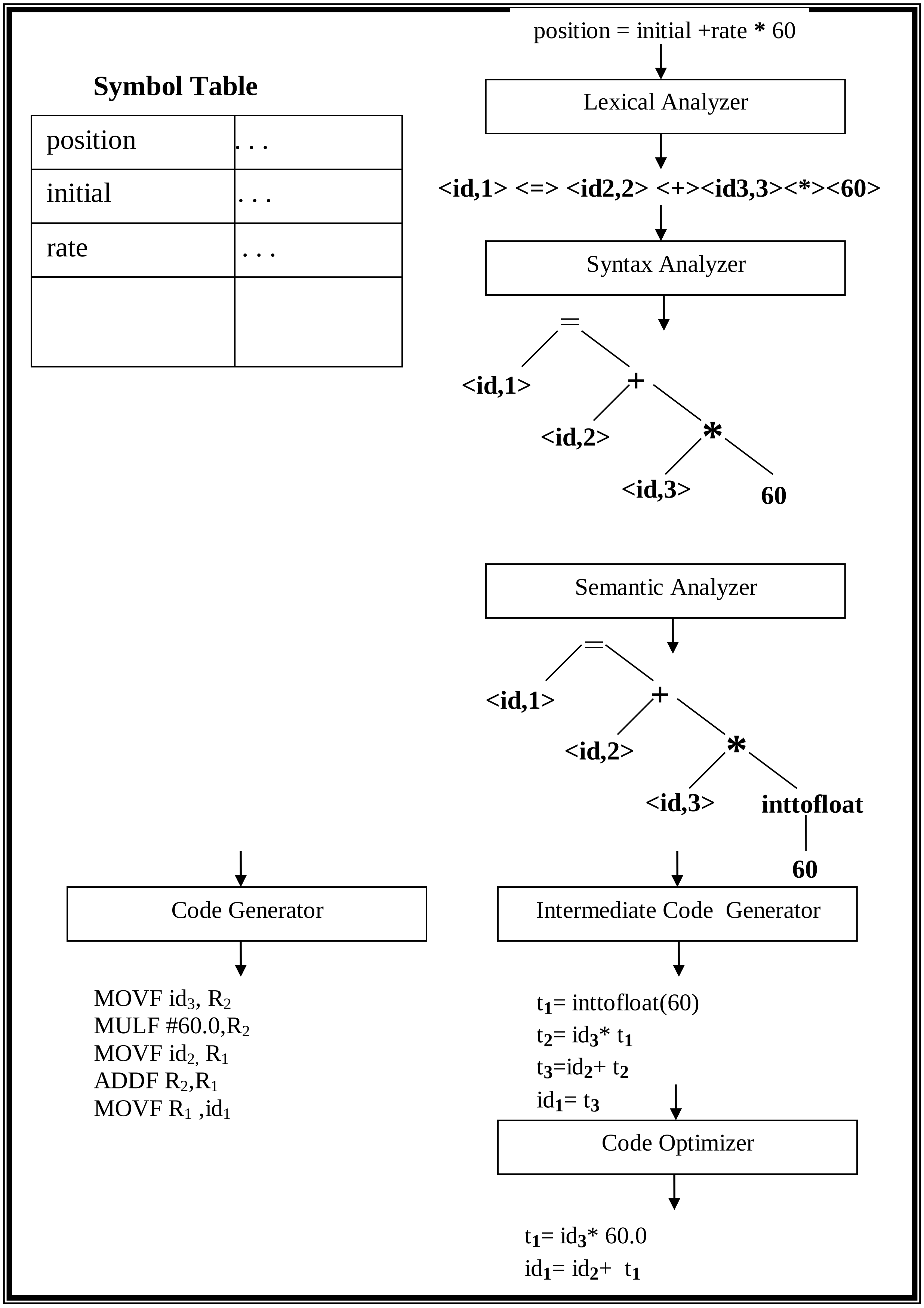
Types of Errors

- **Lexical errors**:- the first phase can detect errors where characters remaining in the input don't form any token of language, few errors are discernible at the lexical level alone, because a lexical analyzer has a very localized view of the source program. For example, For instance, if the string **fi** is encountered for the first time in a C program in the context:

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

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

- **Syntactic errors**:- the syntax phase can detect errors where the token stream violates the structure rules(syntax) of the language.
- **Semantic errors**:- during semantic analysis phase 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 identifiers, one which is the name of an array and the other is the name of procedure.
- **A runtime error**:- means an error which happens, while the program is running.



position	. . .
initial	. . .
rate	. . .

t_1= inttofloat(60)
t_2= id_3* t_1
t_3=id_2+ t_2
id_1= t_3

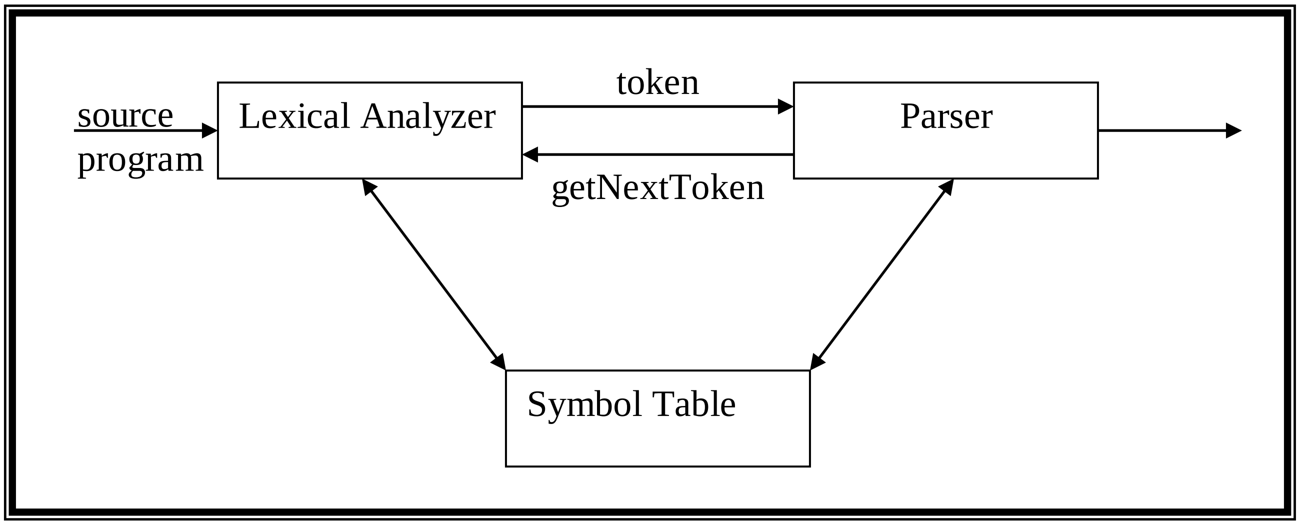
t_1= id_3* 60.0
id_1= id_2+ t_1

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

Lexical Analysis

It is the first phase of a compiler, ***the main task of the lexical analyzer is to read the input characters of the source program, group them into lexemes, and produce as output a sequence of tokens for each lexeme in the source program.*** The stream of tokens is sent to the parser for syntax analysis. The lexical analyzer interacts with the symbol table. When the lexical analyzer discovers a lexeme constituting an identifier, it needs to enter that lexeme into the symbol table.

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



Interactions between the lexical analyzer and the parser

Other tasks may perform by the lexical analyzer:

1) stripping out the comments and whitespace(blank, newline, tab, and others).

2) correlating error messages generated by the compiler with source program.

Sometimes, lexical analyzers are divided into a cascade of two processes:

a) ***Scanning*** consists of the simple processes that do not require tokenization of the input, such as deletion of comments and compaction of consecutive whitespace characters into one.

b) ***Lexical analysis*** proper is the more complex portion, where the scanner produces the sequence of tokens as output.

Issues in Lexical Analysis

There are several reason for separating analysis portion of compiler into Lexical and Parsing, some of these reasons are:

1. Simpler design is the most important consideration. This separation allows us to simplify at least one of these tasks. For example removing white space in the lexical analysis make it easer from dealing with it as syntactic units in parser.
2. Compiler efficiency is improved. A separate lexical analyzer allows us to apply specialized techniques that serve only the lexical task, not the job of parsing. In addition, specialized buffering techniques for reading input characters can speed up the compiler significantly.
3. Compiler portability is enhanced. Input – device – specific peculiarities can be restricted to the lexical analyzer.

Tokens, Patterns, and Lexemes

When discussing lexical analysis, we use three related but distinct terms:

• A ***token*** is a pair consisting of a token name and an optional attribute value. The token name is an abstract symbol representing a kind of lexical unit, e.g., a particular keyword, or a sequence of input characters denoting an identifier. The token names are the input symbols that the parser processes

• A ***pattern*** is a description of the form that the lexemes of a token may take. In the case of a keyword as a token, the pattern is just the sequence of characters that form the keyword. For identifiers and some other tokens, the pattern is a more complex structure that is *matched* by many strings.

• A ***lexeme*** is a sequence of characters in the source program that matches the pattern for a token and is identified by the lexical analyzer as an instance of that token.

Token	Informal description	Lexeme
if	Characters i, f	if
else	Characters e, l, s, e	else
comparison	< or > or <= or >= or == or!=or	
id	Letter followed by letter of digits	pi, score, D2
literal	anything but ",surrounded by " 's	"the first"

Examples of token

The lexical analyzer returns to parser a representation for the token it has found. This representation is:

• an integer code if there is a simple construct such as a left parenthesis, comma or colon .

• or a pair consisting of an integer code and a pointer to a table if the token is more complex element such as an identifier or constant.

Symbol Table

Symbol tables are data structures that are used by compilers to hold information about source-program constructs. The information is collected incrementally by the analysis phases of a compiler and used by the synthesis phases to generate the target code. Entries in the symbol table contain information about an identifier such as its character string (or lexeme), its type, its position in storage, and any other relevant information. Symbol tables typically need to support multiple declarations of the same identifier within a program.

The symbol table must be able to do

1. Determine if a given name is in the table, the symbol table routines are concerned with saving and retrieving tokens.

- **insert(s, t)** : this function is to add a new name to the table.
- **Lookup(s)** : returns index of the entry for string s, or 0 if s is not found.

2. Access the information associated with a given name, and add new information for a given name.
3. Delete a name or group of names from the tables.

Attributes of Token

When more than one lexeme can match a pattern, the lexical analyzer must provide the subsequent compiler phases additional information about the particular lexeme that matched. For example, the pattern for token

number matches both 0 and 1, but it is extremely important for the code generator to know which lexeme was found in the source program. Thus, in many cases the lexical analyzer returns to the parser not only a token name, but an attribute value that describes the lexeme represented by the token; the token name influences parsing decisions, while the attribute value influences translation of tokens after the parse.

We shall assume that tokens have at most one associated attribute, although this attribute may have a structure that combines several pieces of information. The most important example is the token id, where we need to associate with the token a great deal of information. Normally, information about an identifier — e.g., its lexeme, its type, and the location at which it is first found (in case an error message about that identifier must be issued) — is kept in the symbol table. Thus, the appropriate attribute value for an identifier is a pointer to the symbol-table entry for that identifier.

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

E = M * C ** 2

are written below as a sequence of pairs.

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

<assign_op>

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

<mult_op>

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

<exp_op>

<number, integer value 2>

Note that in certain pairs, especially operators, punctuation, and keywords, there is no need for an attribute value. In this example, the token number has been given an integer-valued attribute. In practice, a typical compiler would instead store a character string representing the constant and use as an attribute value for number a pointer to that string.

Transition Diagrams

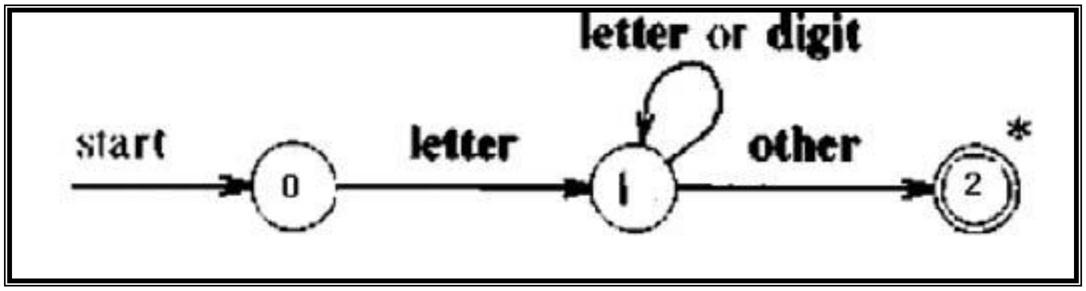
In the construction of lexical analyzer to convert patterns into stylized flowcharts, called "transition diagrams". ***Transition diagrams*** have a collection of **nodes or circles**, called ***states***. Each state represents a **condition** that could occur during the process of scanning the input looking for a lexeme that matches one of several patterns. We may think of a state as summarizing all we need to know about what characters we have seen between the ***lexemeBegin*** pointer and the ***forward*** pointer (as in the situation of Fig. 3.3).

Edges are directed from one state of the transition diagram to another. Each edge is ***labeled*** by a symbol or set of symbols(represent input string to move from one state to another). We shall assume that all our transition diagrams are ***deterministic***, meaning that there is never more than one edge out of a given state with a given symbol among its labels.

One state is designated the ***start state***, or ***initial state;*** it is indicated by an edge, labeled "**start**," entering from nowhere. The transition diagram always begins in the start state before any input symbols have been read.

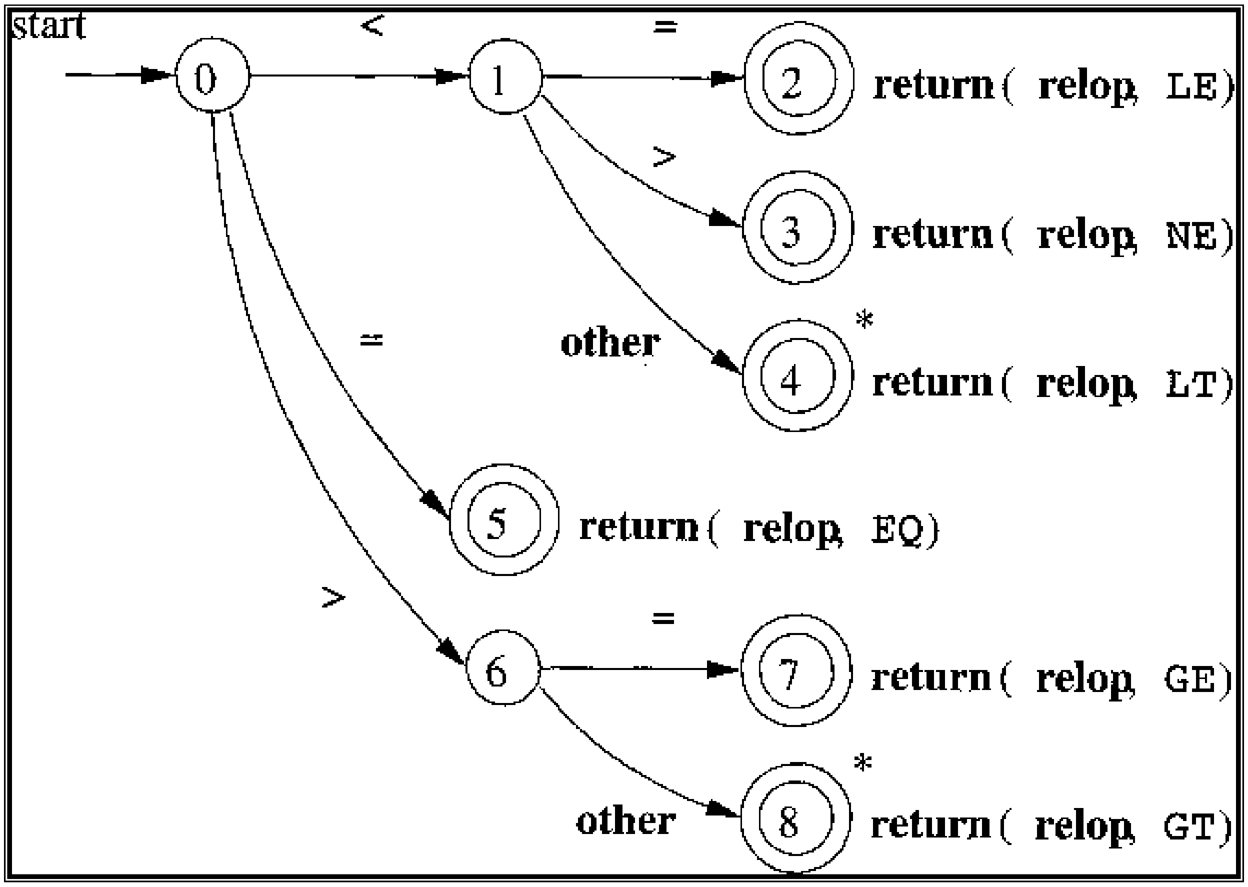
The ***final****(accepting)* ***state***. Representing by a double circle. These states indicate that a lexeme has been found.

Example1) Figure below represent the transition diagram of identifier.



Transition diagram for indentifier

Example2) Transition diagram that represent RELOP



Transition Diagram for RELOP

<u>Note)</u> The * in the transition diagram represent that the forward pointer must retract position.

Exercises)1. draw the transition diagram for the constant

3. draw the transition diagram for the "Begin" keyword.

Syntax Definition

The "context-free grammar," or "grammar" used to specify the syntax of a language. A grammar naturally describes the hierarchical structure of most programming language constructs. For example, an if-else statement in Java can have the form

if (expression) statement **else** statement

That is, an if-else statement is the concatenation of the keyword **if,** 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 structuring rule can be
expressed as

stmt → **if (** *expr* **)** *stmt* **else** *stmt*

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

Context Free Grammars

1. A set of terminal symbols, sometimes referred to as "tokens." The terminals are the elementary symbols of the language defined by the grammar.
2. A set of nonterminals, sometimes called "syntactic variables." Each nonterminal represents a set of strings of terminals. Nonterminals impose a hierarchical structure on the language that is key to syntax analysis and translation.

3. A set of productions, where each production consists of a nonterminal, called the head or left side of the production, an arrow, and a sequence of terminals and/or nonterminals, called the *body* or *right side* of the production. The intuitive intent of a production is to specify one of the written forms of a construct; if the head nonterminal represents a construct, then the body represents a written form of the construct.
4. A designation of one of the nonterminals as the *start* symbol.

We specify grammars by listing their productions, with the productions for the start symbol listed first. We assume that digits, signs such as < and <=, and boldface strings such as **while** are terminals. An italicized name is a nonterminal, and any nonitalicized name or symbol may be assumed to be a terminal. For notational convenience, productions with the same nonterminal as the head can have their bodies grouped, with the alternative bodies separated by the symbol |, which we read as "or."

Example: if we have the strings 9-5+2, 3-1, or 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:

$$\textit{list} \longrightarrow \textit{list} + \textit{digit} \quad \ldots(1)$$

$$\textit{list} \longrightarrow \textit{list} - \textit{digit} \quad \ldots(2)$$

$$\textit{list} \longrightarrow \textit{digit} \quad \ldots(3)$$

$$\textit{digit} \longrightarrow \mathbf{0} \mid \mathbf{1} \mid \mathbf{2} \mid \mathbf{3} \mid \mathbf{4} \mid \mathbf{5} \mid \mathbf{6} \mid \mathbf{7} \mid \mathbf{8} \mid \mathbf{9} \quad \ldots(4)$$

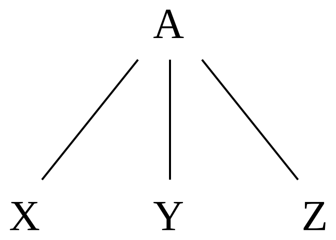
Derivations

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

1. In *leftmost* derivations, the leftmost nonterminal in each sentential is always chosen. If $\alpha \rightarrow \beta$ is a step in which the leftmost nonterminal in α is replaced, we write $\alpha \xrightarrow[lm]{} \beta$
2. In rightmost derivations, the right most nonterminal is always chosen; we write $\alpha \xrightarrow[rm]{} \beta$

Parse Trees

A parse tree pictorially shows how the start symbol of a grammar derives a
string in the language. If nonterminal *A* has a production *A* → *XYZ,* then a parse tree may have an interior node labeled *A* with three children labeled *X, Y,* and *Z*, from left to right:

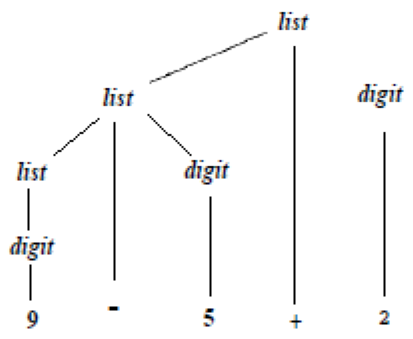


Formally, given a context-free grammar, a *parse tree* according to the grammar is a tree with the following properties:

1. The root is labeled by the start symbol.

2. Each leaf is labeled by a terminal 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, ... , X_n are the labels of the children of that node from left to right, then there must be a production $A \rightarrow X_1 X_2 \ldots X_n$. Here, X_1, X_2, ... , X_n each stand for a symbol that is either a terminal or a nonterminal. As a special case, if $A \rightarrow \varepsilon$ is a production, then a node labeled *A* may have a single child labeled ε.

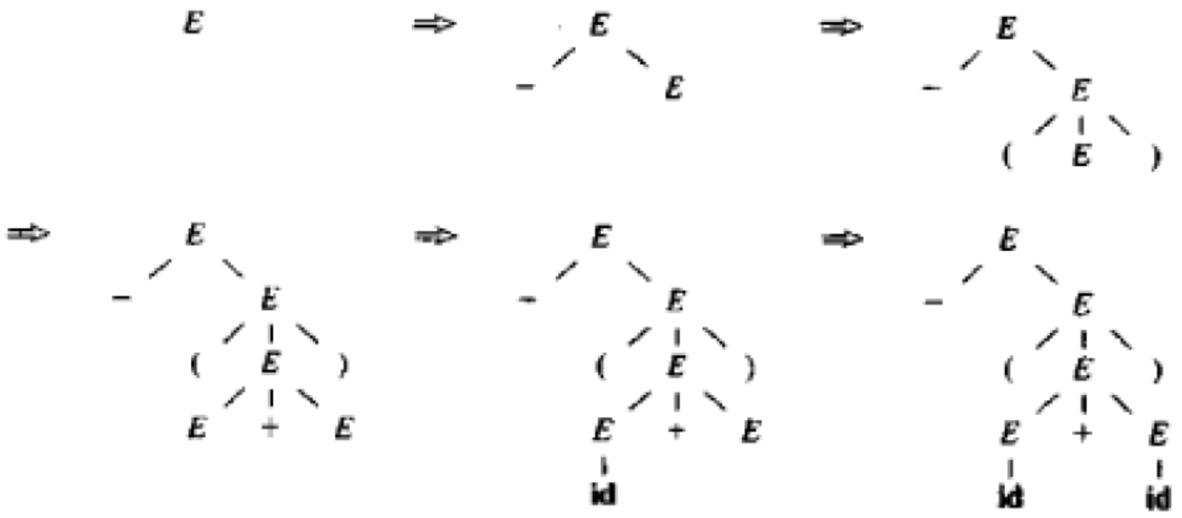
Example) the parse tree for 9-5+2



Parse Trees and Derivations

For example if we have the following grammar E→ E+E | -E | E*E | (E)|id, the parse tree for **- (id + id)** in Figure below

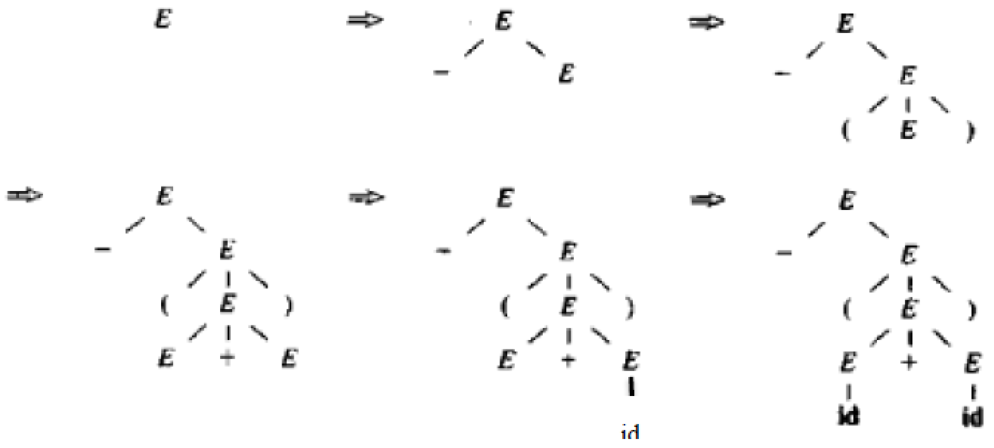
E → **-** *E* → **-** *(E)* → **-** *(E + E)* → **-** *(id +E)* → **-** *(id + id)*



This method for driving word called top-down method type left most derivation.

There is another way for top down derivation that is right most derivation as follows for the same Grammar above

$$E \rightarrow - E \rightarrow - (E) \rightarrow - (E + E) \rightarrow - (E + id) \rightarrow - (id + id)$$



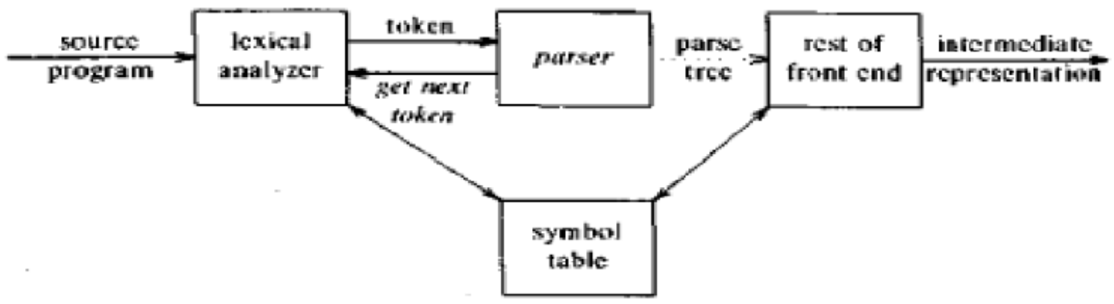
Syntax Analysis

Every programming language has precise rules that prescribe the syntactic structure of well-formed programs. The syntax of programming language constructs can be specified by Context – Free Grammars or BNF (Backus-Naur Form) notation, Grammars offer significant benefits for both language designers and compiler writers.

1. A grammar gives a precise, yet easy-to-understand, syntactic specification of a programming language.
2. From certain classes of grammars, we can construct automatically an efficient parser that determines the syntactic structure of source program. As a side benefit, the parser-construction process can reveal syntactic ambiguities and trouble spots that might have slipped through the initial design phase of a language.
3. The structure imparted to a language by a properly designed grammar is useful for translating source programs into correct object code and for detecting errors.
4. A grammar allows a language to be evolved or developed iteratively, by adding new constructs to perform new tasks. These new constructs can be integrated more easily into an implementation that follows the grammatical structure of the language.

The role of parser

The parser obtains a string of tokens from the lexical analyzer, as shown in Figure bellow, and verifies that the string of token names can be generated by the grammar for the source language. We expect the parser to report any syntax errors in an intelligible fashion and to recover from commonly occurring errors to continue processing the remainder of the program. Conceptually, for well-formed programs, the parser constructs a parse tree and passes it to the rest of the compiler for further processing.



The methods commonly used in compilers can be classified as being either ***top-down*** or ***bottom-up***. As implied by their names, top-down methods build parse trees from the top (root) to the bottom (leaves), while bottom-up methods start from the leaves and work their way up to the root. In either case, the input to the parser is scanned from left to right, one symbol at a time.

Problems of Grammar

1) Ambiguity

A grammar that produces more than one parse tree for some sentence is said to be ***ambiguous. An ambiguous grammar is one that produces more***

than one leftmost derivation or more than one rightmost derivation for the same sentence.

Since a string with more than one parse tree usually has more than one meaning, we need to design unambiguous grammars for compiling applications, or to use ambiguous grammars with additional rules to resolve the ambiguities.

The arithmetic expression grammar (E→ E+E | -E | E*E | (E)|id) permits two distinct leftmost derivations for the sentence **id + id * id** :-

$$
\begin{array}{lll}
E & \Rightarrow & E+E \\
  & \Rightarrow & \mathbf{id}+E \\
  & \Rightarrow & \mathbf{id}+E*E \\
  & \Rightarrow & \mathbf{id}+\mathbf{id}*E \\
  & \Rightarrow & \mathbf{id}+\mathbf{id}*\mathbf{id}
\end{array}
\qquad
\begin{array}{lll}
E & \Rightarrow & E*E \\
  & \Rightarrow & E+E*E \\
  & \Rightarrow & \mathbf{id}+E*E \\
  & \Rightarrow & \mathbf{id}+\mathbf{id}*E \\
  & \Rightarrow & \mathbf{id}+\mathbf{id}*\mathbf{id}
\end{array}
$$

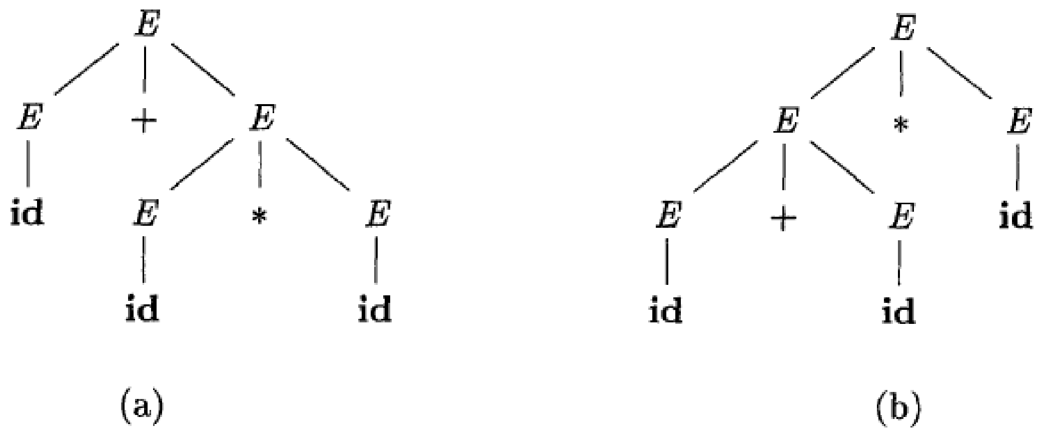


Figure 4.5: Two parse trees for **id+id*id**

For most parsers, it is desirable that the grammar be made unambiguous, for if it is not, we cannot uniquely determine which parse tree to select for a sentence. In other cases, it is convenient to use carefully chosen ambiguous grammars, together with *disambiguating rules* that "throw away" undesirable parse trees, leaving only one tree for each sentence.

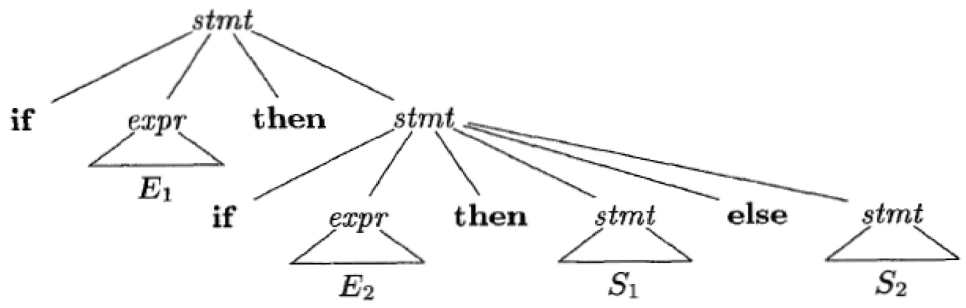
Eliminating Ambiguity

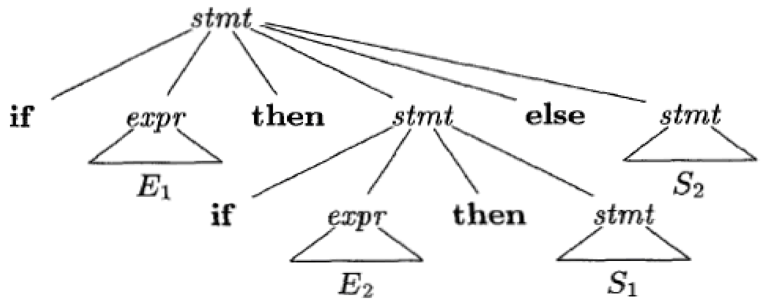
Sometimes an ambiguous grammar can be rewritten to eliminate the ambiguity. As an example, we shall eliminate the ambiguity from the following "dangling-else" grammar:

$$
\begin{array}{lcl}
stmt & \rightarrow & \textbf{if } expr \textbf{ then } stmt \\
 & | & \textbf{if } expr \textbf{ then } stmt \textbf{ else } stmt \\
 & | & \textbf{other}
\end{array}
$$

Here **"other"** stands for any other statement. This grammar is ambiguous since the string

if E_1 **then if** E_2 **then** S_1 **else** S_2





In all programming languages with conditional statements of this form, the first parse tree is preferred. The general rule is, "Match each else with the closest unmatched then". This disambiguating rule can theoretically be incorporated directly into a grammar, but in practice it is rarely built into the productions.

The unambiguous grammar of dangle if then else statement can be take the following form.

$$\begin{array}{rcl} stmt & \rightarrow & matched\_stmt \\ & | & open\_stmt \\ matched\_stmt & \rightarrow & \textbf{if } expr \textbf{ then } matched\_stmt \textbf{ else } matched\_stmt \\ & | & \textbf{other} \\ open\_stmt & \rightarrow & \textbf{if } expr \textbf{ then } stmt \\ & | & \textbf{if } expr \textbf{ then } matched\_stmt \textbf{ else } open\_stmt \end{array}$$

Unambiguous grammar for if-then-else statements

2) Elimination of Left – Recursive

A grammar is ***left recursive*** if it has a nonterminal *A* such that there is a derivation $A \xrightarrow{+} A\alpha$ for some string α. Top-down parsing methods cannot handle left-recursive grammars, so a transformation is needed to eliminate left recursion. If we have the production $A \rightarrow A\alpha \mid \beta$ then can be replaced by the non recursive production

$$A \rightarrow \beta A'$$
$$A' \rightarrow \alpha A' \mid \varepsilon$$

without changing the strings derivable from *A*.

Example:

$$\begin{array}{rcl} E & \rightarrow & E + T \mid T \\ T & \rightarrow & T * F \mid F \\ F & \rightarrow & (E) \mid \textbf{id} \end{array} \quad \Longrightarrow \quad \begin{array}{l} E \rightarrow T\, E' \\ E' \rightarrow +\, T\, E' \\ T \rightarrow F\, T' \\ T' \rightarrow *\, F\, T' \\ F \rightarrow (E) \mid \textbf{id} \end{array}$$

Immediate left recursion can be eliminated by the following technique, which works for any number of A-productions. First, group the productions as

$$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \cdots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n$$

where no $\boldsymbol{\beta_i}$ begins with an *A*. Then, replace the A-productions by

$$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \cdots \mid \beta_n A'$$
$$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \cdots \mid \alpha_m A' \mid \epsilon$$

The nonterminal *A* generates the same strings as before but is no longer left recursive. This procedure eliminates all left recursion from the *A* and *A'* productions (provided no is e), but it does not eliminate left recursion involving derivations of two or more steps. For example, consider the grammar

$$S \rightarrow A\ a \mid b$$
$$A \rightarrow A\ c \mid S\ d \mid \epsilon$$

The nonterminal S is left recursive because S→ Aa → Sda, but it is not immediately left recursive.

3) Left Factoring:-

Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive, or top-down, parsing. When the choice between two alternative *A*-productions is not clear, we may be able to rewrite the productions to defer the decision until enough of the input has been seen that we can make the right choice.

stmt → **if** *expr* **then** *stmt* **else** *stmt*

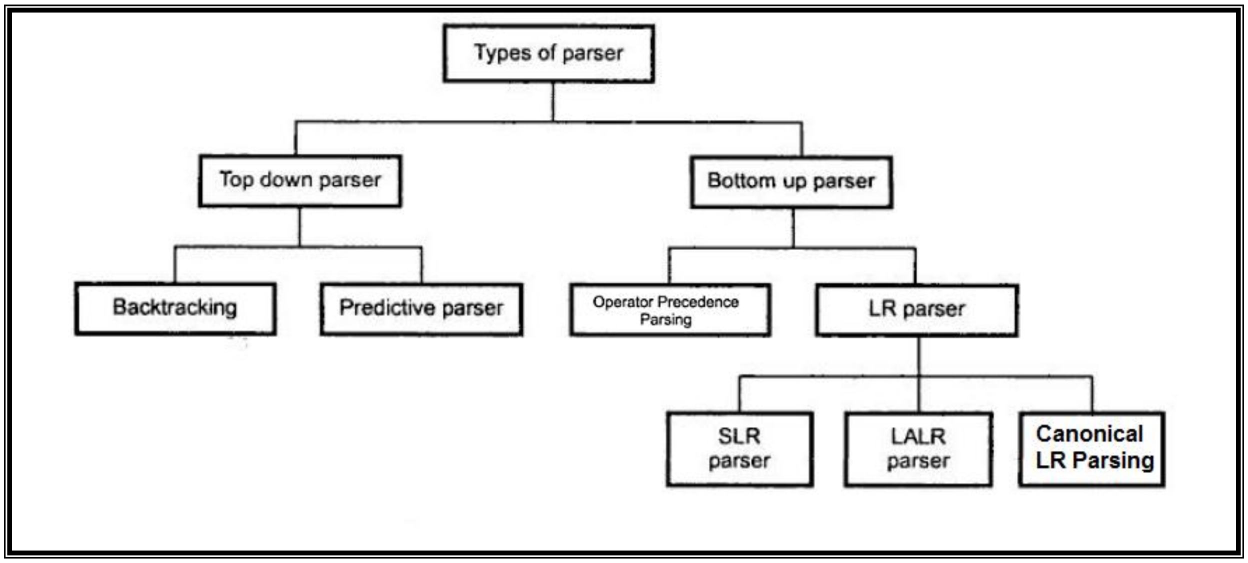
stmt → **if** *expr* **then** *stmt*

on seeing the input token if, we cannot immediately tell which production to choose to expand stmt. In general, if A → α β1| α β2 are two *A*- productions, and the input begins with a non-empty string derived from α ,we do not know whether to expand A to α β1 or to α β2. However, we may defer the decision by expanding A to αA', then after seeing the input derived from α, we expand A' to β1 or to β2. that is, left factored, the original production become:

A →αA'

$A' \rightarrow \beta1 | \beta2$

Parsing



Top Down Parsing

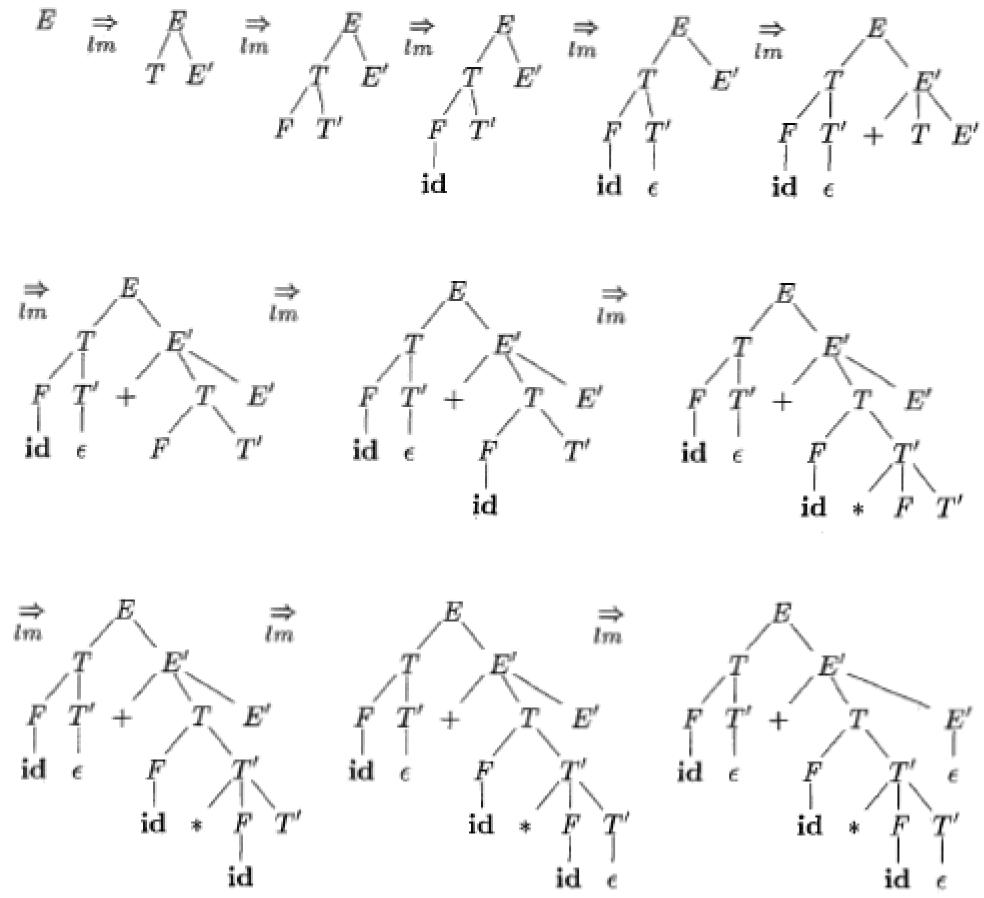
Top-down parsing can be viewed as the problem of constructing a parse tree for the input string, it can be viewed as finding a leftmost derivation for an input string. The top-down construction of a parse tree is done by starting from the root and creating the nodes of the parse tree in preorder. For example consider the grammar

Example : The sequence of parse trees in Figure below for the input **id + id*id** is a top-down parse according to the grammar:

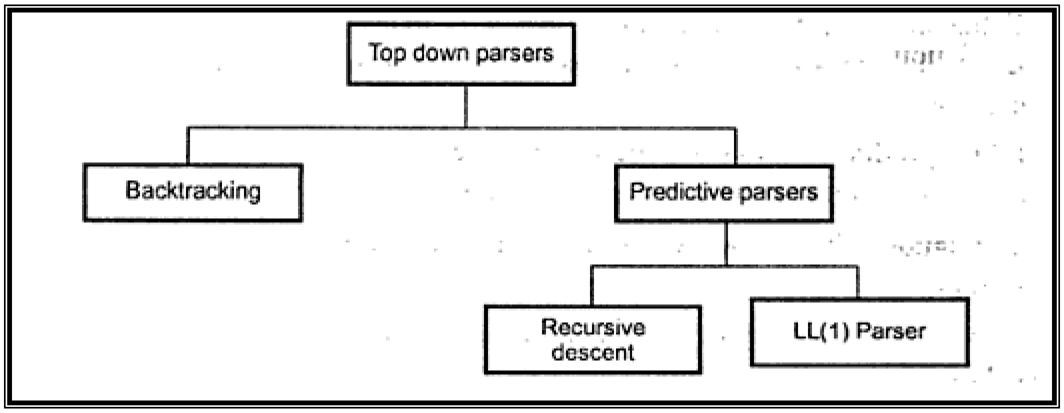
$$\begin{array}{lcl} E & \rightarrow & T\ E' \\ E' & \rightarrow & +\ T\ E'\ |\ \epsilon \\ T & \rightarrow & F\ T' \\ T' & \rightarrow & *\ F\ T'\ |\ \epsilon \\ F & \rightarrow & (\ E\ )\ |\ \mathbf{id} \end{array}$$

At each step of a top-down parse, the key problem is that of determining the production to be applied for a nonterminal, say A. Once an A-production is chosen, the rest of the parsing process consists of "matching" the terminal symbols in the production body with the input string.

A general form of top-down parsing, called recursive-descent parsing, which may require backtracking to find the correct A- production to be applied.



The Top down parsing divided into two parts:-



A backtracking parser will try different production rules to find the match for the input string by backtracking each time. The backtracking is powerful than predictive parsing. But this technique is slower and it requires exponential time in general. Hence backtracking is not preferred for practical compilers.

As the name suggests the predictive parser tries to predict the next construction using one or more lookahead symbols from input string. There are two types of predictive parsers:-

1. Recursive Descent
2. LL(1) Parser

FIRST and FOLLOW

The construction of both top-down and bottom-up parsers is aided by two functions, **FIRST** and **FOLLOW,** associated with a grammar G. During top-down parsing, **FIRST** and **FOLLOW** allow us to choose which production to apply, based on the next input symbol.

FIRST

Define ***FIRST(α),*** as a set of terminal symbols that are first symbols appear at right side in derivation *α*

1. If X is a terminal, then FIRST(X) = $\{X\}$.
2. If $X \rightarrow \varepsilon$ is a production, then add ε to FIRST(X).
3. if X a nonterminal and $X \rightarrow Y_l Y_2 \ldots Y_k$ is a production for some $k \geq 1$, then place α in FIRST(X) if for some ***i,*** a is in FIRST(Y_i), and r is in all of FIRST(Y_1),…, FIRST(Y_{i-1}); that is, $Y_1 \ldots Y_{i-1} \xrightarrow{*} \varepsilon$. If ε is in FIRST(Y_j) for all j = 1,2, . . . , k, then add ε to FIRST(X). For example, everything in FIRST(Y_1) is surely in FIRST(X). If Y_1 does not derive ε, then we add nothing more to FIRST(X), but if $Y_l \xrightarrow{*} \varepsilon$, then we add F1RST(Y2), and so on.

<u>FOLLOW</u>

Define FOLLOW(A) as the set of terminal Symbols that appear immediately to the right of A in other words:-

FOLLOW(A) = {a | S $\xrightarrow{*}$ α Aa β , where α and β are some grammar symbols may be terminal or nonterminal }

To compute FOLLOW(A) for all nonterminals A, apply the following rules until nothing can be added to any FOLLOW set.

1. Place \$ in FOLLOW(S), where S is the start symbol, and \$ is the input right endmarker.
2. If there is a production $A \rightarrow \alpha B \beta$, then everything in FIRST(β) except ε is in FOLLOW(B).
3. If there is a production $A \rightarrow \alpha B$, or a production $A \rightarrow \alpha B \beta$, where FIRST($\beta$) contains ε, then everything in FOLLOW(A) is in FOLLOW (B).

Example 1)

Compute the FIRST and FOLLOW for the following grammar:-

$$
\begin{aligned}
E &\rightarrow TE' \\
E' &\rightarrow +TE' \mid \varepsilon \\
T &\rightarrow FT' \\
T' &\rightarrow *FT' \mid \varepsilon \\
F &\rightarrow (E) \mid id
\end{aligned}
$$

Solution :

AS E → TE′ is a rule in which the first symbol at RHS is T. Now T→ FT′ in which the first symbol at RHS is F and there is a rule for F as F → (E)|id.

FIRST(E)=FIRST(T)=FIRST(F)

As F → (E)

F → **id**

Hence **FIRST(E) = FIRST(T) = FIRST(F) ={ (,id }**

FIRST(E′)=(+, ε)

As E′ → +TE′

E′ → ε by referring computation rule 2

The first symbol appearing at RHS of production rule for E′ is added in the FIRST function.

FIRST(T′)=(*, ε)

As T′→* TE′

E′→ ε

The first terminal symbol appearing at RHS of production rule for T′ is added in the FIRST function. Now we will compute FOLLOW function.

FOLLOW(E) –

i) As there is a rule F→ (E) the symbol ')' appears immediately to the right of E. Hence ')' will be in FOLLOW (E).

ii) The computation rule is A → αBβ we can map this rule with F → (E) then A=F, α = (, B=E, β =).

FOLLOW(B) = FIRST(β) = FIRST( )) = {) }
FOLLOW(E) = {)}

Since E is a start symbol,add $ to follow of E.

Hence **FOLLOW(E) = {),$}**

FOLLOW(E') –

i) $E \rightarrow TE'$ the computational rule is $A \rightarrow \alpha B\beta$.

$\therefore$ $A = E'$, $\alpha = T$, $B = E'$, $\beta = \varepsilon$ then by computational rule 3 everything in FOLLOW(A) is in FOLLOW(B) i.e. everything in FOLLOW(E) is in FOLLOW(E')

$\therefore$ **FOLLOW(E') = {), $ }**

ii) $E' \rightarrow +TE'$ the computational rule is $A \rightarrow \alpha B\beta$.

$\therefore$ $A = E'$, $\alpha = +T$, $B = E'$, $\beta = \varepsilon$ then by computational rule 3 everything in FOLLOW(A) is in FOLLOW(B) i.e. everything in FOLLOW(E') is in FOLLOW(E')

FOLLOW(E') = {),$ }

We can observe in the given grammar that) is really following E.

FOLLOW(T) –

We have to observe two rules

$$E \rightarrow TE'$$
$$E' \rightarrow +TE'$$

i) Consider

$E \rightarrow TE'$ we will map it with $A \rightarrow \alpha\ B\ \beta$

$A = E$, $\alpha = \varepsilon$, $B = T$, $\beta = E'$ by computational rule 2 FOLLOW(B)={FIRST (β) – ε}.

That is FOLLOW(T)
= {FIRST(E') –ε}

$$= \{\{+,\varepsilon\} - \varepsilon\}$$
$$= \{+\}$$

ii) Consider $E' \rightarrow +TE'$ we will map it with $A \rightarrow \alpha B\beta$

$A = E'$, $\alpha = +$, $B = T$, $\beta = E'$ by computational rule 3 FOLLOW(A)=FOLLOW(B) i.e. FOLLOW(E')=FOLLOW(T)

FOLLOW(T) = {),$}

Finally **FOLLOW(T) = {+} ∪ {),$}**

= {+ ,), $}

We can observe in the given grammar that + and) are really following T.

FOLLOW(T') –

$T \rightarrow FT'$

We will map this rule with $A \rightarrow \alpha B\beta$ then $A = T$, $\alpha = F$, $B=T'$, $\beta=\varepsilon$ then FOLLOW(T')=FOLLOW(T) = {+,),$}

$T \rightarrow *FT'$

We will map this rule with $A \rightarrow \alpha B\beta$ then $A=T$, $\alpha = *F$, $B=T'$, $\beta=\varepsilon'$ then FOLLOW(T') = FOLLOW(T) = {+,),$}

Hence **FOLLOW(T') = {+,),$}**

FOLLOW(F) –

Consider $T \rightarrow FT'$ or $T' \rightarrow *FT'$ then by computational rule 2,

$T \rightarrow FT'$	$T' \rightarrow *FT'$
$A \rightarrow \alpha B\beta$	$A \rightarrow \alpha B\beta$
$A=T, \alpha=\varepsilon, B=F, \beta=T'$	$A=T', \alpha=*, B=F, \beta=T'$
FOLLOW(B)={FIRST (β) – ε}	FOLLOW(B)= {FIRST(β) – ε}
FOLLOW(F)={FIRST(T') – ε}	FOLLOW(F)= {FIRST(T') – ε}
FOLLOW(F)={*}	FOLLOW(F)={*}

Consider $T' \rightarrow *FT'$ by computational rule 3

$T' \rightarrow *FT'$

$A \rightarrow \alpha B\beta$

$A = T', \alpha = *, B = F, \beta = T'$

FOLLOW(A) = FOLLOW (B)

FOLLOW(T') = FOLLOW(F)

But FOLLOW (T) = { +,), $}

Hence FOLLOW(F) = {+,), $}

Finally FOLLOW(F) = {*} ∪ {+,),$}

FOLLOW(F) = {+,*,),$}

Symbol	FIRST	FOLLOW
E	**{(, id}**	**{$,)}**
E'	**{+, ε }**	**{$,)}**
T	**{(, id }**	**{+,$,)}**
T'	**{*, ε }**	**{+,$,)}**
F	**{(, id}**	**{+,*,$,)}**

Example2)

Compute the FIRST and FOLLOW for the following grammar

S→ABCDE
A→ a|ε
B→ b|ε
C→ c
D→ d|ε
E→ e|ε

Symbol	FIRST	FOLLOW
S	{a,b,c}	{$}
A	{a,ε}	{b,c}
B	{b,ε}	{c}
C	{c}	{d,e,$}
D	{d,ε}	{e,$}
E	{e,ε}	{$}

Example3)

Compute the FIRST and FOLLOW for the following grammar

S→ aABC
A→ a|bb
B→ a|ε
C→ b|ε

Symbol	FIRST	FOLLOW
S	{a}	{$}
A	{a,b}	{a,b,$}
B	{a,ε}	{b,$}
C	{b,ε}	{$}

Example4)

Compute the FIRST and FOLLOW for the following grammar

S→ Bb|Cd
B→ aB| ε
C→ cC| ε

Symbol	FIRST	FOLLOW

S	**{a,b,c,d}**	**{$}**
B	**{a,ε}**	**{b}**
C	**{c,ε}**	**{d}**

<u>Example5)</u>

Compute the FIRST and FOLLOW for the following grammar

S→ ABC|CbB|Ba
A→ da| BC
B→ g| ε
C→ h| ε

Symbol	FIRST	FOLLOW
S	**{d,g,h,b,a,ε}**	**{$}**
A	**{d,g,h, ε }**	**{$,g,h}**
B	**{g,ε}**	**{$,a,h,g}**
C	**{h,ε}**	**{$,b,h,g}**

<u>Example6)</u>

Compute the FIRST and FOLLOW for the following grammar

S→ aABb
A→ c| ε
B→ d| ε

Symbol	FIRST	FOLLOW
S	**{a}**	**{$}**
A	**{c,ε }**	**{d,b}**
B	**{d,ε}**	**{b}**

<u>Example7)</u>

Compute the FIRST and FOLLOW for the following grammar

S→ aBDh
B→ cC
C→ bC| ε
D→ EF
E→ g|ε
F→ f| ε

Symbol	FIRST	FOLLOW
S	{a}	{$}
B	{c}	{g,f,h}
C	{b.ε}	{g,f,h}
D	{g,f,ε}	{h}
E	{g,ε}	{f,h}
F	{f,ε}	{h}

Example8)

Compute the FIRST and FOLLOW for the following grammar

S → aSb │X
X → cXb │b
X → bXZ
Z → n

Symbol	FIRST	FOLLOW
S	{a,c,b}	{$,b}
X	{c,b}	{$,b,n}
Z	{n}	{$,b,n}

Example9)

Compute the first and follow for the following grammar

S→ bXY
X→ b|c
Y→ b|ε

Symbol	FIRST	FOLLOW
S	{b}	{$}
X	{b,c}	{$,b}
Y	{b,ε}	{$}

Example10)

Compute the first and follow for the following grammar

S → ABb │bc
A → ε │abAB
B → bc │cBS

Symbol	FIRST	FOLLOW
S	**{a,b,c }**	**{$,b,c,a}**
A	**{a,ε}**	**{b,c}**
B	**{b,c}**	**{b,c,a}**

<u>Example11)</u>

Compute the first and follow for the following grammar

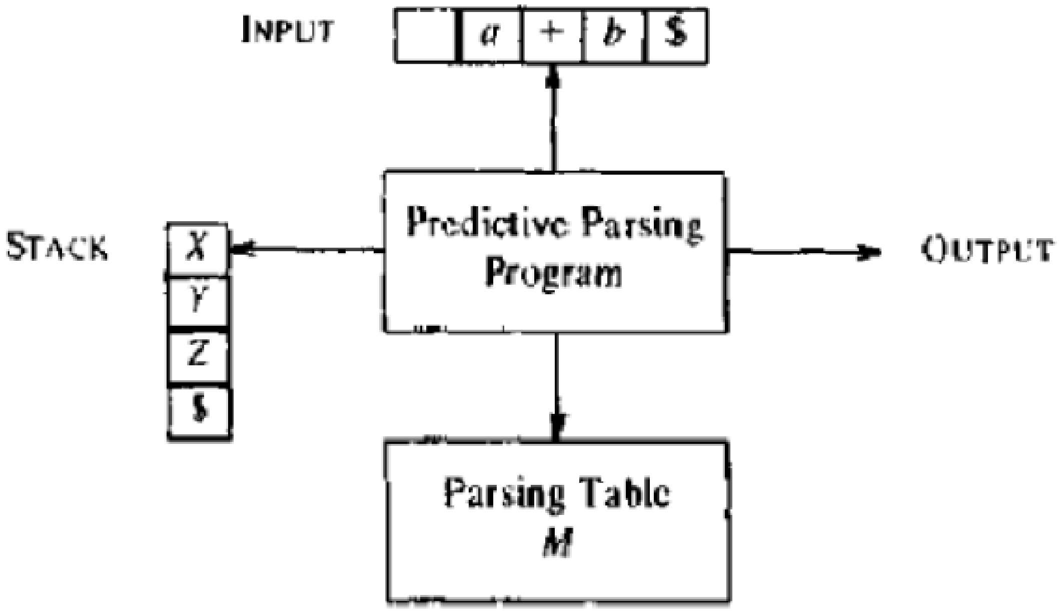
X → ABC │ nX
A → bA │ bb │ ε
B → bA │CA
C → ccC │CA │ cc

Symbol	FIRST	FOLLOW
X	**{b,c,n}**	**{$}**
A	**{b,ε}**	**{b,c,$}**
B	**{b,c}**	**{c}**
C	**{c}**	**{$,b}**

<u>Predictive Parsing LL(1).</u>

This top down parsing algorithm is of non recursive type. In this type of parsing a table is built. For LL(1)- the first L means the input is scanned from left to right, the second L means it uses leftmost derivation for input string. And the number 1 in the input symbol means it uses only one input symbol (lookahead) to predict the parsing process.

The predictive parsing has an input, a stack, a parsing table, and an output. The input contains the string to be parsed, followed by $, the right endmarker. The stack contains a sequence of grammar symbols, preceded by \$, the bottom – of – stack marker. Initially, the stack contains the start symbol of the grammar preceded by \$. The parsing table is a two dimensional array M[*A*, *a*], where *A* is a nonterminal, and *a* is a terminal or the symbol \$.



The parser is controlled by a program that behaves as follows. The program determines X, the symbol on top of the stack, and a, the current input symbol. These two symbols determine the action of the parser. There are three possibilities.

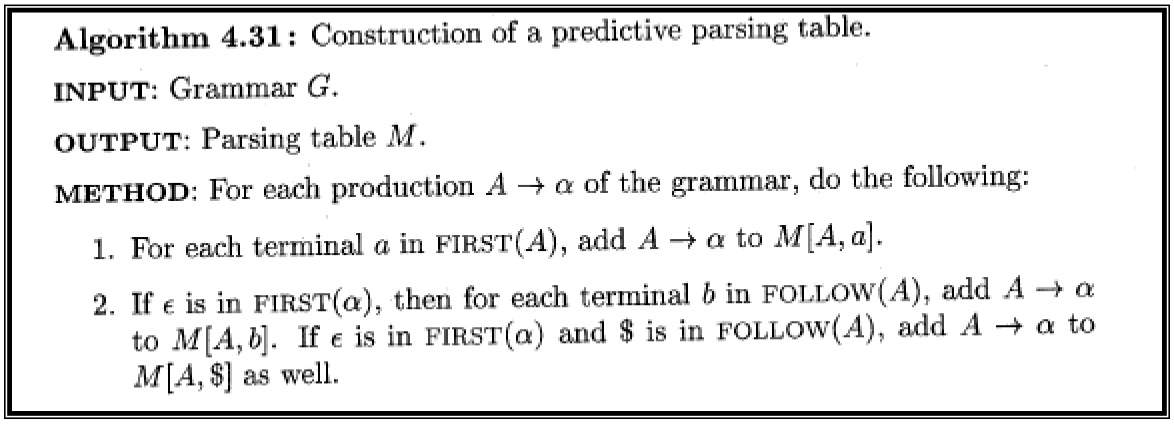
1. If X =*a*=\$, the parser halts and announces successful completion of parsing.
2. If X = *a* ≠\$, the parser pops X off the stack and advances the input pointer to the next input symbol.
3. If X is a nonterminal, the program consults entry M[*X* ,*a*] of the parsing table M. this entry will be either an *X*- production of the

grammar or an error entry. If M[*X*, *a*] = {X→ *UVW*}, the parser replaces X on top of the stack by *UVW*(with U on top). As output, the grammar does the semantic action associated with this production, which, for the time being, we shall assume is just printing the production used. If M[*X* ,*a*] =error, the parser calls an error recovery routine.

Example) parse the input id*id+id in the grammar.

$$\begin{array}{lcl} E & \rightarrow & T\ E' \\ E' & \rightarrow & +\ T\ E' \mid \epsilon \\ T & \rightarrow & F\ T' \\ T' & \rightarrow & *\ F\ T' \mid \epsilon \\ F & \rightarrow & (\ E\ ) \mid \mathbf{id} \end{array}$$

Symbol	FIRST	FOLLOW
E	**{(, id}**	**{$,)}**
E'	**{+, ε }**	**{$,)}**
T	**{(, id }**	**{+,$,)}**
T'	**{*, ε }**	**{+,$,)}**
F	**{(, id}**	**{+,*,$,)}**



Algorithm 4.31: Construction of a predictive parsing table.

INPUT: Grammar G.

OUTPUT: Parsing table M.

METHOD: For each production $A \rightarrow \alpha$ of the grammar, do the following:

1. For each terminal a in FIRST(A), add $A \rightarrow \alpha$ to $M[A, a]$.
2. If ϵ is in FIRST(α), then for each terminal b in FOLLOW(A), add $A \rightarrow \alpha$ to $M[A, b]$. If ϵ is in FIRST(α) and $ is in FOLLOW(A), add $A \rightarrow \alpha$ to $M[A, \$]$ as well.

The parse table M for the grammar:

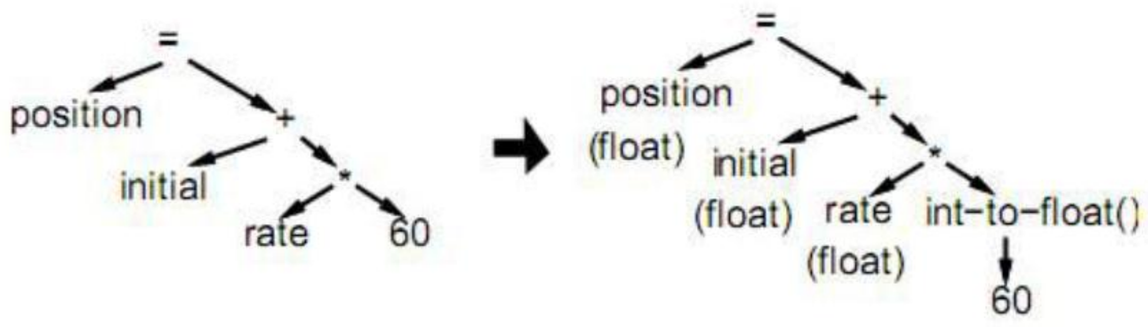
NONTER-MINAL	INPUT SYMBOL id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \epsilon$	$E' \to \epsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to$ id			$F \to (E)$		

The moves made by predictive parser on input id+id*id

STACK	INPUT	OUTPUT
$E	id + id * id$	
$E'T	id + id * id$	$E \to TE'$
$E'T'F	id + id * id$	$T \to FT'$
$E'T'id	id + id * id$	$F \to$ id
$E'T'	+ id * id$	
$E'	+ id * id$	$T' \to \epsilon$
$E'T +	+ id * id$	$E' \to +TE'$
$E'T	id * id$	
$E'T'F	id * id$	$T \to FT'$
$E'T'id	id * id$	$F \to$ id
$E'T'	* id$	
$E'T'F*	* id$	$T' \to *FT'$
$E'T'F	id$	
$E'T'id	id$	$F \to$ id
$E'T'	$	
$E'	$	$T' \to \epsilon$
$	$	$E' \to \epsilon$

Semantic Analysis

- The semantic analysis phase checks the source program for semantic errors and gathers type information for the subsequent code-generation phase.
- It uses the parse tree to identify the operators and operands of expressions and statements, see the figure below
- An important component is type checking.
- Here the compiler checks that each operator has operands that are permitted by the source language specification.
- Static semantic checks are performed at compile time
    - Type checking
    - Every variable is declared before used
    - Identifiers are used in appropriate contexts
    - Check subroutine call arguments
- Dynamic semantic check are performed at run time, and the compiler produces code that performs these checks
    - Array subscript values are within bounds
    - Arithmetic errors, e.g. division by zero
    - A variable is used but hasn't been initialized
    - When a check fails at run time, an exception is raised



Type checking

A type checker verifies that the type of a construct matches that expected by its context. For example, the –in arithmetic operator mod in Pascal requires integer operands, so a type checker must verify that the operands of mod have type integer.

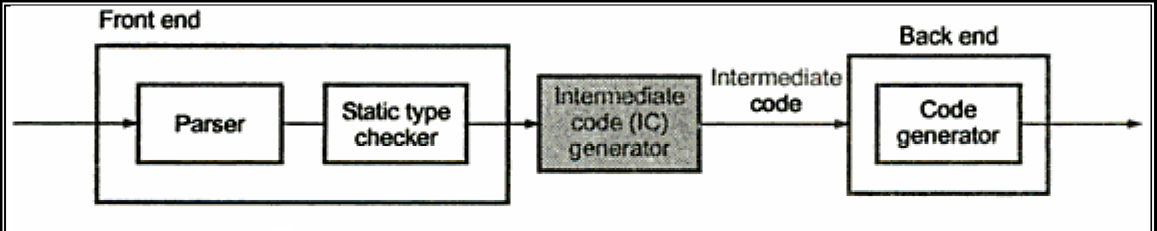
Intermediate Code Generation

Introduction

In the analysis – synthesis model of a compiler, the front end translates a source program into an intermediate representation from which the back end generates target code. Details of the target language are confined to the back end, as far as possible. Although a source program can be translated directly into the target language. Some benefits of using a machine – independent intermediate form are:-

1. Retargeting is facilitated: a compiler for a different machine can be created by attaching a back end for the new machine to an existing front end.
2. A machine – independent code optimizer can be applied to the intermediate representation.

The role of intermediate code generator in compiler is depicted as follow:-



Intermediate code generator in compiler

The tasks of intermediate representation is

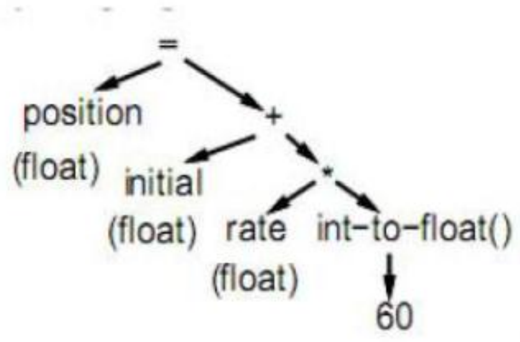
Translate from abstract-syntax trees to intermediate codes.

Generating a low-level intermediate representation with two properties:

- It should be easy to produce
- It should be easy to translate into the target machine

One of the popular intermediate code is ***three-address code***. A three address code:

- Each statement contains at most 3 operands; in addition to ": =", i.e., assignment, at most one operator.
- An" easy" and "universal" format that can be translated into most assembly languages.



```
temp1 := int-to-float(60)
temp2 := rate * temp1
temp3 := initial + temp2
position := temp3
```

Some of the basic operations which in the source program, to change in the Assembly language:

operations	H.L.L	Assembly language
Math. OP	+ , - , * , /	Add, sub, mult, div
Boolean. OP	&, | , ~	And, or, not
Assignment	=	mov
Jump	goto	JP, JN, JC
conditional	If, then	CMP
Loop instruction	For, do, repeat until, while do	These most have and I.C.G before change it to assembly language.

The operation which change H.L.L to assembly language, is called the intermediate code generation and there is the division operation come with it, which mean every statement have a single operation.

<u>Ex</u>

$$X = \underbrace{\underbrace{A + \underbrace{\underbrace{B * C}_{T1} / D}_{T2}}_{T4} - \underbrace{Y * N}_{T3}}_{T5}$$

T1 = B * C
T2 = T1 / D
T3 = Y * N
T4 = A + T2
T5 = T4 - T3

<u>Ex2</u>

$$Y = \underbrace{\underbrace{\underbrace{\cos \underbrace{(A * B)}_{T1}}_{T2} + \underbrace{C / N}_{T4}}_{T5} - \underbrace{Y * P}_{T3}}_{T6}$$

T1 = A * B
T2 = cos T1
T3 = Y * P
T4 = C / N
T5 = T2 + T4
T6 = T5 – T3

There are three representation for Three Address Code (Quadruples, Triples, Indirect triples)

1. Triple form

Ex: X = A + B * C / (- N)

	OP	Arg1	Arg2
(0)	*	B	C
(1)	-	N	__
(2)	/	(0)	(1)
(3)	+	A	(2)
	=	X	(3)

Ex: Y = A + C * X / B [i]

	OP	Arg1	Arg2
(0)	*	C	X
(1)	=[ ]	B	i
(2)	/	(0)	(1)
(3)	+	A	(2)
	=	Y	(3)

Ex: X[i] = N * C / Y[i]

	OP	Arg1	Arg2
(0)	*	N	C
(1)	=[ ]	Y	i
(2)	/	(0)	(1)
(3)	[ ]=	X	i
	=	(3)	(2)

Ex: X = A + B * (c / d) - y

	OP	Arg1	Arg2
(0)	/	c	d
(1)	*	B	(0)
(2)	+	A	(1)
(3)	-	(2)	y
	=	X	(3)

Ex: A = C * X [i,j]

	OP	Arg1	Arg2
(0)	=[ ]	X	P
(1)	*	C	(0)
	=	A	(1)

2. Quadruple form

Ex: X = A * C / N + P

OP	Arg1	Arg2	Result
*	A	C	t1
/	t1	N	t2
+	t2	P	t3
=	t3		X

Ex: A = N[i] * C / N

OP	Arg1	Arg2	Result
=[ ]	N	i	t1
*	t1	C	t2
/	t2	N	t3
=	t3		A

Ex: A = C * y / X[i,j]

OP	Arg1	Arg2	Result
*	C	y	t1
=[ ]	X	P	t2
/	t1	t2	t3
=	t3		A

Ex: X = A + B * (c / d) – y

OP	Arg1	Arg2	Result
/	c	d	t1
*	B	t1	t2
+	A	t2	t3
-	t3	y	t4
=	t4		X

Ex: X[i] = a * c + y[i] – n[j] / V

OP	Arg1	Arg2	Result
*	a	c	t1
=[ ]	n	j	t2
/	t2	V	t3
=[ ]	y	i	t4
+	t1	t4	t5
-	t5	t3	t6
[ ]=	X	i	t7
=	t6		t7

Code Optimization

Compilers should produce target code that is as good as can be written by hand. The code produced by straightforward compiling algorithms can often be made to run faster or take less space, or both. This improvement is achieved by program transformations that are traditionally called ***Optimizations***.

Function- Preserving Transformations

There are a number of ways in which a compiler can improve a program without changing the function it computes. Various function preserving transformations are,

1. ***Common subexpression elimination.***
2. ***Copy propagation.***
3. ***Dead – code elimination.***
4. ***Constant folding.***

1. Common subexpression elimination.

An expression E is called a common subexpression if it was previously computed, and the values of variables in E have not changed since the previous computation.

In common subexpression elimination we have to avoid re-computing of the expression if it's value is already been computed.

```
Ex1: X = A + C * N – M
          Y = B + C * N * e
          Sol: Q = C * N
          X = A + Q – M
```

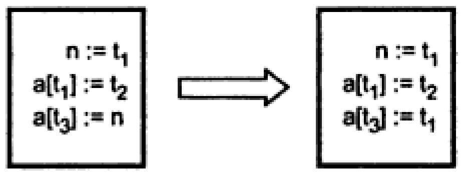
```
Y = B + Q * e
```

Before optimize	after optimize
t6 = 4 * i	t6 = 4 * i
X = a [t6]	X = a [t6]
t7 = 4 * i	t8 = 4 * j
t8 = 4 * j	t9 = a [t8]
t9 = a [t8]	a [t6] = t9
a [t7] = t9	a[t8] = X
t10 = 4 * j	
a[t10] = X	

<u>Note:</u> The value of the variable which are optimize will not be change.

2. <u>Copy Propagation</u>

Copy propagation means use of one variable instead of another. For instance: if "x := y" is a statement which is called copy statement, then copy propagation is a kind of transformation in which use y for x wherever possible after copy statement x:= y.



$n := t_1$
$a[t_1] := t_2$
$a[t_3] := n$

$n := t_1$
$a[t_1] := t_2$
$a[t_3] := t_1$

Copy propagation

Although this is not an improvement but it will definitely help in eliminating assignment to n.

3. <u>Dead Code Elimination</u>

A variable is said to be live in the program if its value can be used subsequently, otherwise it is dead at that point.

The dead code is basically an useless code. An optimization can be performed by eliminating such a dead code.

For example

```
i= j;
...
...
x=i+10;
...
```

The dead- code elimination can be done by eliminating the assignment statement i =j. This assignment statement is called dead assignment.

Another example

```
i= 0;
if (i==1)
{
   a= x+5;
}
```

Here if statement is a dead – code as this condition never gets satisfied. Hence if we remove if statement, optimization can be done. One advantage of copy propagation is that it often turns the copy statement into dead – code. Hence after copy propagation if dead –code elimination is done, then the useless statements can be removed.

Loop Optimization

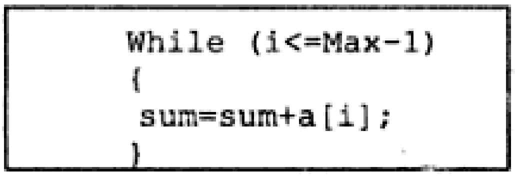
The code optimization can be significantly done in loops of the program. Specially inner loops is a place where program spend large amount of time. Hence if number of instructions are less in inner loop the running time of the program will get decreased to a large extent (i.e. The running time of a program may be improved if we decrease the number of instructions in an inner loop). The loop optimization is carried out by following three methods:-

1. ***Code Motion***
2. ***Induction Variable***
3. ***Reduction in Strength***

1. <u>Code Motion</u>

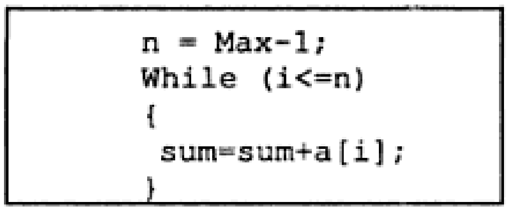
Code motion is a technique which moves the code outside the loop. Hence is the name. If there lies some expression in the loop whose result remains unchanged even after executing the loop for several times, then such an expression should be placed just before the loop (i.e. outside the loop). Here before the loop means at the entry of the loop

For example



```
While (i<=Max-1)
{
 sum=sum+a[i];
}
```

The above code can be optimized by removing the computation of Max -1 outside the loop. Hence the optimized code can be

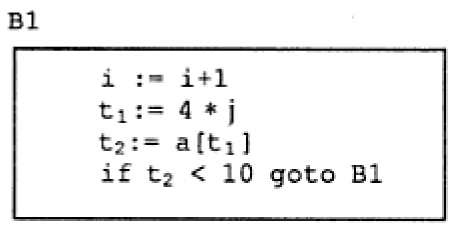


```
n = Max-1;
While (i<=n)
{
 sum=sum+a[i];
}
```

2. <u>Induction Variable</u>

A variable x is called an induction variable of loop L if the value of variable gets changed every time. It is either decremented or incremented by some constant

For example, consider the block of code given below



```
B1
i := i+1
t1:= 4 * j
t2:= a[t1]
if t2 < 10 goto B1
```

In above code the values of I and t_1 are in locked state. That is, when value of i gets incremented by 1 then t_1 gets incremented by 4. Hence I and t t_1 are induction variable.

When there are two or more induction variables in a loop, it may be possible to get rid of all but one.

3. Reduction in Strength

The strength of certain operators is higher than others.

For example: Strength of * is higher than +. In strength reduction technique the higher strength operators can be replaced by lower strength operators.

For example

```
For (i =0 ; i<=50; i++)
{
     ...
     count =i*7;
     ...
}
```

Here we get values of count as 7, 14, 21 and so on up to 50.

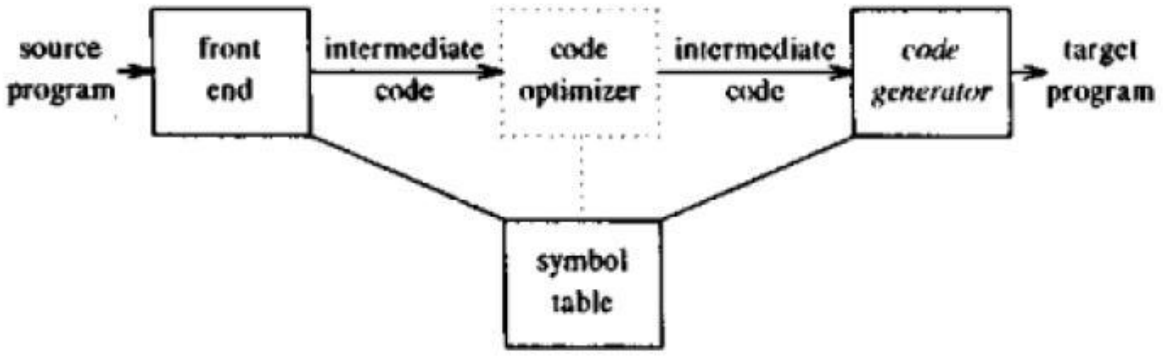
This code can be replaced by using strength reduction as follows.

```
temp = 7;
For (i =0 ; i<=50; i++)
{
     ...
     count =temp;
     temp = temp +7;
}
```

The replacement of multiplication by addition will speed up the object code. Thus the strength of operation is reduced without changing the meaning of above code.

Code Generation

The final phase in compiler is the code generator. It takes as input an intermediate representation of the source program and produces as output an equivalent target program, as indicated in Figure below



Position of code generator

Code generation takes a linear sequence of 3-address intermediate code instructions, and translates each instruction into one or more instructions. The big issues in code generation are:

● ***Instruction selection***

● ***Register allocation and assignment***

Instruction selection: for each type of three-address statement, we can design a code skeleton that outlines the target code to be generated for that construct.

Example: every three address statement of the form X = Y + Z, where X,Y and Z are statically allocated, can be translated into the code sequence

```
Mov Y , R0    /* load Y into register R0 */
Add Z , R0    /* add Z to R0 */
Mov R0 , X    /* store R0 into X */
```

Register allocation and assignment

The efficient utilization of registers involving operands is particularly important in generating good code. The use of registers is often subdivided into two sub problems:

Register allocation: selecting the set of variables that will reside in registers at each point in the program.

Resister assignment: selecting specific register that a variable reside in, the goal of these operations is generally to minimize the total number of memory accesses required by the program.

Ex:

Consider the statement `d =(a + b)+(a - c)+(a - c)`

This may be translated into the following three address code

```
T = a + b
U = a - c
V = t + u
D = v + u
```

● The code

```
Mov R0,a
Mov R1,b
Add R1,R0
```

```
Mov R2,c
Sub R0,R2
Add R1 , R0
Add R0 , R1
Mov d, R0
```