PART A: Overview of compilation

1. OVERVIEW OF LANGUAGE PROCESSING SYSTEM

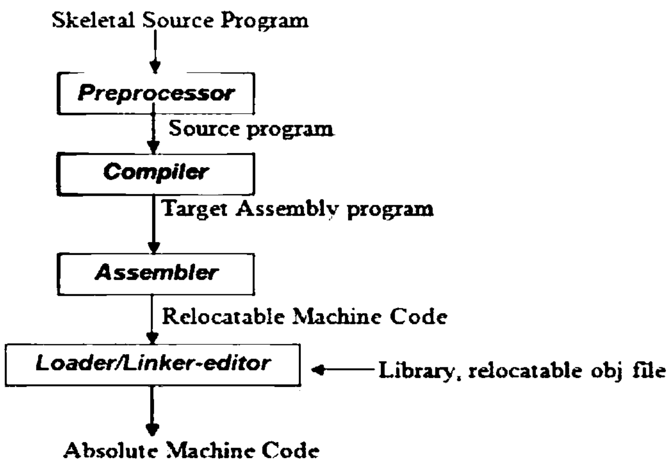


Fig 1.1 Language processing System

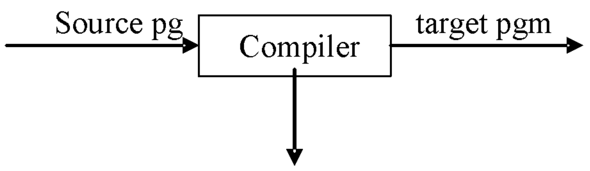
1.1 Preprocessor

A preprocessor produce input to compilers. They may perform the following functions.

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

1.2Compiler

Compiler is a translator program that translates a program written in (HLL) the source program and translates it into an equivalent program in (MLL) the target program. As an important part of a compiler is error showing to the programmer.

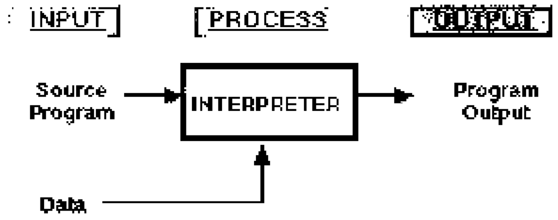


Error msg

Executing a program written n HLL programming language is basically of two parts. the source program must first be compiled translated into a object program. Then the results object program is loaded into a memory executed.

1.3ASSEMBLER*:* programmers found it difficult to write or read programs in machine language. They begin to use a mnemonic (symbols) for each machine instruction, which they would subsequently translate into machine language. Such a mnemonic machine language is now called an assembly language. Programs known as assembler were written to automate the translation of assembly language in to machine language. The input to an assembler program is called source program, the output is a machine language translation (object program).

1.4 INTERPRETER*:* An interpreter is a program that appears to execute a source program as if it were machine language.



Languages such as BASIC, SNOBOL, LISP can be translated using interpreters. JAVA also uses interpreter. The process of interpretation can be carried out in following phases.
1. Lexical analysis
2. Syntax analysis
3. Semantic analysis
4. Direct Execution

Advantages:

- Modification of user program can be easily made and implemented as execution proceeds.
- Type of object that denotes various may change dynamically.
- Debugging a program and finding errors is simplified task for a program used for interpretation.
- The interpreter for the language makes it machine independent.

Disadvantages:

- The execution of theprogramis slower.
- Memory consumption is more.

1.5 Loader and Link-editor:

Once the assembler procedures an object program, that program must be p laced into memory and executed. The assembler could place the object program directly in memory and transfer control to it, thereby causing the machine language program to be execute. This would waste core by leaving the assembler in memory while the user's program was being executed. Also the programmer would have to retranslate his program with each execution, thus wasting translation time. To overcome this problems of wasted translation time and memory. System programmers developed another component called loader.

"A loader is a program that places programs into memory and prepares them for execution." It would be more efficient if subroutines could be translated into object form the loader could "relocate" directly behind the user's program. The task of adjusting programs o they may be placed in arbitrary core locations is called relocation. Relocation loaders perform four functions.

1.6 TRANSLATOR

A translator is a program that takes as input a program written in one language and produces as output a program in another language. Beside program translation, the translator performs another very important role, the error-detection. Any violation of d HLL specification would be detected and reported to the programmers. Important role of translator are:

1 Translating the hll program input into an equivalent ml program.

2 Providing diagnostic messages wherever the programmer violates specification of the hll.

1.6.1TYPE OF TRANSLATORS:-

- Interpreter
- Compiler
- preprocessor

1.6.2 LIST OF COMPILERS

1. Ada compilers
2. ALGOL compilers
3. BASIC compilers
4. C# compilers
5. C compilers
6. C++ compilers
7. COBOL compilers

8. Java compilers

2. PHASES OF A COMPILER:

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

a. Analysis (Machine Independent/Language Dependent)
b. Synthesis (Machine Dependent/Language independent)

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

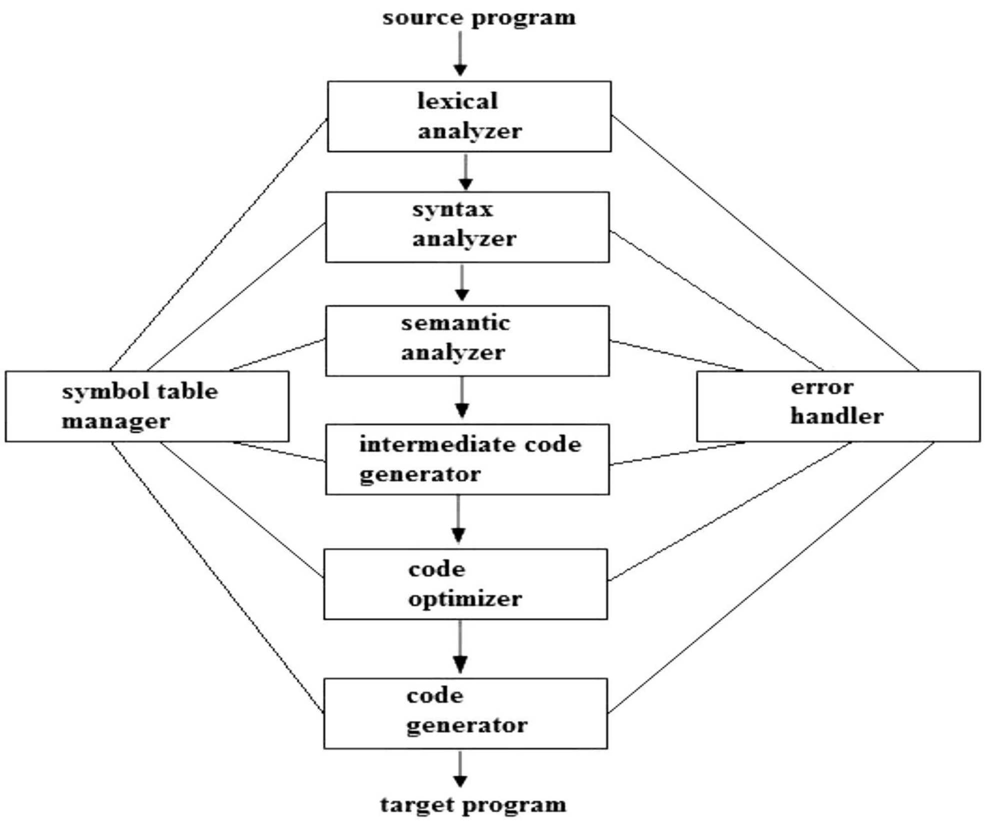


Fig 1.5 Phases of a compiler

Lexical Analysis:-

LA or Scanners reads the source program one character at a time, carving the source program into a sequence of automatic units called **tokens.**

Syntax Analysis:-

The second stage of translation is called syntax analysis or parsing. In this phase expressions, statements, declarations etc… are identified by using the results of lexical analysis. Syntax analysis is aided by using techniques based on formal grammar of the programming language.

Intermediate Code Generations:-

An intermediate representation of the final machine language code is produced. This phase bridges the analysis and synthesis phases of translation.

Code Optimization:-

This is optional phase described to improve the intermediate code so that the output runs faster and takes less space.

Code Generation:-

The last phase of translation is code generation. A number of optimizations to **Reduce the length of machine language program** are carried out during this phase. The output of the code generator is the machine language program of the specified computer.

Table Management (or) Book-keeping:-

This is the portion to **keep the names** used by the program and records essential information about each. The data structure used to record this information called a **'Symbol Table'.**

Error Handlers:-

It is invoked when a flaw error in the source program is detected. The output of LA is a stream of tokens, which is passed to the next phase, the syntax analyzer or parser. The SA groups the tokens together into syntactic structure called as **expression.** Expression may further be combined to form statements. The syntactic structure can be regarded as a tree whose leaves are the token called as parse trees.

The parser has two functions. It checks if the tokens from lexical analyzer, occur in pattern that are permitted by the specification for the source language. It also imposes on tokens a tree-like structure that is used by the sub-sequent phases of the compiler.

Example, if a program contains the expression **A+/B** after lexical analysis this expression might appear to the syntax analyzer as the token sequence **id+/id.** On seeing the /, the syntax analyzer should detect an error situation, because the presence of these two adjacent binary operators violates the formulations rule of an expression.

Syntax analysis is to make explicit the hierarchical structure of the incoming token stream by **identifying which parts of the token stream should be grouped**.

Example, (A/B*C has two possible interpretations.)

1- divide A by B and then multiply by C or

2- multiply B by C and then use the result to divide A.

Each of these two interpretations can be represented in terms of a parse tree.

Intermediate Code Generation:-

The intermediate code generation uses the structure produced by the syntax analyzer to create a stream of simple instructions. Many styles of intermediate code are possible. One common style uses instruction with one operator and a small number of operands.The output of the syntax analyzer is some representation of a parse tree. The intermediate code generation phase transforms this parse tree into an intermediate language representation of the source program.

Code Optimization:-

This is optional phase described to improve the intermediate code so that the output runs faster and takes less space. Its output is another intermediate code program that does the same job as the original, but in a way that saves time and / or spaces.

/* 1, Local Optimization:-

There are local transformations that can be applied to a program to make an improvement. For example,

If **A** > **B** goto **L2**
Goto **L3 L2 :**

This can be replaced by a single statement If **A** < **B** goto **L3**

Another important local optimization is the elimination of common sub-expressions

A := B + C + D
E := B + C + F

Might be evaluated as

T1 := B + C
A := T1 + D
E := T1 + F

Take this advantage of the common sub-expressions **B + C.**

Loop Optimization:-

Another important source of optimization concerns about **increasing the speed of loops**. A typical loop improvement is to move a computation that produces the same result each time around the loop to a point, in the program just before the loop is entered.*/

Code generator :-

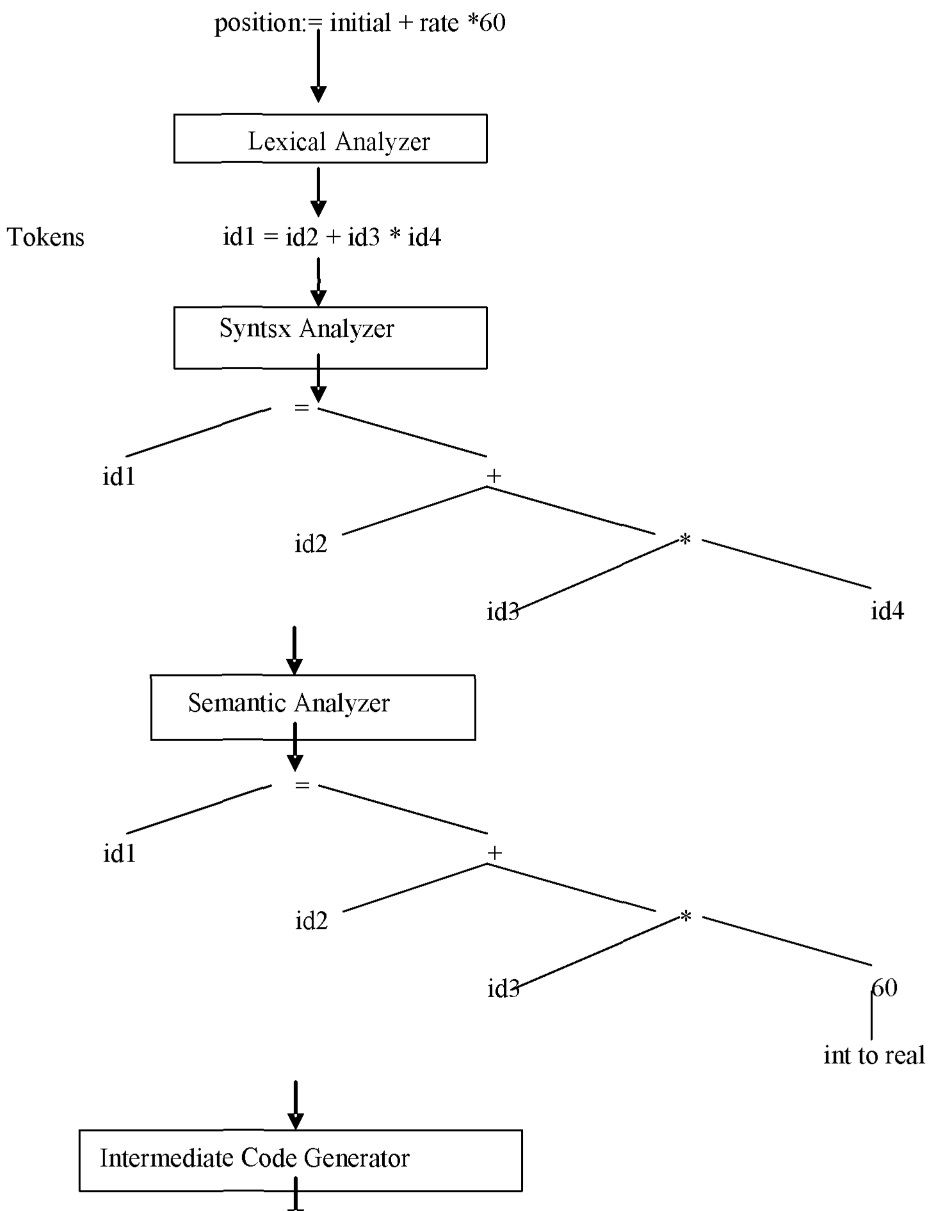
C produces the object code by deciding on the memory locations for data, selecting code to access each data and selecting the registers in which each computation is to be done. Many computers have only a few high speed registers in which computations can be performed quickly. A good code generator would attempt to utilize registers as efficiently as possible.

Error Handing :-

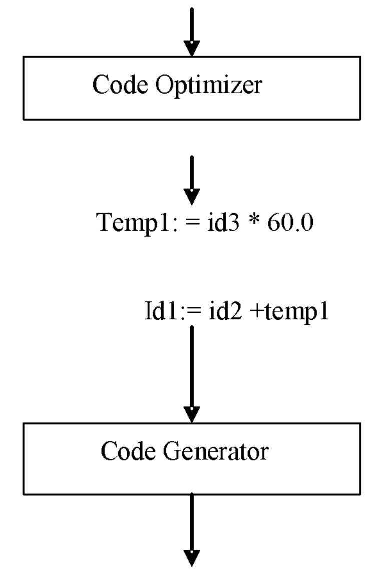
One of the most important functions of a compiler is the detection and reporting of errors in the source program. The error message should allow the programmer to determine exactly where the errors have occurred. Errors may occur in all or the phases of a compiler.

Whenever a phase of the compiler discovers an error, it must report the error to the error handler, which issues an appropriate diagnostic msg. Both of the table-management and error-Handling routines interact with all phases of the compiler.

Example:



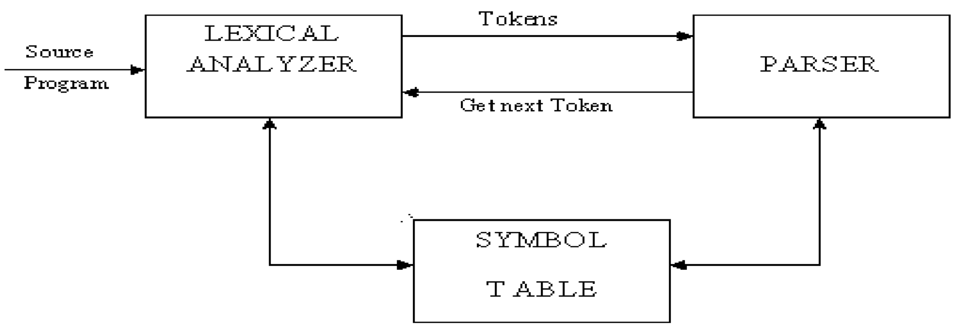
```
temp1:= int  to  real
(60)  temp2:=  id3  *
temp1 temp3:= id2 +
temp2
    id1:= temp3.
```



```
MOVF    id3,
r2       MULF
*60.0,       r2
MOVF    id2,
r2       ADDF
              r2
,  r1   MOVF
r1, id1
```

2.1 LEXICAL ANALYZER:

The LA is the first phase of a compiler. Lexical analysis is called as linear analysis or scanning. In this phase the stream of characters making up the source program is read from left-to-right and grouped into tokens that are sequences of characters having a collective meaning.



Upon receiving a ‘get next token’ command form the parser, the lexical analyzer reads the input character until it can identify the next token. The LA return to the parser representation for the token it has found. The representation will be an integer code, if the token is a simple construct such as parenthesis, comma or colon.

LA may also perform certain secondary tasks as the user interface. One such task is striping out from the source program the commands and white spaces in the form of blank, tab and new line characters. Another is correlating error message from the compiler with the source program.

2.1.1Lexical Analysis Vs Parsing:

Lexical analysis	Parsing
A Scanner simply turns an input String (say a file) into a list of tokens. These tokens represent things like identifiers, parentheses, operators etc. The lexical analyzer (the "lexer") parses individual symbols from the source code file into tokens. From there, the "parser" proper turns those whole tokens into sentences of your grammar	A parser converts this list of tokens into a Tree-like object to represent how the tokens fit together to form a cohesive whole (sometimes referred to as a sentence). A parser does not give the nodes any meaning beyond structural cohesion. The next thing to do is extract meaning from this structure (sometimes called contextual analysis).

2.1.2Token, Lexeme, Pattern:

Token: Token is a sequence of characters that can be treated as a single logical entity. Typical tokens are,
1) Identifiers 2) keywords 3) operators 4) special symbols 5) constants

Pattern: 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.
Lexeme: A lexeme is a sequence of characters in the source program that is matched by the pattern for a token.
Example:

Description of token

Token	**lexeme**	**pattern**
const	const	const
if	if	If

relation	<,<=,= ,< >,>=,>	< or <= or = or < > or >= or letter followed by letters & digit
i	pi	any numeric constant
nun	3.14	any character b/w "and "except"
literal	"core"	pattern

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

2.1.3 Lexical Errors:

Lexical errors are the errors thrown by the lexer when unable to continue. Which means that there's no way to recognise a lexeme as a valid token for you lexer? Syntax errors, on the other side, will be thrown by your scanner when a given set of **already** recognized valid tokens don't match any of the right sides of your grammar rules. Simple panic-mode error handling system requires that we return to a high-level parsing function when a parsing or lexical error is detected.

Error-recovery actions are:

- Delete one character from the remaining input.
- Insert a missing character in to the remaining input.
- Replace a character by another character.
- Transpose two adjacent characters.

3. Difference Between Compiler And Interpreter:

1. A compiler converts the high level instruction into machine language while an interpreter converts the high level instruction into an intermediate form.
2. Before execution, entire program is executed by the compiler whereas after translating the first line, an interpreter then executes it and so on.
3. List of errors is created by the compiler after the compilation process while an interpreter stops translating after the first error.
4. An independent executable file is created by the compiler whereas interpreter is required by an interpreted program each time.
5. The compiler produce object code whereas interpreter does not produce object code. In the process of compilation the program is analyzed only once and then the code is generated whereas source program is interpreted every time it is to be executed and every time the source program is analyzed. Hence interpreter is less efficient than compiler.

6. **Examples of interpreter:** A UPS Debugger is basically a graphical source level debugger but it contains built in C interpreter which can handle multiple source files.
7. **Example of compiler:** Borland c compiler or Turbo C compiler compiles the programs written in C or C++.

4. REGULAR EXPRESSIONS:

4.1: SPECIFICATION OF TOKENS

There are 3 specifications of tokens:
1) Strings
2) Language
3) Regular expression

Strings and Languages

An **alphabet** or character class is a finite set of symbols.
A **string** over an alphabet is a finite sequence of symbols drawn from that alphabet.
A **language** is any countable set of strings over some fixed alphabet.

In language theory, the terms "sentence" and "word" are often used as synonyms for "string." The length of a string s, usually written |s|, is the number of occurrences of symbols in s. For example, banana is a string of length six. The empty string, denoted ε, is the string of length zero.

Operations on strings
The following string-related terms are commonly used:

1. A **prefix** of string s is any string obtained by removing zero or more symbols from the end of strings.
   For example, ban is a prefix of banana.

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

3. A **substring** of s is obtained by deleting any prefix and any suffix from s.
   For example, nan is a substring of banana.

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

5. A subsequence of s is any string formed by deleting zero or more not necessarily consecutive positions of s.

For example, baan is a subsequence of banana.

Operations on languages:
The following are the operations that can be applied to languages:
1.Union
2.Concatenation
3.Kleene closure 4.Positive closure

The following example shows the operations on strings:

Let L={0,1} and S={a,b,c}

1. Union	: L U S={0,1,a,b,c}
2. Concatenation	: L.S={0a,1a,0b,1b,0c,1c}
3. Kleene closure	: L^*={ ε,0,1,00….}
4. Positive closure	: L^+={0,1,00….}

4.2Regular Expressions:

Each regular expression r denotes a language L(r).

Here are the rules that define the regular expressions over some alphabet Σ and the languages that those expressions denote:

1. ε is a regular expression, and L(ε) is { ε }, that is, the language whose sole member is the empty string.

2. If'a'is a symbol in Σ, then 'a'is a regular expression, and L(a) = {a}, that is, the language with one string, of length one, with 'a'in its one position.

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

   - (r)|(s) is a regular expression denoting the language L(r) U L(s).
   - (r)(s) is a regular expression denoting the language L(r)L(s).
   - (r)* is a regular expression denoting (L(r))*.
   - (r) is a regular expression denoting L(r).

4. The unary operator * has highest precedence and is left associative.

5. Concatenation has second highest precedence and is left associative.
   has lowest precedence and is left associative.

4.2.1 REGULAR DEFINITIONS:

For notational convenience, we may wish to give names to regular expressions and to define regular expressions using these names as if they were symbols.

Identifiers are the set or string of letters and digits beginning with a letter. The following regular definition provides a precise specification for this class of string.

Example-1,

Ab*|cd? Is equivalent to (a(b*)) | (c(d?)) Pascal identifier

Letter - A | B | ……| Z | a | b |……| z| Digits - 0 | 1 | 2 | …. | 9

Id - letter (letter / digit)*

Shorthand's

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

1. One or more instances (+):

- The unary postfix operator + means " one or more instances of" .
- If r is a regular expression that denotes the language L(r), then $(r)^+$ is a regular expression that denotes the language $(L (r))^+$
- Thus the regular expression a^+ denotes the set of all strings of one or more a's.
- The operator $^+$ has the same precedence and associativity as the operator *.

2. Zero or one instance (?):

- The unary postfix operator ? means "zero or one instance of".

- The notation r? is a shorthand for r | ε.

- If 'r' is a regular expression, then (r)? is a regular expression that denotes the language L(r) U { ε }.

3. Character Classes:

- The notation [abc] where a, b and c are alphabet symbols denotes the regular expression a | b | c.
- Character class such as [a – z] denotes the regular expression a | b | c | d | ….|z.
- We can describe identifiers as being strings generated by the regular expression,
  [A–Za–z][A–Za–z0–9]*

Non-regular Set

A language which cannot be described by any regular expression is a non-regular set. Example: The set of all strings of balanced parentheses and repeating strings cannot be described by a regular expression. This set can be specified by a context-free grammar.

4.2.2RECOGNITION OF TOKENS:

Consider the following grammar fragment:

stmt → if expr then stmt

|if expr then stmt else stmt |ε

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	→	$digit^{+}$ $(.digit^{+})?(E(+\|-)?digit^{+})?$

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

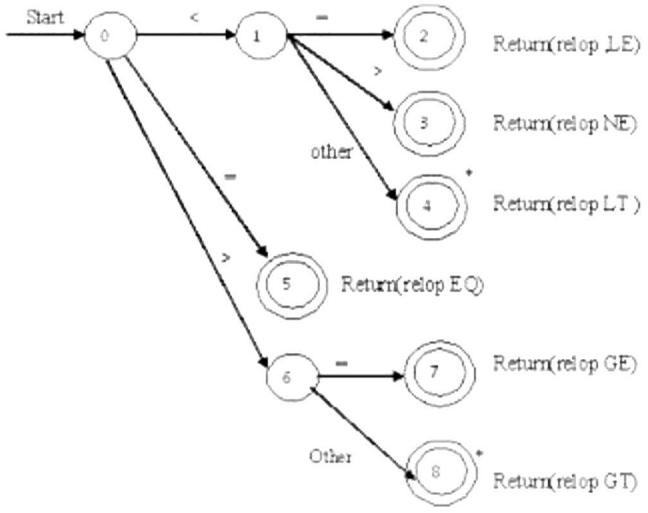
Lexeme	Token Name	Attribute Value
Any ws	_	_
if	if	_
then	then	_
else	else	_
Any id	id	pointer to table entry
Any number	number	pointer to table entry
<	relop	LT
<=	relop	LE
=	relop	ET
< >	relop	NE

4.3 TRANSITION DIAGRAM:

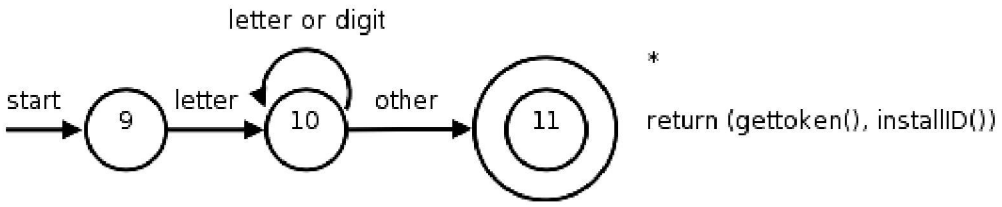
Transition Diagram has a collection of nodes or circles, called states. Each state represents a condition that could occur during the process of scanning the input looking for a lexeme that matches one of several patterns .Edges are directed from one state of the transition diagram to another. each edge is labeled by a symbol or set of symbols.If we are in one state s, and the next input symbol is a, we look for an edge out of state s labeled by a. if we find such an edge ,we advance the forward pointer and enter the state of the transition diagram to which that edge leads.

Some important conventions about transition diagrams are

1. Certain states are said to be accepting or final .These states indicates that a lexeme has been found, although the actual lexeme may not consist of all positions b/w the lexeme Begin and forward pointers we always indicate an accepting state by a double circle.
2. In addition, if it is necessary to return the forward pointer one position, then we shall additionally place a * near that accepting state.
3. One state is designed the state ,or initial state ., it is indicated by an edge labeled "start" entering from nowhere .the transition diagram always begins in the state before any input symbols have been used.



As an intermediate step in the construction of a LA, we first produce a stylized flowchart, called a transition diagram. Position in a transition diagram, are drawn as circles and are called as states.



The above TD for an identifier, defined to be a letter followed by any no of letters or digits.A sequence of transition diagram can be converted into program to look for the tokens specified by the diagrams. Each state gets a segment of code.

4.4 Automata:

Automation is defined as a system where information is transmitted and used for performing some functions without direct participation of man.

1. An automation in which the output depends only on the input is **called automation without memory.**
2. An automation in which the output depends on the input and state also is **called as automation with memory**.
3. An automation in which the output depends only on the state of the machine is **called a Moore machine**.
4. An automation in which the output depends on the state and input at any instant of time is **called a mealy machine**.

4.4.1 DESCRIPTION OF AUTOMATA

1. An automata has a mechanism to read input from input tape,
2. Any language is recognized by some automation, Hence these automation are basically language 'acceptors' or 'language recognizers'.

Types of Finite Automata

Deterministic Automata
Non-Deterministic Automata.

Deterministic Automata:

A deterministic finite automata has at most one transition from each state on any input. A DFA is a special case of a NFA in which:-

1. it has no transitions on input €,
2. Each input symbol has at most one transition from any state.

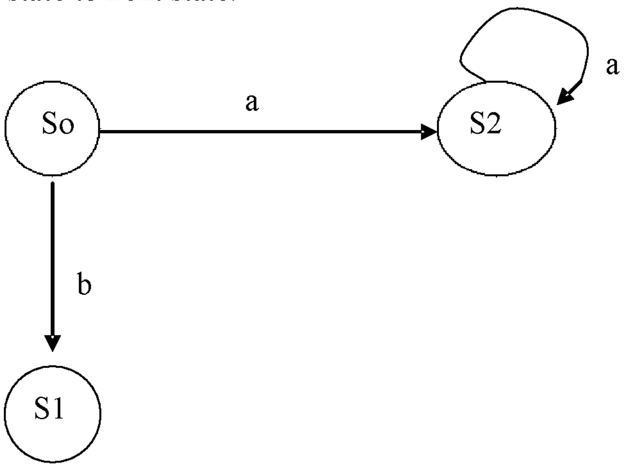
DFA formally defined by 5 tuple notation M = (Q, ∑, δ, qo, F), where Q is a finite 'set of states', which is non empty.

∑ is 'input alphabets', indicates input set.
qo is an 'initial state' and qo is in Q ie, qo, ∑, Q F is
a set of 'Final states',
δ is a 'transmission function' or mapping function, using this function the next state can be determined.

The regular expression is converted into minimized DFA by the following procedure:

Regular expression → NFA → DFA → Minimized DFA

The Finite Automata is called DFA if there is only one path for a specific input from current state to next state.



From state S0 for input 'a' there is only one path going to S2. similarly from so there is only one path for input going to S1.

Nondeterministic Automata:

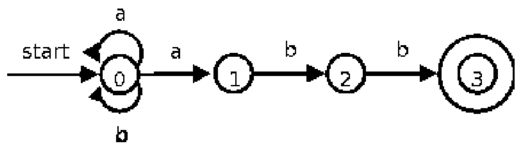
A NFA ia A mathematical model consists of

- A set of states S.
- A set of input symbols ∑.
- A transition is a move from one state to another.
- A state so that is distinguished as the start (or initial) state
- A set of states F distinguished as accepting (or final) state.
- A number of transition to a single symbol.

A NFA can be diagrammatically represented by a labeled directed graph, called a transition graph, in which the nodes are the states and the labeled edges represent the transition function.

This graph looks like a transition diagram, but the same character can label two or more transitions out of one state and edges can be labeled by the special symbol € as well as input symbols.

The transition graph for an NFA that recognizes the language (a|b)*abb is shown



5. Bootstrapping:

When a computer is first turned on or restarted, a special type of absolute loader, called as bootstrap loader is executed. This bootstrap loads the first program to be run by the computer usually an operating system. The bootstrap itself begins at address O in the memory of the machine. It loads the operating system (or some other program) starting at address 80. After all of the object code from device has been loaded, the bootstrap program jumps to address 80, which begins the execution of the program that was loaded.

Such loaders can be used to run stand-alone programs independent of the operating system or the system loader. They can also be used to load the operating system or the loader itself into memory.

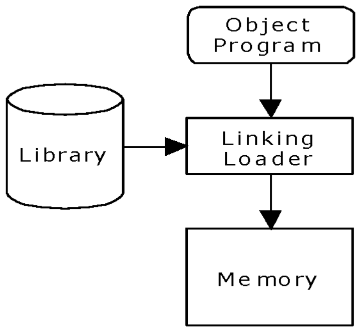
Loaders are of two types:

- Linking loader.
- Linkage editor.

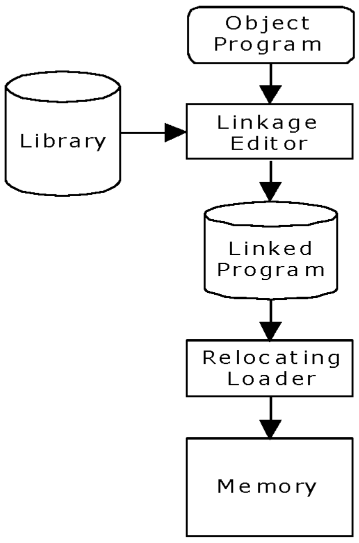
Linkage loaders, perform all linking and relocation at load time.

Linkage editors, perform linking prior to load time and dynamic linking, in which the linking function is performed at execution time.

A linkage editor performs linking and some relocation; however, the linkaged program is written to a file or library instead of being immediately loaded into memory. This approach reduces the overhead when the program is executed. All that is required at load time is a very simple form of relocation.



Linking Loader



Linkage Editor

6. Pass And Phases Of Translation:

Phases: (Phases are collected into a front end and back end)

Frontend:

The front end consists of those phases, or parts of phase, that depends primarily on the source language and is largely independent of the target machine. These normally include lexical and syntactic analysis, the creation of the symbol 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 tha goes along with each of these phases.

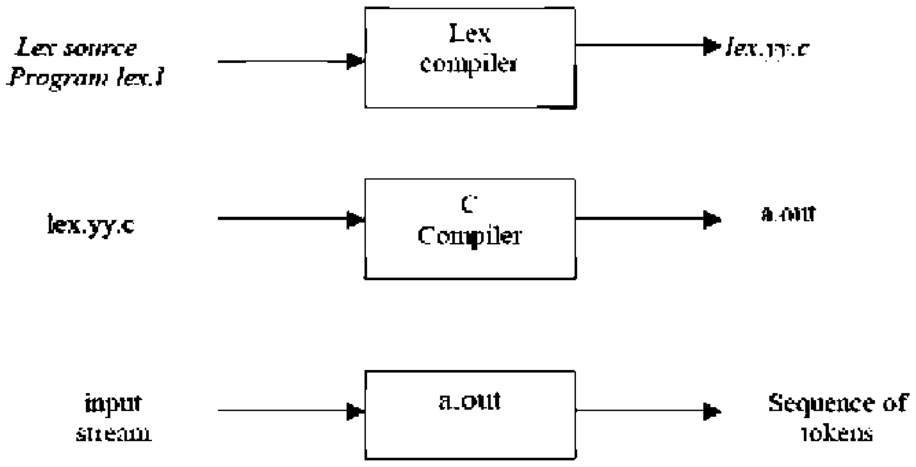
Back end:

The back end includes those portions of the compiler that depend on the target machine and generally, these portions do not depend on the source language .

7. Lexical Analyzer Generator:

7.1Creating a lexical analyzer with Lex:

- First, a specification of a lexical analyzer is prepared by creating a program lex.l in the Lex language. Then, lex.l is run through the Lex compiler to produce a C program lex.yy.c.
- Finally, lex.yy.c is run through the C compiler to produce an object program a.out, which is the lexical analyzer that transforms an input stream into a sequence of tokens.



Lex Specification

A Lex program consists of three parts:

```
{ definitions }
%%
{ rules }
%%
{ user subroutines }
```

- **Definitions** include declarations of variables, constants, and regular definitions
- **Rules** are statements of the form p1 {action1}p2 {action2} … pn {action}
- where pi is regular expression and actioni describes what action the lexical analyzer should take when pattern pi matches a lexeme. Actions are written in C code.
- **User subroutines** are auxiliary procedures needed by the actions. These can be compiled separately and loaded with the lexical analyzer.

8. INPUT BUFFERING

The LA scans the characters of the source program one at a time to discover tokens. Because of large amount of time can be consumed scanning characters, specialized buffering techniques have been developed to reduce the amount of overhead required to process an input character.

Buffering techniques:

1. Buffer pairs
2. Sentinels

The lexical analyzer scans the characters of the source program one a t a time to discover tokens. Often, however, many characters beyond the next token many have to be examined before the next token itself can be determined. For this and other reasons, it is desirable for the lexical analyzer to read its input from an input buffer. Figure shows a buffer divided into two halves of, say 100 characters each. One pointer marks the beginning of the token being discovered. A look ahead pointer scans ahead of the beginning point, until the token is discovered .we view the position of each pointer as being between the character last read and the character next to be read. In practice each buffering scheme adopts one convention either a pointer is at the symbol last read or the symbol it is ready to read.

Token beginnings look ahead pointer, The distance which the look ahead pointer may have to travel past the actual token may be large.

For example, in a PL/I program we may see: DECALRE (ARG1, ARG2… ARG *n*) without knowing whether DECLARE is a keyword or an array name until we see the character that follows the right parenthesis.

PART-B

TOPDOWN PARSING

1. Context-free Grammars: Definition:

Formally, a context-free grammar G is a 4-tuple G = (V, T, P, S), where:

1. V is a finite set of variables (or nonterminals). These describe sets of "related" strings.
2. T is a finite set of terminals (i.e., tokens).
3. P is a finite set of productions, each of the form

   $A \rightarrow \alpha$

   where $A \in V$ is a variable, and $\alpha \in (V \cup T)^*$ is a sequence of terminals and nonterminals.

$S \in V$ is the start symbol.

Example of CFG:

E ==>EAE | (E) | -E | id

A==> + | - | * | / |

Where E, A are the non-terminals while id, +, *, -, /,(,) are the terminals.

2. Syntax analysis:

In syntax analysis phase the source program is analyzed to check whether if conforms to the source language's syntax, and to determine its phase structure. This phase is often separated into two phases:

- Lexical analysis: which produces a stream of tokens?
- Parser: which determines the phrase structure of the program based on the context-free grammar for the language?

2.1 PARSING:

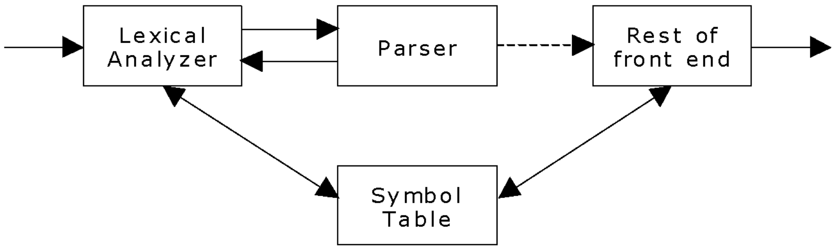
Parsing is the activity of checking whether a string of symbols is in the language of some grammar, where this string is usually the stream of tokens produced by the lexical analyzer. If the string is in the grammar, we want a parse tree, and if it is not, we hope for some kind of error message explaining why not.

There are two main kinds of parsers in use, named for the way they build the parse trees:

- Top-down: A top-down parser attempts to construct a tree from the root, applying productions forward to expand non-terminals into strings of symbols.
- Bottom-up: A Bottom-up parser builds the tree starting with the leaves, using productions in reverse to identify strings of symbols that can be grouped together.

In both cases the construction of derivation is directed by scanning the input sequence from left to right, one symbol at a time.

Parse Tree:



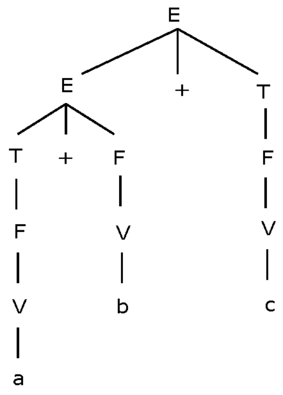
A parse tree is the graphical representation of the structure of a sentence according to its grammar.

Example:
Let the production P is:

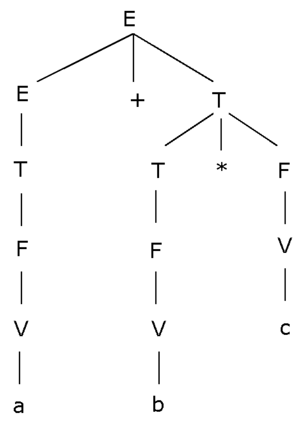
E→ T | E+T
T→ F | T*F
F→ V | (E)
V→ a | b | c |d

The parse tree may be viewed as a representation for a derivation that filters out the choice regarding the order of replacement.

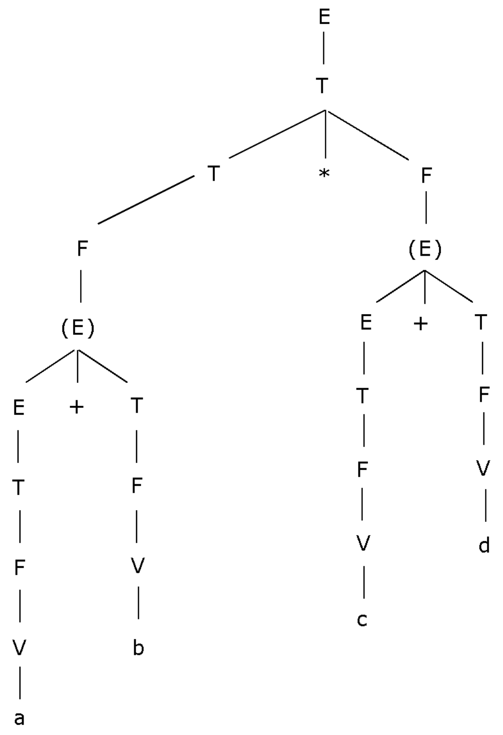
Parse tree for a * b + c



Parse tree for a + b * c is:



Parse tree for (a * b) * (c + d)



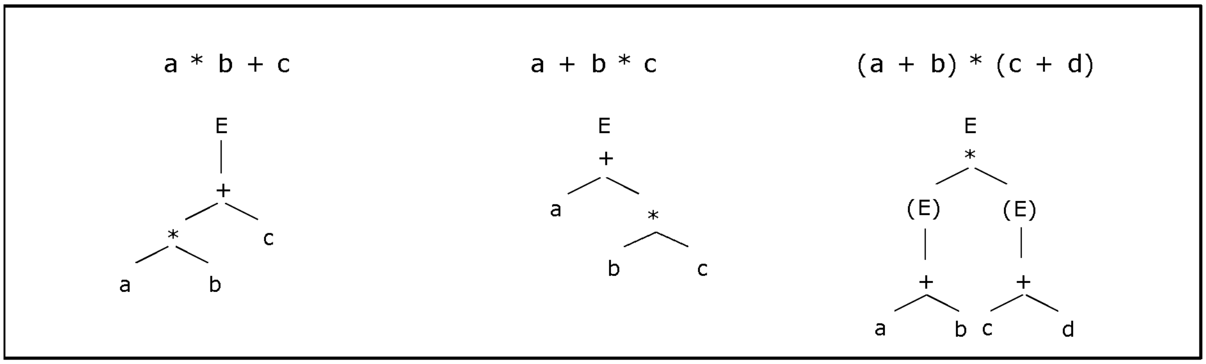
2.2 SYNTAX TREES:

Parse tree can be presented in a simplified form with only the relevant structure information by:

- Leaving out chains of derivations (whose sole purpose is to give operators difference precedence).

- Labeling the nodes with the operators in question rather than a non-terminal.

The simplified Parse tree is sometimes called as structural tree or syntax tree.



Syntax Trees

Syntax Error Handling:

If a compiler had to process only correct programs, its design & implementation would be greatly simplified. But programmers frequently write incorrect programs, and a good compiler should assist the programmer in identifying and locating errors.The programs contain errors at many different levels.
For example, errors can be:

1) Lexical – such as misspelling an identifier, keyword or operator
2) Syntactic – such as an arithmetic expression with un-balanced parentheses.
3) Semantic – such as an operator applied to an incompatible operand.
4) Logical – such as an infinitely recursive call.

Much of error detection and recovery in a compiler is centered around the syntax analysis phase.
The goals of error handler in a parser are:

- It should report the presence of errors clearly and accurately.
- It should recover from each error quickly enough to be able to detect subsequent errors.
- It should not significantly slow down the processing of correct programs.

2.3Ambiguity:

Several derivations will generate the same sentence, perhaps by applying the same productions in a different order. This alone is fine, but a problem arises if the same sentence has two distinct parse trees. A grammar is ambiguous if there is any sentence with more than one parse tree.

Any parses for an ambiguous grammar has to choose somehow which tree to return. There are a number of solutions to this; the parser could pick one arbitrarily, or we can provide some hints about which to choose. Best of all is to rewrite the grammar so that it is not ambiguous.

There is no general method for removing ambiguity. Ambiguity is acceptable in spoken languages. Ambiguous programming languages are useless unless the ambiguity can be resolved.

Fixing some simple ambiguities in a grammar:

	Ambiguous	language	unambiguous
(i)	$A \rightarrow B \mid AA$	Lists of one or more B's	$A \rightarrow BC$
		$C \rightarrow A \mid E$	
(ii)	$A \rightarrow B \mid A;A$	Lists of one or more B's with punctuation	$A \rightarrow BC$
		$C \rightarrow ;A \mid E$	
(iii)	$A \rightarrow B \mid AA \mid E$	lists of zero or more B's	$A \rightarrow BA \mid E$

Any sentence with more than two variables, such as (arg, arg, arg) will have multiple parse trees.

2.4 Left Recursion:

If there is any non terminal A, such that there is a derivation $A \overset{+}{\Rightarrow} A\,\alpha$ for some stringα, then the grammar is left recursive.

Algorithm for eliminating left Recursion:

1. Group all the A productions together like this:

   $A \Rightarrow A\,\alpha_1 \mid A\,\alpha_2 \mid - - - \mid A\,\alpha_m \mid \beta_1 \mid \beta_2 \mid - - - \mid \beta_n$

   Where,

   A is the left recursive non-terminal,

   α is any string of terminals and

   β is any string of terminals and non terminals that does not begin with A.

2. Replace the above A productions by the following:

   $A \Rightarrow \beta_1\,A^I \mid \beta_2\,A^I \mid - - - \mid \beta_n\,A^I$

   $A^I \Rightarrow \alpha_1\,A^I \mid \alpha_2\,A^I \mid - - - \mid \alpha_m\,A^I \mid \in$

   Where, A^I is a new non terminal.

Top down parsers cannot handle left recursive grammars.

If our expression grammar is left recursive:

- This can lead to non termination in a top-down parser.
- for a top-down parser, any recursion must be right recursion.
- we would like to convert the left recursion to right recursion.

Example 1:
Remove the left recursion from the production: $A \to \underbrace{A\ \alpha}_{\textbf{Left Recursive. Eliminate}} \mid \beta$

Applying the transformation yields:

$A \to \beta\ A^I$
$A^I \to \underset{\uparrow}{\alpha}\ A^I \mid \in$
Remaining part after A.

Example 2:
Remove the left recursion from the productions:
$E \to E + T \mid T$
$T \to T * F \mid F$
Applying the transformation yields:
$E \to T\ E^I$ $\quad$ $T \to F\ T^I$
$E^I \to T\ E^I \mid \in$ $\quad$ $T^I \to *\ F\ T^I \mid \in$
Example 3:
Remove the left recursion from the productions:
$E \to E + T \mid E - T \mid T$
$T \to T * F \mid T/F \mid F$
Applying the transformation yields:

$E \to T\ E^I$ $\quad$ $T \to F\ T^I$
$E \to +\ T\ E^I \mid -\ T\ E^I \mid \in$ $\quad$ $T^I \to *\ F\ T^I \mid /F\ T^I \mid \in$
Example 4:
Remove the left recursion from the productions:
$S \to A\ a \mid b$
$A \to A\ c \mid S\ d \mid \in$
1. The non terminal S is left recursive because $S \to A\ a \to S\ d\ a$
   But it is not immediate left recursive.
2. Substitute S-productions in $A \to S\ d$ to obtain:
   $A \to A\ c \mid A\ a\ d \mid b\ d \mid \in$
3. Eliminating the immediate left recursion:

$S \rightarrow A\ a \mid b$
$A \rightarrow b\ d\ A^I \mid A^I$
$A^I \rightarrow c\ A^I \mid a\ d\ A^I \mid \in$

Example 5:
Consider the following grammar and eliminate left recursion.
$S \rightarrow A\ a \mid b$
$A \rightarrow S\ c \mid d$
The nonterminal S is left recursive in two steps:
$S \rightarrow A\ a \rightarrow S\ c\ a \rightarrow A\ a\ c\ a \rightarrow S\ c\ a\ c\ a - - -$
Left recursion causes the parser to loop like this, so remove:
Replace $A \rightarrow S\ c \mid d$ by $A \rightarrow A\ a\ c \mid b\ c \mid d$
and then by using Transformation rules:
$A \rightarrow b\ c\ A^I \mid d\ A^I$
$A^I \rightarrow a\ c\ A^I \mid \in$

2.5 Left Factoring:
Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive parsing.
When it is not clear which of two alternative productions to use to expand a non-terminal A, we may be able to rewrite the productions to defer the decision until we have some enough of the input to make the right choice.

Algorithm:
For all $A \in$ non-terminal, find the longest prefix α that occurs in two or more right-hand sides of A.
If $\alpha \neq \in$ then replace all of the A productions,
$A \rightarrow \alpha\ \beta_I \mid \alpha\ \beta_2 \mid - - - \mid \alpha\ \beta_n \mid r$
With
$A \rightarrow \alpha\ A^I \mid r$
$A^I \rightarrow \beta_I \mid \beta_2 \mid - - - \mid \beta_n \mid \in$
Where, A^I is a new element of non-terminal.
Repeat until no common prefixes remain.
It is easy to remove common prefixes by left factoring, creating new non-terminal.
For example consider:
$V \rightarrow \alpha\ \beta \mid \alpha\ r$
Change to:
$V \rightarrow \alpha\ V^I$
$V^I \rightarrow \beta \mid r$

Example 1:
Eliminate Left factoring in the grammar:
S → V := int
V → alpha '[' int ']' | alpha

Becomes:

$S \rightarrow V := \text{int}$
$V \rightarrow \text{alpha } V^I$
$V^I \rightarrow$ '[' int '] | $\in$

2.6 TOP DOWN PARSING:

Top down parsing is the construction of a Parse tree by starting at start symbol and "guessing" each derivation until we reach a string that matches input. That is, construct tree from root to leaves.

The advantage of top down parsing in that a parser can directly be written as a program. Table-driven top-down parsers are of minor practical relevance. Since bottom-up parsers are more powerful than top-down parsers, bottom-up parsing is practically relevant.

For example, let us consider the grammar to see how top-down parser works:

S → if E then S else S | while E do S | print
E → true | False | id

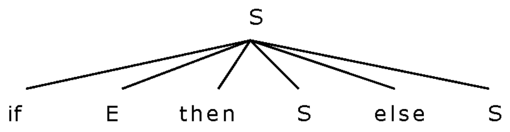
The input token string is: If id then while true do print else print.

1. Tree:

S

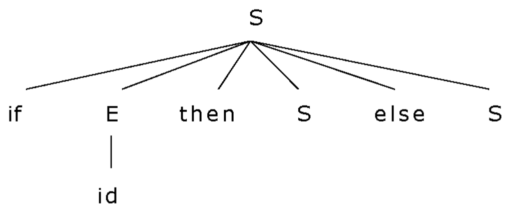
Input: if id then while true do print else print.
Action: Guess for S.

2. Tree:



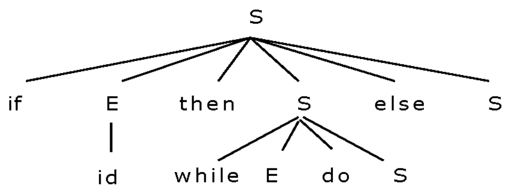
Input: if id then while true do print else print.
Action: if matches; guess for E.

3. Tree:

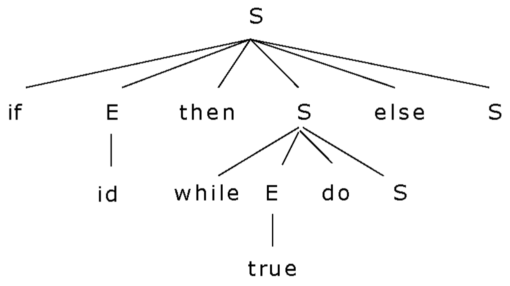


Input: id then while true do print else print.
Action: id matches; then matches; guess for S.

4. Tree:

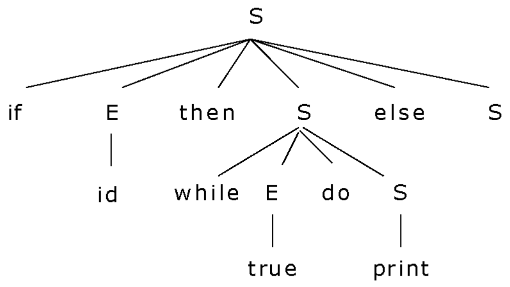


Input: while true do print else print.
Action: while matches; guess for E.
5. Tree:



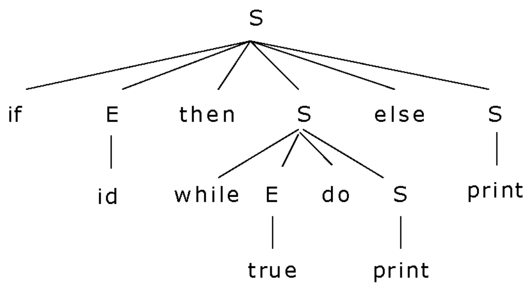
Input: true do print else print
Action:true matches; do matches; guess S.

6. Tree:



Input: print else print.
Action: print matches; else matches; guess for S.

7. Tree:



Input: print.
Action: print matches; input exhausted; done.

2.6.1. Recursive Descent Parsing:

Top-down parsing can be viewed as an attempt to find a left most derivation for an input string. Equivalently, it can be viewd as a attempt to construct a parse tree for the input starting from the root and creating the nodes of the parse tree in preorder.

The special case of recursive –decent parsing, called predictive parsing, where no backtracking is required. The general form of top-down parsing, called recursive descent, that may involve backtracking, that is, making repeated scans of the input.

Recursive descent or predictive parsing works only on grammars where the first terminal symbol of each sub expression provides enough information to choose which production to use.

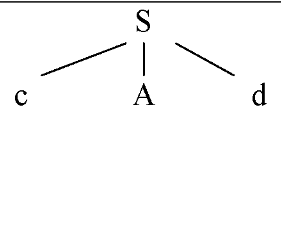
Recursive descent parser is a top down parser involving backtracking. It makes a repeated scans of the input. Backtracking parsers are not seen frequently, as backtracking is very needed to parse programming language constructs.

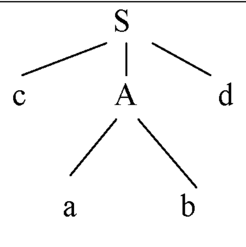
Example: consider the grammar

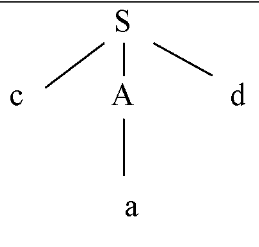
S→cAd

A→ab|a

And the input string w=cad. To construct a parse tree for this string top-down, we initially create a tree consisting of a single node labeled scan input pointer points to c, the first symbol of w. we then use the first production for S to expand tree and obtain the tree of Fig(a).


Fig(a)


Fig(b)


Fig(c)

The left most leaf, labeled c, matches the first symbol of w, so we now advance the input pointer to a ,the second symbol of w, and consider the next leaf, labeled A. We can then expand A using the first alternative for A to obtain the tree in Fig (b). we now have a match for the second input symbol so we advance the input pointer to d, the third, input symbol, and compare d against the next leaf, labeled b. since b does not match the d ,we report failure and go back to A to see where there is any alternative for Ac that we have not tried but that might produce a match.

In going back to A, we must reset the input pointer to position2,we now try second alternative for A to obtain the tree of Fig(c).The leaf matches second symbol of w and the leaf d matches the third symbol .

The left recursive grammar can cause a recursive- descent parser, even one with backtracking, to go into an infinite loop.That is ,when we try to expand A, we may eventually find ourselves again trying to ecpand A without Having consumed any input.

2.6.2. Predictive Parsing:

Predictive parsing is top-down parsing without backtracking or look a head. For many languages, make perfect guesses (avoid backtracking) by using 1-symbol look-a-head. i.e., if:

$A \rightarrow \alpha_I \mid \alpha_2 \mid - - - \mid \alpha_n$.

Choose correct α_i by looking at first symbol it derive. If $\in$ is an alternative, choose it last.

This approach is also called as predictive parsing. There must be at most one production in order to avoid backtracking. If there is no such production then no parse tree exists and an error is returned.

The crucial property is that, the grammar must not be left-recursive.

Predictive parsing works well on those fragments of programming languages in which keywords occurs frequently.

For example:

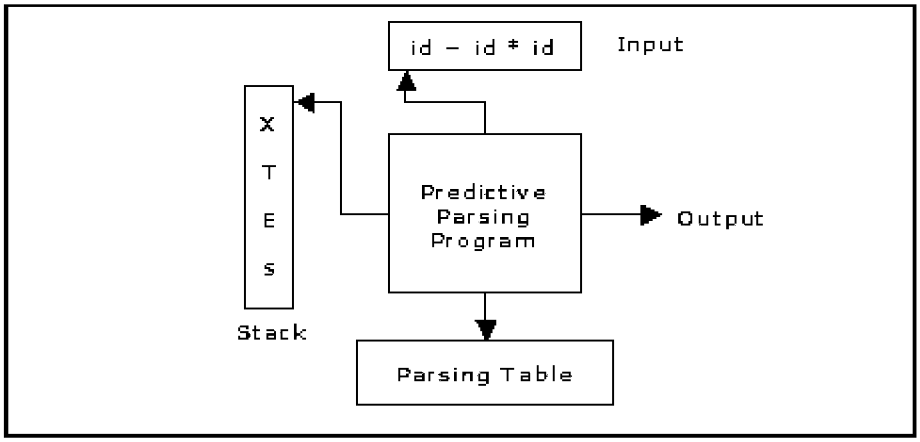
stmt → **if** exp **then** stmt **else** stmt

| **while** expr **do** stmt

| **begin** stmt-list **end**.

then the keywords if, while and begin tell, which alternative is the only one that could possibly succeed if we are to find a statement.

The model of predictive parser is as follows:



A predictive parser has:

- Stack
- Input
- Parsing Table
- Output

The input buffer consists the string to be parsed, followed by $, a symbol used as a right end marker to indicate the end of the input string.
The stack consists of a sequence of grammar symbols with $ on the bottom, indicating the bottom of the stack. Initially the stack consists of the start symbol of the grammar on the top of $.
Recursive descent and LL parsers are often called predictive parsers, because they operate by predicting the next step in a derivation.

The algorithm for the Predictive Parser Program is as follows:
Input: A string w and a parsing table M for grammar G
Output: if w is in L(g),a leftmost derivation of w; otherwise, an error indication.
Method: Initially, the parser has $S on the stack with S, the start symbol of G on top, and w$ in the input buffer. The program that utilizes the predictive parsing table M to produce a parse for the input is:

```
Set ip to point to the first symbol of w$;
repeat
    let x be the top stack symbol and a the symbol pointed to by ip;
    if X is a terminal or $ then
        if X = a then
            pop X from the stack and advance ip
        else error()
    else                            /* X is a non-terminal */
        if M[X, a] = X → $Y_1$ $Y_2$ . . . . . . . $Y_k$ then begin
```

```
            pop X from the stack;
            push $Y_k, Y_{k-1}, \ldots Y_1$ onto the stack, with $Y_1$ on top;
            output the production $X \rightarrow Y_1 Y_2 \ldots Y_k$
        end
        else error()
    until X = $ /*stack is empty*/
```

2.6.3 FIRST and FOLLOW:

The construction of a predictive parser is aided by two functions with a grammar G. these functions, FIRST and FOLLOW, allow us to fill in the entries of a predictive parsing table for G, whenever possible. Sets of tokens yielded by the **FOLLOW** function can also be used as synchronizing tokens during pannic-mode error recovery.

If α is any string of grammar symbols, let FIRST (α) be the set of terminals that begin the strings derived from α. If α=>€,then € is also in FIRST(α).

Define FOLLOW (A), for nonterminals A, to be the set of terminals a that can appear immediately to the right of A in some sentential form, that is, the set of terminals a such that there exist a derivation of the form S=>αAaβ for some α and β. If A can be the rightmost symbol in some sentential form, then $ is in FOLLOW(A).

Computation of FIRST ():

To compute **FIRST(X)** for all grammar symbols X, apply the following rules until no more terminals or € can be added to any FIRST set.

- If X is terminal, then FIRST(X) is {X}.
- If X→€ is production, then add € to FIRST(X).
- If X is nonterminal and X→Y_1 Y_2......Y_k is a production, then place a in FIRST(X) if for some i,a is in FIRST(Y_i),and € is in all of FIRST(Y_i),and € is in all of FIRST(Y_1),..... FIRST(Y_{i-1});that is Y_1......... Y_{i-1}==>€.if € is in FIRST(Y_j), for all j=,2,3.......k, then add € to FIRST(X).for example, everything in FIRST(Y_1) is surely in FIRST(X).if Y_1 does not derive €,then we add nothing more to FIRST(X),but if Y_1=>€,then we add FIRST(Y_2) and so on.

FIRST (A) = FIRST (α_1) U FIRST (α_2) U - - - U FIRST (α_n)
Where, A → α_1 | α_2 | - - - |α_n, are all the productions for A.
FIRST (Aα) = if $\in \notin$ FIRST (A) then FIRST (A)
else (FIRST (A) - {$\in$}) U FIRST (α)

Computation of FOLLOW ():

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

- Place \$ in FOLLOW(s), where S is the start symbol and \$ is input right end marker .
- If there is a production A→αBβ,then everything in FIRST(β) except for € is placed in FOLLOW(B).
- If there is production A→αB, or a production A→αBβ where FIRST (β) contains € (i.e.,β→€),then everything in FOLLOW(A)is in FOLLOW(B).

Example:

Construct the FIRST and FOLLOW for the grammar:

$A \to BC \mid EFGH \mid H$
$B \to b$
$C \to c \mid \in$
$E \to e \mid \in$
$F \to CE$
$G \to g$
$H \to h \mid \in$

Solution:

1. Finding first () set:
   1. $\text{first}(H) = \text{first}(h) \cup \text{first}(\varepsilon) = \{h, \varepsilon\}$
   2. $\text{first}(G) = \text{first}(g) = \{g\}$
   3. $\text{first}(C) = \text{first}(c) \cup \text{first}(\varepsilon) = c, \varepsilon\}$
   4. $\text{first}(E) = \text{first}(e) \cup \text{first}(\varepsilon) = \{e, \varepsilon\}$
   5. $\text{first}(F) = \text{first}(CE) = (\text{first}(c) - \{\varepsilon\}) \cup \text{first}(E)$

      $= (c, \varepsilon\}\ \{\varepsilon\}) \cup \{e, \varepsilon\} = \{c, e, \varepsilon\}$
   6. $\text{first}(B) = \text{first}(b) = \{b\}$
   7. $\text{first}(A) = \text{first}(BC) \cup \text{first}(EFGH) \cup \text{first}(H)$

      $= \text{first}(B) \cup (\text{first}(E) - \{\varepsilon\}) \cup \text{first}(FGH) \cup \{h, \varepsilon\}$

      $= \{b, h, \varepsilon\} \cup \{e\} \cup (\text{first}(F) - \{\varepsilon\}) \cup \text{first}(GH)$

      $= \{b, e, h, \varepsilon\} \cup \{C, e\} \cup \text{first}(G)$

      $= \{b, c, e, h, \varepsilon\} \cup \{g\} = \{b, c, e, g, h, \varepsilon\}$

2. Finding follow() sets:

1. follow(A) = {\$}
2. follow(B) = first(C) – {ε} ∪ follow(A) = {C, \$}
3. follow(G) = first(H) – {ε} ∪ follow(A)

   ={h, ε} – {ε} ∪ {\$} = {h, \$}
4. follow(H) = follow(A) = {\$}
5. follow(F) = first(GH) – {ε} = {g}
6. follow(E) = first(FGH) m- {ε} ∪ follow(F)

   = ((first(F) – {ε}) ∪ first(GH)) – {ε} ∪ follow(F)

   = {c, e} ∪ {g} ∪ {g} = {c, e, g}
7. follow(C) = follow(A) ∪ first (E) – {ε} ∪ follow (F)

   ={\$} ∪ {e, ε} ∪ {g} = {e, g, \$}

Example 1:

Construct a predictive parsing table for the given grammar or Check whether the given grammar is LL(1) or not.

$E \rightarrow E + T \mid T$
$T \rightarrow T * F \mid F$
$F \rightarrow (E) \mid id$

Step 1:
Suppose if the given grammar is left Recursive then convert the given grammar (and ∈) into non-left Recursive grammar (as it goes to infinite loop).
$E \rightarrow T E^I$
$E^I \rightarrow + T E^I \mid \in$
$T^I \rightarrow F T^I$
$T^I \rightarrow * F T^I \mid \in$
$F \rightarrow (E) \mid id$

Step 2:
Find the FIRST(X) and FOLLOW(X) for all the variables.

The variables are: $\{E, E^I, T, T^I, F\}$
Terminals are: {+, *, (,), id} and \$
Computation of FIRST() sets:

FIRST (F) = FIRST ((E)) U FIRST (id) = {(, id}
FIRST (T^I) = FIRST (*FT^I) U FIRST ($\in$) = {*, $\in$}
FIRST (T) = FIRST (FT^I) = FIRST (F) = {(, id}
FIRST (E^I) = FIRST (+TE^I) U FIRST ($\in$) = {+, $\in$}
FIRST (E) = FIRST (TE^I) = FIRST (T) = {(, id}

Computation of FOLLOW () sets:

	Relevant production
FOLLOW (E) = {\$} U FIRST ( )) = {\$,)}	$F \rightarrow (E)$
FOLLOW (E^I) = FOLLOW (E) = {\$,)}	$E \rightarrow TE^I$
FOLLOW (T) = (FIRST (E^I) - {$\in$}) U FOLLOW (E) U FOLLOW (E^I) = {+,), \$}	$E \rightarrow TE^I$ $E^I \rightarrow +TE^I$
FOLLOW (T^I) = FOLLOW (T) = {+,), \$}	$T \rightarrow FT^I$
FOLLOW (F) = (FIRST (T^I) - {$\in$}) U FOLLOW (T) U FOLLOW (T^I) = {*, +,), \$}	$T \rightarrow T^I$

Step 3:

Construction of parsing table:

Terminals / Variables	+	*	(	)	id	\$
E			$E \rightarrow TE^I$		$E \rightarrow TE^I$	
E^I	$E^I \rightarrow +TE^I$			$E^I \rightarrow \in$		$E^I \rightarrow \in$
T			$T \rightarrow FT^I$		$T \rightarrow FT^I$	
T^I	$T^I \rightarrow \in$	$T^I \rightarrow *FT$		$T^I \rightarrow \in$		$T^I \rightarrow \in$
F			$F \rightarrow (E)$		$F \rightarrow id$	

Table 3.1. Parsing Table

Fill the table with the production on the basis of the FIRST(α). If the input symbol is an $\in$ in FIRST(α), then goto FOLLOW(α) and fill $\alpha \rightarrow \in$, in all those input symbols.

3.1. Let us start with the non-terminal E, FIRST(E) = {(, id}.
So, place the production $E \rightarrow TE^I$ at (and id.

3.2. For the non-terminal E^I, FIRST (E^I) = {+, $\in$}.
So, place the production $E^I \rightarrow +TE^I$ at + and also as there is a $\in$ in FIRST(E^I), see FOLLOW(E^I) = {\$,)}. So write the production $E^I \rightarrow \in$ at the place \$ and).

Similarly:

3.3. For the non-terminal T, FIRST(T) = {(, id}.
So place the production $T \rightarrow FT^I$ at (and id.

3.4. For the non-terminal T^I, FIRST (T^I) = {*, ∈}
So place the production $T^I \rightarrow *FT^I$ at * and also as there is a ∈ in FIRST (T^I), see FOLLOW (T^I) = {+, $,)}, so write the production $T^I \rightarrow \in$ at +, $ and).

3.5. For the non-terminal F, FIRST (F) = {(, id}.
So place the production $F \rightarrow id$ at id location and $F \rightarrow (E)$ at (as it has two productions.

3.6. Finally, make all undefined entries as error.
As these were no multiple entries in the table, hence the given grammar is LL(1).

Step 4:
Moves made by predictive parser on the input id + id * id is:

STACK	INPUT	REMARKS
$ **E**	**id** + id * id $	E and id are not identical; so see E on id in parse table, the production is $E \rightarrow TE^I$; pop E, push E^I and T i.e., move in reverse order.
$ E^I **T**	**id** + id * id $	See T on id the production is $T \rightarrow F\,T^I$; Pop T, push T^I and F; Proceed until both are identical.
$ E^I T^I **F**	**id** + id * id $	$F \rightarrow id$
$ E^I T^I **id**	**id** + id * id $	Identical; pop id and remove id from input symbol.
$ E^I $\mathbf{T^I}$	+ id * id $	See T^I on +; $T^I \rightarrow \in$ so, pop T^I
$ $\mathbf{E^I}$	+ id * id $	See E^I on +; $E^I \rightarrow +T\,E^I$; push E^I , + and T
$ E^I T +	+ id * id $	Identical; pop + and remove + from input symbol.
$ E^I **T**	**id** * id $	
$ E^I T^I **F**	**id** * id $	$T \rightarrow F\,T^I$
$ E^I T^I **id**	**id** * id $	$F \rightarrow id$
$ E^I $\mathbf{T^I}$	* id $	
$ E^I T^I F *	* id $	$T^I \rightarrow *\,F\,T^I$
$ E^I T^I **F**	**id** $	
$ E^I T^I **id**	**id** $	$F \rightarrow id$

$\$ E^I T^I$	$	$T^I \rightarrow \in$
$\$ E^I$	$	$E^I \rightarrow \in$
$	$	Accept.

Table 3.2 Moves made by the parser on input id + id * id

Predictive parser accepts the given input string. We can notice that $ in input and stuck, i.e., both are empty, hence accepted.

2.6.3 LL (1) Grammar:

The first L stands for "Left-to-right scan of input". The second L stands for "Left-most derivation". The '1' stands for "1 token of look ahead".
No LL (1) grammar can be ambiguous or left recursive.

If there were no multiple entries in the Recursive decent parser table, the given grammar is LL (1).

If the grammar G is ambiguous, left recursive then the recursive decent table will have atleast one multiply defined entry.

The weakness of LL(1) (Top-down, predictive) parsing is that, must predict which production to use.

Error Recovery in Predictive Parser:

Error recovery is based on the idea of skipping symbols on the input until a token in a selected set of synchronizing tokens appear. Its effectiveness depends on the choice of synchronizing set. The Usage of FOLLOW and FIRST symbols as synchronizing tokens works reasonably well when expressions are parsed.

For the constructed table., fill with **synch** for rest of the input symbols of FOLLOW set and then fill the rest of the columns with **error** term.

Terminals / Variables	+	*	(	)	id	$
E	error	error	$E \rightarrow TE^I$	synch	$E \rightarrow TE^I$	synch
E^I	$E^I \rightarrow +TE^I$	error	error	$E^I \rightarrow \in$	error	$E^I \rightarrow \in$
T	synch	error	$T \rightarrow FT^I$	synch	$T \rightarrow FT^I$	synch
T^I	$T^I \rightarrow \in$	$T^I \rightarrow *FT$	error	$T^I \rightarrow \in$	error	$T^I \rightarrow \in$
F	synch	synch	$F \rightarrow (E)$	synch	$F \rightarrow id$	synch

Table3.3 :Synchronizing tokens added to parsing table for table 3.1.

If the parser looks up entry in the table as synch, then the non terminal on top of the stack is popped in an attempt to resume parsing. If the token on top of the stack does not match the input symbol, then pop the token from the stack.

The moves of a parser and error recovery on the erroneous input) id*+id is as follows:

STACK	INPUT	REMARKS
\$ **E**	**)** id * + id \$	Error, skip)
\$ **E**	**id** * + id \$	
\$ E^I **T**	**id** * + id \$	
\$ E^I T^I **F**	**id** * + id \$	
\$ E^I T^I **id**	**id** * + id \$	
\$ E^I $\mathbf{T^I}$	* + id \$	
\$ E^I T^IF *	* + id \$	
\$ E^I T^I **F**	+ id \$	Error; F on + is synch; F has been popped.
\$ E^I $\mathbf{T^I}$	+ id \$	
\$ $\mathbf{E^I}$	+ id \$	
\$ E^I T +	+ id \$	
\$ E^I **T**	**id** \$	
\$ E^I T^I **F**	**id** \$	
\$ E^I T^I **id**	**id** \$	
\$ E^I $\mathbf{T^I}$	**\$**	
\$ $\mathbf{E^I}$	**\$**	
\$	**\$**	Accept.

Table 3.4. Parsing and error recovery moves made by predictive parser

Example 2:

Construct a predictive parsing table for the given grammar or Check whether the given grammar is LL(1) or not.

$S \rightarrow iEtSS^I \mid a$
$S^I \rightarrow eS \mid \in$
$E \rightarrow b$

Solution:

1. Computation of First () set:

    1. First (E) = first (b) = {b}
    2. First (S^I) = first (eS) $\cup$ first (ε) = {e, ε}
    3. first (S) = first (iEtSS^I) $\cup$ first (a) = {i, a}

2. Computation of follow() set:
    1. follow (S) = {\$} $\cup$ first (S^I) – {ε} $\cup$ follow (S) $\cup$ follow (S^I)

       = {\$} $\cup$ {e} = {e, \$}
    2. follow (S^I) = follow (S) = {e, \$}
    3. follow (E) = first (tSS^I) = {t}

3. The parsing table for this grammar is:

	a	b	e	i	t	\$
S	$S \rightarrow a$			$S \rightarrow iEtSS^I$		
S^I			$S^I \rightarrow \in$ $S^I \rightarrow eS$			$S^I \rightarrow \in$
E		$E \rightarrow b$				

As the table multiply defined entry. The given grammar is not LL(1).

Example 3:

Construct the FIRST and FOLLOW and predictive parse table for the grammar:

$S \rightarrow AC\$$
$C \rightarrow c \mid \in$
$A \rightarrow aBCd \mid BQ \mid \in$
$B \rightarrow bB \mid d$
$Q \rightarrow q$

Solution:

1. Finding the first () sets:

   First (Q) = {q}

   First (B) = {b, d}

First (C) = {c, ε}

First (A) = First (aBCd) ∪ First (BQ) ∪ First (ε)

= {a} ∪ First (B) ∪ First (d) ∪{ε}

= {a} ∪ First (bB) ∪ First (d) ∪ {ε}

= {a} ∪ {b} ∪ {d} ∪ {ε}

= {a, b, d, ε}

First (S) = First (AC$)

= (First (A) – {ε}) ∪ (First (C) – {ε}) ∪ First (ε)

= ({a, b, d, ε} – {ε}) ∪ ({c, ε} – {ε}) ∪ {ε}

= {a, b, d, c, ε}

2. Finding Follow () sets:

Follow (S) = {#}

Follow (A) = (First (C) – {ε}) ∪ First ($) = ({c, ε} – {ε}) ∪ {$}

Follow (A) = {c, $}

Follow (B) = (First (C) – {ε}) ∪ First (d) ∪ First (Q)

= {c} ∪ {d} ∪ {q} = {c, d, q}

Follow (C) = (First ($) ∪ First (d) = {d, $}

Follow (Q) = (First (A) = {c, $}

3. The parsing table for this grammar is:

	a	b	c	D	q	$	#
S	S→AC$	S→AC $	S→AC $	S→AC $		S→AC $	
A	A→aBCd	A→BQ	A→ε	A→BQ		A→ε	
B		B→bB		B→d			
C			C→c	C→ ε		C→ε	
Q					Q→q		

4. Moves made by predictive parser on the input abdcdc$ is:

Stack symbol	Input	Remarks
#S	abdcdc$#	S → AC$
#$CA	abdcdc$#	A → aBCd
#$CdCBa	abdcdc$#	Pop a
#$CdCB	bdcdc$#	B → bB
#$CdCBb	bdcdc$#	Pop b
#$CdCB	dcdc$#	B → d
#$CdCd	dcdc$#	Pop d
#$CdC	cdc$#	C → c
#$Cdc	cdc$#	Pop C
#$Cd	dc$#	Pop d
#$C	c$#	C → c
#$c	c$#	Pop c
#$	$#	Pop $
#	#	Accepted