LANGUAGE PROCESSORS

Introduction to Language processor:

A program that performs task such as translating and interpreting required for processing a specified programming language. The different types of language processors are

Assemblers- A language translator whose source language is assembly language and the target language is machine language.

Compiler- A language translator whose source language is a high level language(c, c++, java etc) and the target language is low level language (machine language or assembly language)

Detranslator:-converts machine language to assembly level language.

Interpreter: - Interpreter takes one statement of a high level language at a time and translates it into a machine instruction. Interpreters are easy to write and they do not require large memory space in the computer. The main disadvantage of interpreter is that they require more time to execute in the computer.

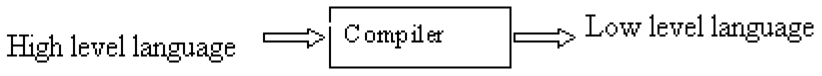
Module 2:Compiler

Translator:

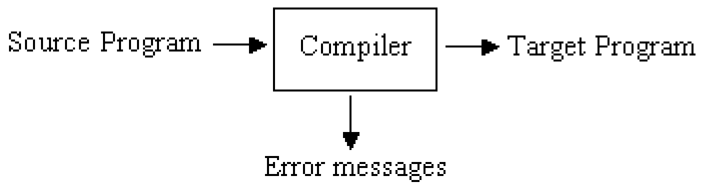
A translator is a program that takes as input a program written in one programming language(source language) and produces as output a program in another language(object or target language).eg:compiler,assembler.

Compiler:

A compiler is a program that accepts a program written in a high level language and produces output in low level language.



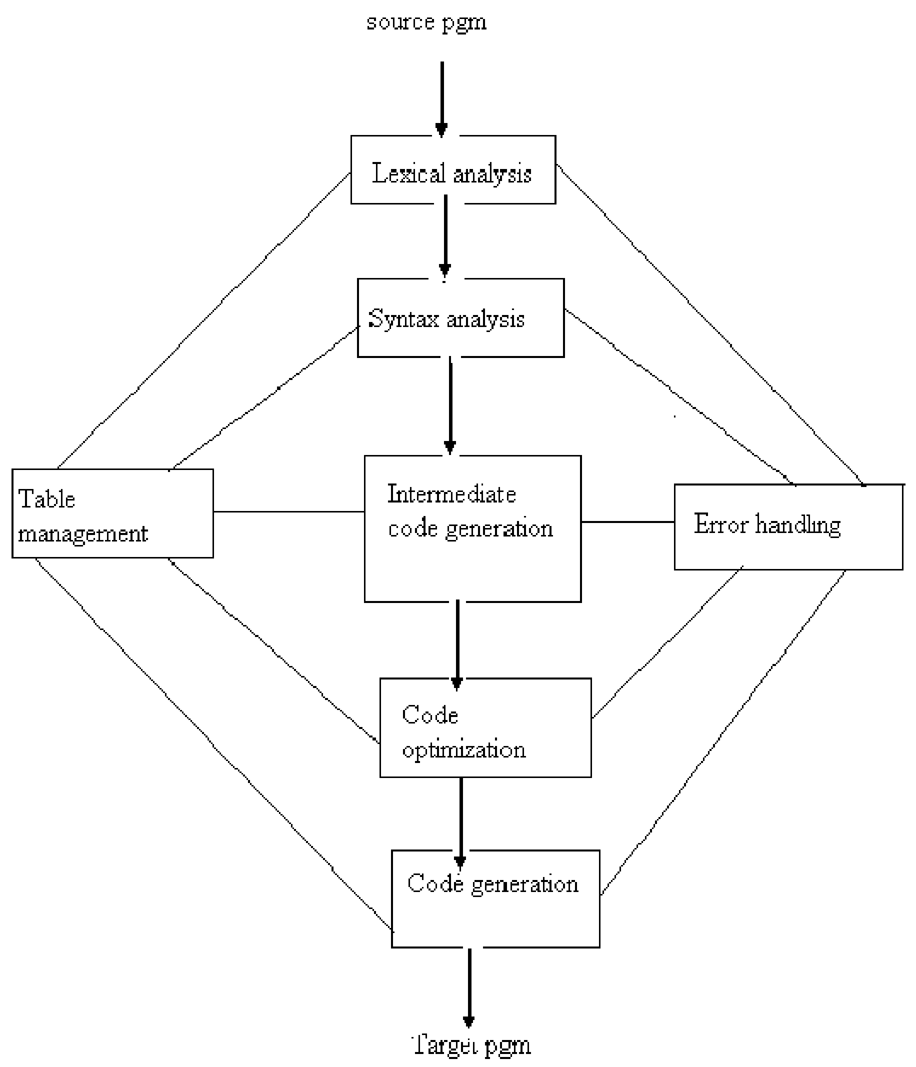
Programming languages are just notations for describing computations. So, before execution, they have to be converted to the machine understandable form – the machine language. This translation is done by the compiler. The translation process should also report the presence of errors in the source program. This can be diagrammatically represented as



An interpreter is similar to a compiler, except that it directly executes the program with the supplied inputs to give the output. Usually, compiler is faster than interpreter, but the interpreter has better diagnostics, since the execution is step – by – step. Java uses a hybrid compiler.

The Structure of a Compiler:

A Compiler takes as input a soure program and produces as output an equivalent sequence of machine instructions. This process is so complex that it is not reasonable, either from a logical point of view or from an implementation point of view, to consider the compilation process as occurring in one single step. For this reason, it is customary to partition the compilation process into a series of sub processes called phases. A phase is a logically cohesive operation that takes as input one representation of the source program and produces as output another representation.



Lexical Analysis

The first phase, called lexical analyzer or scanner, separates characters of the source language into groups that logically belong together, these groups are called tokens. The usual tokens are keywords, identifiers, operators, punctuation symbols. The output of the lexical analyzer is a stream of tokens which is passed to the next phase, the syntax analyzer or parser. White space and comments are ignored. The scanner produce error messages. It also stores the information in the symbol table.

Syntax Analysis

This is also called parsing. The syntax analyzer groups tokens together into syntactic structures and a parse tree is generated. For example, the three tokens representing A+B might be grouped into a syntactic structure called an expression. Syntactical errors are determined with the help of this parse tree.

Intermediate Code Generation

The intermediate code generator uses the structure produced by the syntax analyzer to create a stream of simple instructions. Examples for intermediate codes are three address codes, postfix notations etc.The primary difference between intermediate code and assembly code is that the intermediate code need not specify the registers to be used for each operation.

Code Optimization

Code optimization is the process of modifying a intermediate code to improve its efficiency so that the object program runs faster or takes less space.Its output is another intermediate program.

Code Generation

This phase generates the target code. Allocating memory for each variables, translating intermediate instruction into machine instruction etc. are functions of this phase.

Table Management

Table management or book keeping keeps track of the names used by the program and records essential information about each such as its type. The data structure used to record this information is called a symbol table.

Error handling

The error handler is invoked when an error in the source program is detected. Both the table management and error handling routines interact with all phases of the compiler.

An example showing the various phases by which an arithmetic expression translated into a machine code is given below.

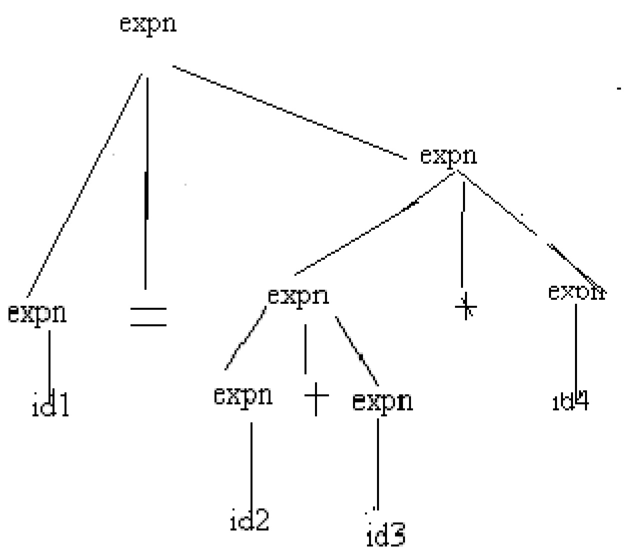
S=A+B*C (high level language statement)

$\downarrow$

Id1 AssinOp id2 PlusOp id3 MultiplyOp id4 (Token Stream produced by the scanner)

$\downarrow$

(Parse Tree produced by the parser)



$\downarrow$

(Intermediate code generated by intermediate code generator)

T1=id2+id3
T2=T1*id4
T3=T1+T2
id1=T3

$\downarrow$

(Optimized code generated by Code Optimizer)

T1=id2+id3
T2=T1*id4
id1=T2

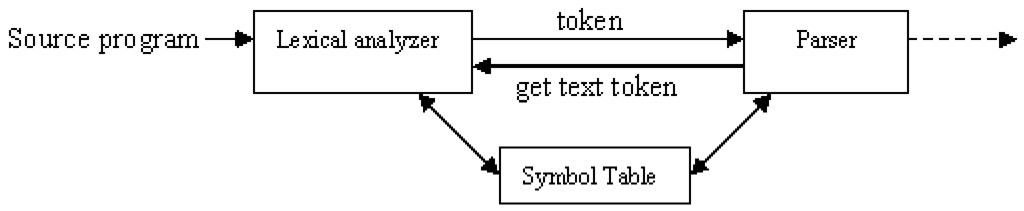
$\downarrow$

(Target code generated by Code Generator)

LOAD R1, A
ADD R1, B
MUL R1, C
STORE..R1

Lexical Analysis

The lexical analyzer is the interface between the source program and the compiler. The lexical analyzer reads the source program one character at a time, carving the source program into a sequence of atomic units called tokens. Each token represents a sequence of characters that can be treated as a single logical entity.Identifiers, keywords, constants, operators and punctuation symbols are typical tokens. The scanner produce error messages. It also stores the information in the symbol table. The purpose of producing these tokens is usually to forward them as input to another program, such as a parser. The block diagram of a lexical analyzer is given below.



For example in FORTRAN statement

IF (5.EO.MAX) GO TO 100 ---- (1)

We find the following eight tokens: IF; (; .EQ. MAX ;) ; GOTO; 100.

There are two kinds of token:
1. Specific strings (IF or a semicolon)
2. Classes of strings (identifiers, constants or labels)

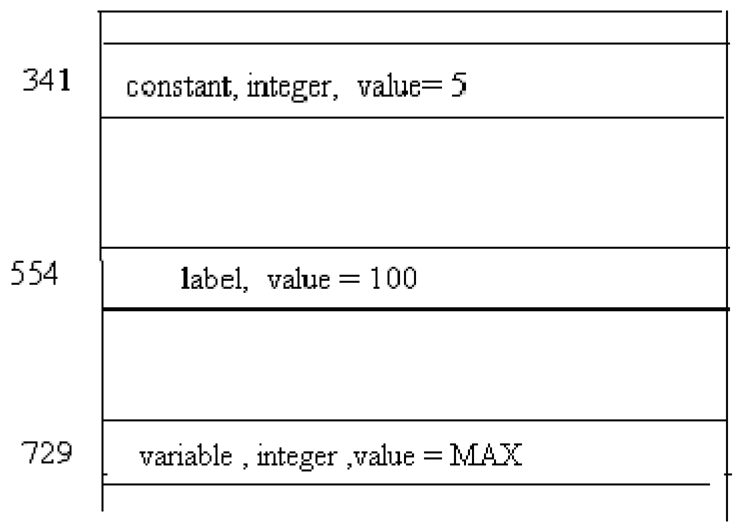
A token consists of two parts, a token type and token value. A token consisting of a specific string such as a semicolon will be treated as having a type(string) but no value. A token such as identifier MAX has a type “identifier” and a value consisting of the string MAX.
The lexical analyzer passes the two components of the token to the parser. The first is a code for the token type (identifier), and the second is the value, a pointer to the place in the symbol table reserved for the specific value found.

When statement (1) is completely processed by the lexical analyzer, the token stream might look like

if ([**const**,341] **eq** [**id**,729]) **goto** [**label**,554]

The tokens having an associated value are represented by pairs in square brackets. The second component of the pair can be interpreted as an index into the symbol table where the information about constants, variables, and labels is kept.

The relevant entries of the symbol table are suggested in Fig:



Symblo table

Syntax Analysis

Syntax analysis is a process in compilers recognizes the structure of programming languages. It is also known as parsing. Context-free grammar is usually used for describing the structure of languages. A parser

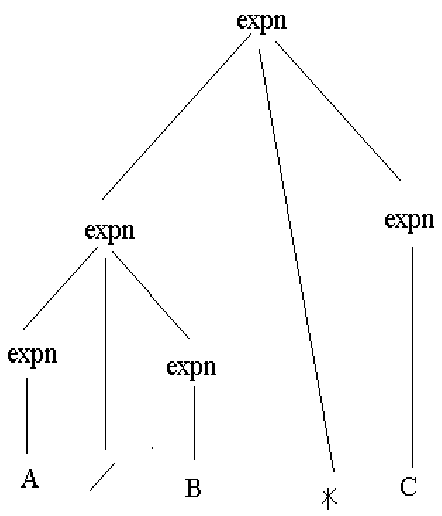
- Detects and reports any syntactical errors
- Collect information into symbol table
- Produce a parse tree from which intermediate code can be generated

For example, the expression A / B * C has two possible interpretations:

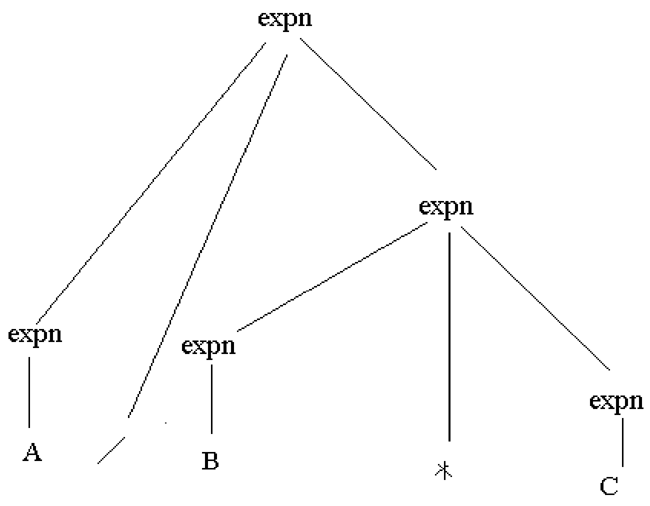
a) Divide A by B and then multiply by C (as in FORTRAN); or
b) Multiply B by C and then use the result to divide A (as in APL)

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

a)



b)



Context-Free Grammar

For the syntactic specification of a programming language we shall use a notation called context free grammar, which is also called **BNF (Backus-Naur Form)**. Context-free grammars are powerful enough to describe the syntax of most programming languages. This notation has a number of significant advantages as a method of specification for the syntax of a language.

- A grammar gives an efficient, yet easy to understand, syntactic specification for the programs of a particular programming language.
- An efficient parser can be constructed automatically from a properly designed grammar.

- A grammar imparts a structure to a program that is useful for its translation into object code and for the detection of errors.

  For example,

  1) If S1 and S2 are statements and E is an expression, then
     "if E then S1 else S2" is a statement

  2) If S1, S2……..Sn are statements, then
     "begin S1; S2; ……..Sn; end"

  3) If E1 and E2 are expressions, then "E1+E2" is an expression.

  If we use the syntactic category "statement" to denote the class of statements and "expression" to denote the class of expressions, then eg: 1 can be expressed by the rewriting or production

  statement→ if expression then statement else statement

  Similarly eg: 3 can be written as

  expression → expression + expression

  To express eg: 2 by rewriting rules, we can introduce a new syntactic category "statement-list" denoting any sequence of statements separated by semicolons. Then eg: 2 becomes

  statement→ begin statement-list end

  In general a grammar involves four quantities:
  1) Terminals
  2) Non-terminals
  3) Start symbol
  4) Productions

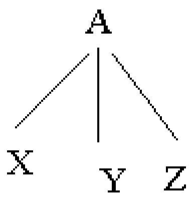
  The basic symbols of which strings in the language are composed is called terminals. In the above eg: certain keywords such as begin and end are terminals.
  Non-terminals are special symbols that denote set of strings. In the above eg: the syntactic categories such as statement, expression, and statement-list are non-terminals.
  One non-terminal symbol is selected as a start symbol and it denotes the language in which we can truly interested.

  The production or rewriting rules define the ways in which the syntactic category may be build up from one another and from the terminals.Each production consists of a non-terminal and followed by an arrow followed by a string of non-terminals and terminals.

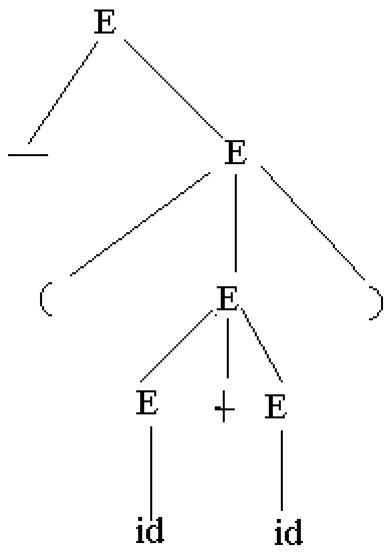
Eg: statement→ begin statement-list end

Parse Tree:

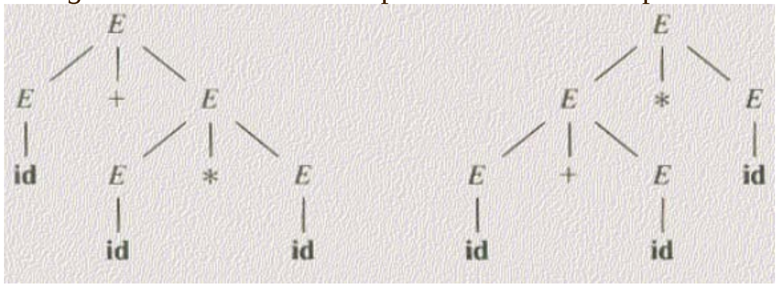
A parse tree is a tree that represents the syntactic structure of a string. If the leaves of the trees are traversed from left to right. The leaves contain the terminal symbols, and the internal nodes contain the non-terminal symbols. For example, if A→XYZ is a production, then the parse tree for that derivation will have the subtree



- (id + id)



A grammar that produces more than one parse tree for some sentence is said to be *ambiguous.* Given below are parse trees for the input "id+id*id"



Basic Parsing Techniques:

The aim of parsing is to determine the validity of a source string. If the string is valid the parser built a parse tree for that string.

Parsing is of 2 types:

- Top-down
- Bottom-up

Bottom-up parsers build parse tree from the bottom to the root, while top down parsers start with the root and work down to the leaves. In both cases the input to the parser is being scanned from left to right, one symbol at a time.

Bottom-up Parsing:

Shift Reduce parser
This uses the bottom up style of parsing. It attempts to construct a parse tree for an input string beginning at the leaves and working towards the root. It uses a stack to implement the parser. The stack is empty at the beginning of the parse and will contain the start symbol at the end of a successful parse.

For example, consider the grammar

S→aAcBe
A→Ab\ b
B→d

And the string abbcde.We want to reduce this string to S.We scan abbcde looking for substrings that match the right side of the production. The substrings b and d qualify. Let us choose the leftmost b and replace it by A, the left side of the production A→b.We obtain the string aAbcde.We now find that Ab, b, and d each match the right side of some production. We choose to replace the substring Ab by A, the left side of the production A→Ab.We now obtain aAcde.Then replacing d by B,the left side of the production B→d,we obtain aAcBe.We can now replace this entire string by S.

Stack Implementation of shift Reduce Parsing:

A convenient way to implement a parser is to use a stack and an input buffer. We shall use the '\$' (dollar) to mark the bottom of the stack and the right end of the input.

Stack	Input
\$	w\$

The parser operates by shifting zero or more input symbols onto the stack until a handle β is on top of the stack. The parser then reduces β to the left side of the appropriate production. The parser repeats this cycle until it has detected an error or until the stack contains the start symbol and its input is empty.

Stack	Input
\$S	\$

In this configuration the parser halts and announces successful completion of parsing.

Eg:Let us step through the actions a shift reduce parser might make in parsing the input: id1+id2*id3 according to the grammar

1) E->E+E
2) E->E*E
3) E->(E)
4) E-> id   and using the derivation

$E \rightarrow \underline{E + E}$
$\rightarrow E + \underline{E * E}$
$\rightarrow E + E * \underline{id3}$
$\rightarrow E + \underline{id2} * id3$
$\rightarrow \underline{id1} + id2 * id3$

The parsing steps are:

Step	Stack	Input	Action
1	\$	id1+id2*id3\$	Shift
2	\$id1	+id2*id3\$	Reduce by E -> id
3	\$E	+id2*id3\$	Shift
4	\$E +	id2*id3\$	Shift
5	\$E +id2	*id3\$	Reduce by E -> id
6	\$E + E	*id3\$	Shift
7	\$E + E *	id3\$	Shift
8	\$E + E * id3	\$	Reduce by E ->id
9	\$E + E * E	\$	Reduce by E -> E *E
10	\$E + E	\$	Reduce by E -> E + E
11	\$E	\$	Accept

While the primary operations of the parser are shift and reduce, there are actually four possible actions a shift reduce parser can make: 1) shift 2) reduce 3) accept 4) error.

1. In a shift action, the next input symbol is shifted to the top of the stack.
2. In a reduce action, the parser knows the right end of the handle is at the top of the stack. It must then locate the left end of the handle within the stack and decide with what nonterminal to replace the handle.
3. In an accept action, the parser announces successful completion of parsing.
4. In an error action, the parser discovers that a syntax error has occurred and calls an error recovery routine.

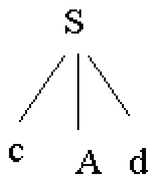
Top-down parsing
In top-down parsing, traversal occurs from the top to leaves. Top down parsing can be viewed as an attempt to find a left most derivation of an input string. This is of 2 forms:

- Backtracking
- Predictive

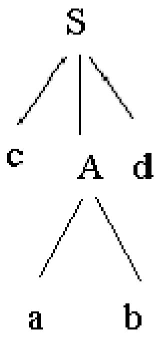
Backtracking involves reading the input token stream many times. So, this is a time-consuming method. So, we usually go for predictive ones. A predictive parser is a recursive descent parser with no backtracking.
For example, consider the grammar

S→cAd
A→ab \ a   and the input w=cad.

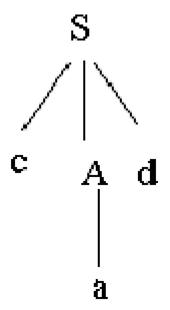
To construct a parse tree for this sentence top down, we initially create a tree consisting of a single node labeled S. An input pointer points to c, the first symbol of w.We then use the first production for S to expand the tree and obtain



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 alternate for A to obtain the tree



We now have a match for the second input symbol. We now consider d, the third input symbol, and the next leaf, labeled b.Since b does not match d, we report failure and go back to A to see whether there is another alternate for A that we have not tried but which might produce a match.
In going back to A we must reset the input pointer to position 2,the position it had when we first came to A.We now try second alternate for A to obtain the tree



The leaf a matches the second symbol of w and the leaf d matches the third symbol. Since we have now produced a parse tree for w, we halt and announce successful completion of parsing.

<u>Recursive procedures for top down parsing</u>

```
procedure S ( );
begin
         if input symbol = ‘c’ then
           begin
               ADVANCE ( );
               if A ( ) then
                   if input symbol =‘d’   then
                    begin ADVANCE ( ); return true end
                end
           return false
 end
```

(a) procedure S

```
procedure A ( );
begin
isave: =input pointer;
if input symbol = ‘a’ then
  begin
       ADVANCE ( );
        if input symbol = ‘b’ then
         begin ADVANCE ( ); return true end
          end
    input pointer:=isave;
```

```
/*failure to find ab */
if input symbol = 'a'        then
   begin ADVANCE ( ); return true end
 else return false
end
```

(b) Procedure A

Left Recursion

A grammar G is said to be left recursive, if it has a non-terminal A such that there is a derivation A =>Aα for some α.A left recursive grammar can cause a top down parser to go into an infinite loop.

Elimination of left recursion

If we have the left recursive pair of productions $A \rightarrow A\alpha / \beta$, where β does not begin with an A, then we can eliminate the left recursion by replacing this pair of productions with

$A \rightarrow \beta A^{|}$

$A^{|} \rightarrow \alpha A^{|} / £$

Example: Consider the following grammar

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid id$

Eliminating the left-recursion, we obtain

$E \rightarrow TE^{|}$

$E^{|} \rightarrow + TE^{|} \mid £$

$T \rightarrow FT^{|}$

$T^{|} \rightarrow * FT^{|} \mid £$

$F \rightarrow (E) \mid id$

Recursive Descent Parsing

A parser that uses a set of recursive procedures to recognize its input with no back tracking is called a recursive descent parser. It is a commonly used predictive parser.

Recursive procedures for recursive descent parsing

```
procedure E ( );
begin
        T ( );
         EPRIME ( );
end;
procedure EPRIME ( );
if input symbol = ‘+’ then
     begin
         ADVANCE ( );
          T ( );
           EPRIME ( );
      end;
procedure T ( );
begin
         F ( );
          TPRIME ( );
end;
procedure TPRIME ( );
if input symbol = ‘*’ then
    begin
         ADVANCE ( );
          F ( );
           TPRIME ( );
    end;
procedure F ( );
if input-symbol = ’id’ then
     ADVANCE ( );
else if input-symbol = ‘(’ then
    begin
          ADVANCE ( );
           E ( );
            if input-symbol = ‘)’ then
                ADVANCE ( );
            else ERROR ( );
     end
else ERROR ( );
```