UNIT - 4

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 HLL specification would be detected and reported to the programmers. Important role of translator are:

1 Translating the HLL program input into an equivalent machine language program.
2 Providing diagnostic messages wherever the programmer violates specification of the HLL.

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 HLL specification would be detected and reported to the programmers. Important role of translator are:

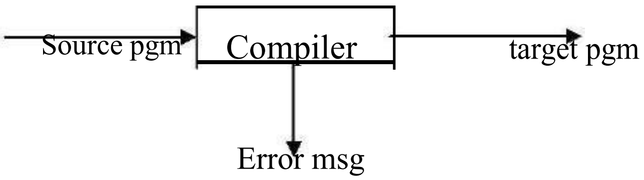
1 Translating the hll program input into an equivalent ml program.
2 Providing diagnostic messages wherever the programmer violates specification of the hll.

TYPE OF TRANSLATORS:-

a. **Compiler**
b. **Interpreter**
c. **Preprocessor**

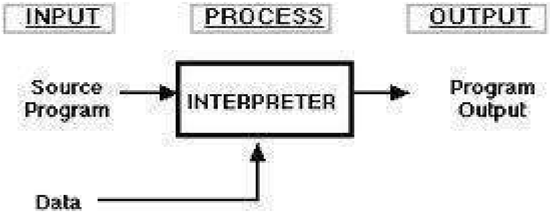
Compiler

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



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

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. Synatx analysis
3. Semantic analysis
4. Direct Execution

Advantages:

Modification of user program can be easily made and implemented as execution proceeds.

5. Type of object that denotes a 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 the program is *slower*.

Memory consumption is more.

OVERVIEW OF LANGUAGE PROCESSING SYSTEM

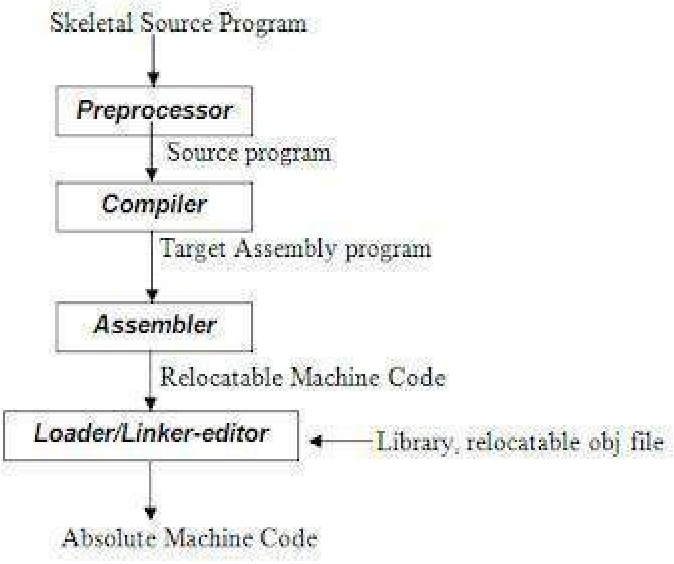


Fig 1.1 Language –processing System

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

Assembler*:* 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).

Loader and Link-editor:

Once the assembler procedures an object program, that program must be placed 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 othey may be placed in arbitrary core locations is called relocation.

STRUCTURE OF A COMPILER

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 p**hases'**.

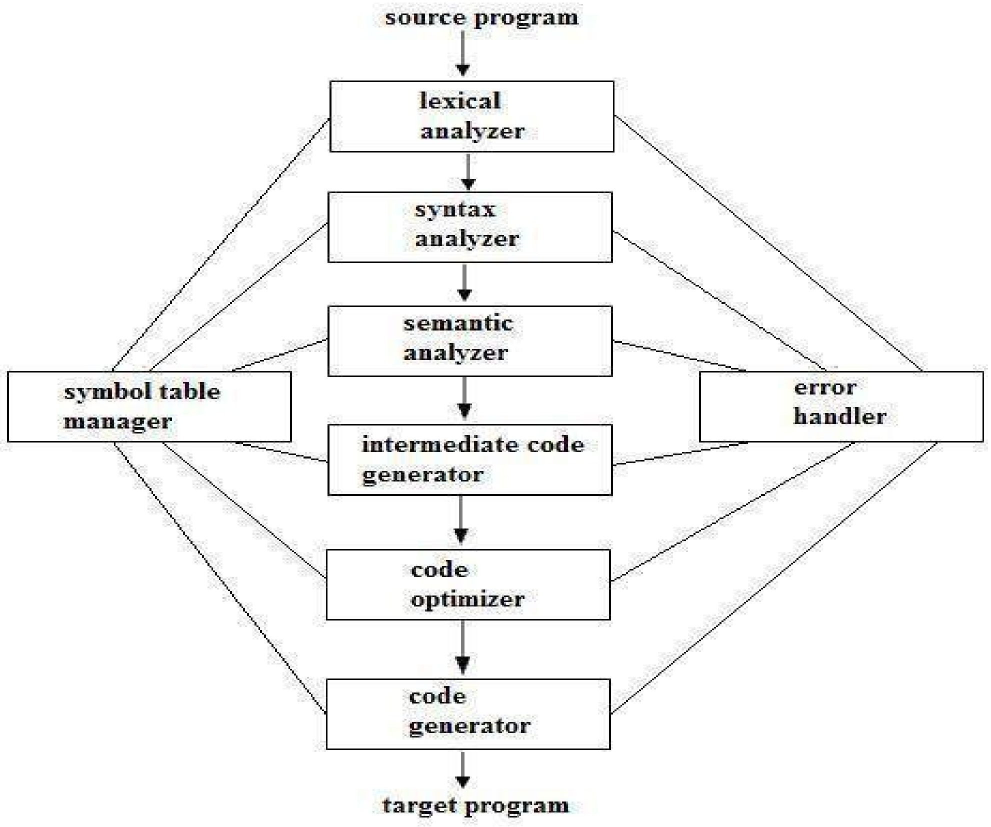


Fig 1.5 Phases of a compiler

Lexical Analysis:-

LA or Scanners reads the source program one character at a time, carving the source program into a sequence of automic 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 some 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.**

2, 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 :-

Code Generator produces the object code by deciding on the memory locations for data, selecting code to access each datum 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.

Table Management OR Book-keeping :-

A compiler needs to collect information about all the data objects that appear in the source program. The information about data objects is collected by the early phases of the compiler-lexical and syntactic analyzers. The data structure used to record this information is called as Symbol Table.

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.

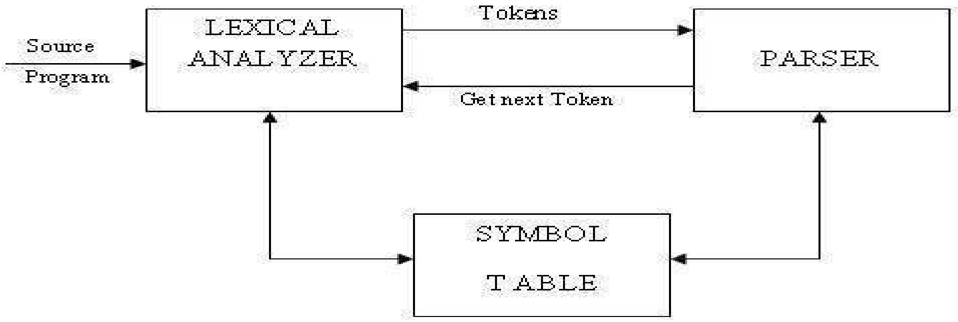
LEXICAL ANALYSIS

OVER VIEW OF LEXICAL ANALYSIS

- To identify the tokens we need some method of describing the possible tokens that can appear in the input stream. For this purpose we introduce regular expression, a notation that can be used to describe essentially all the tokens of programming language.
- Secondly , having decided what the tokens are, we need some mechanism to recognize these in the input stream. This is done by the token recognizers, which are designed using transition diagrams and finite automata.

ROLE OF LEXICAL ANALYZER

the LA is the first phase of a compiler. It main task is to read the input character and produce as output a sequence of tokens that the parser uses for syntax analysis.



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.

LEXICAL 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).

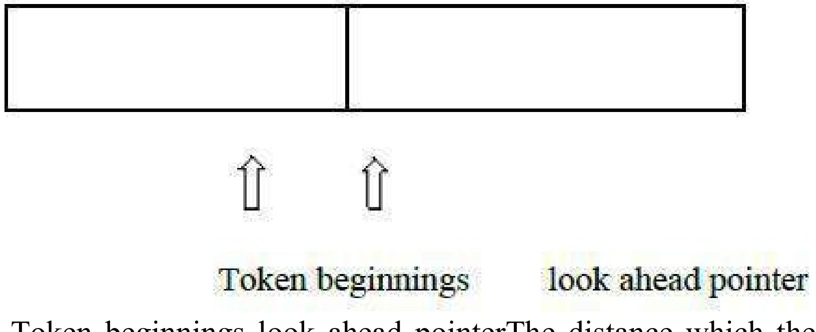
INPUT BUFFERING

The LA scans the characters of the source pgm 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 thelexical analyzer to read its input from an input buffer. Figure shows a buffer divided into two haves 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 thecharacter next to be read. In practice each buffering scheme adopts one convention either apointer is at the symbol last read or the symbol it is ready to read.



Token beginnings look ahead pointerThe distance which the lookahead pointer may have to travel past the actual token may belarge. 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. In either case, the token itself ends at the second E. If the look ahead pointer travels beyond the buffer half in which it began, the other half must be loaded with the next characters from the source file. Since the buffer shown in above figure is of limited size there is an implied constraint on how much look ahead can be used before the next token is discovered. In the above example, ifthe look ahead traveled to the left half and all the way through the left half to the middle, we could not reload the right half, because we would lose characters that had not yet been groupedinto tokens. While we can make the buffer larger if we chose or use another buffering scheme,we cannot ignore the fact that overhead is limited.

TOKEN, 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

Recognition of tokens:

We learn how to express pattern using regular expressions. Now, we must study how to take the patterns for all the needed tokens and build a piece of code that examins the input string and finds a prefix that is a lexeme matching one of the patterns.

```
 Stmt  ->   if expr then stmt
           | If expr then else stmt
           | є
Expr -->  term relop term
          |term

Term -->id
```

For relop ,we use the comparison operations of languages like Pascal or SQL where = is "equals" and < > is "not equals" because it presents an interesting structure of lexemes. The terminal of grammar, which are if, then , else, relop ,id and numbers are the names of tokens as far as the lexical analyzer is concerned, the patterns for the tokens are described using regular definitions.

```
digit      -->[0,9]
digits     -->digit+
number -->digit(.digit)?(e.[+-]?digits)?
letter      -->[A-Z,a-z]
 id         -->letter(letter/digit)*
 if          --> if
 then       -->then
 else       -->else
relop      --></>/<=/>=/==/< >
```

In addition, we assign the lexical analyzer the job stripping out white space, by recognizing the "token" we defined by:

ws --> (blank/tab/newline)$^+$

Here, blank, tab and newline are abstract symbols that we use to express the ASCII characters of the same names. Token ws is different from the other tokens in that ,when we recognize it, we do not return it to parser ,but rather restart the lexical analysis from the character that follows the white space . It is the following token that gets returned to the parser.

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

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.

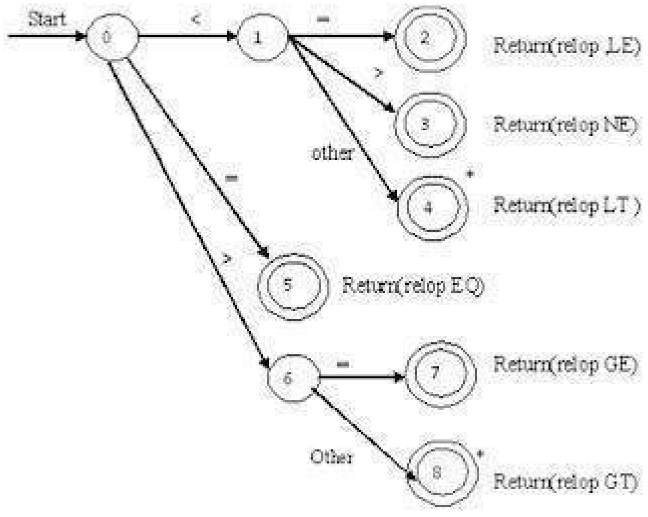
If we are in one state s, and the next input symbol is a, we look for an edge out of state s labeled by a. if we find such an edge ,we advance the forward pointer and enter the state of the transition diagram to which that edge leads.

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.

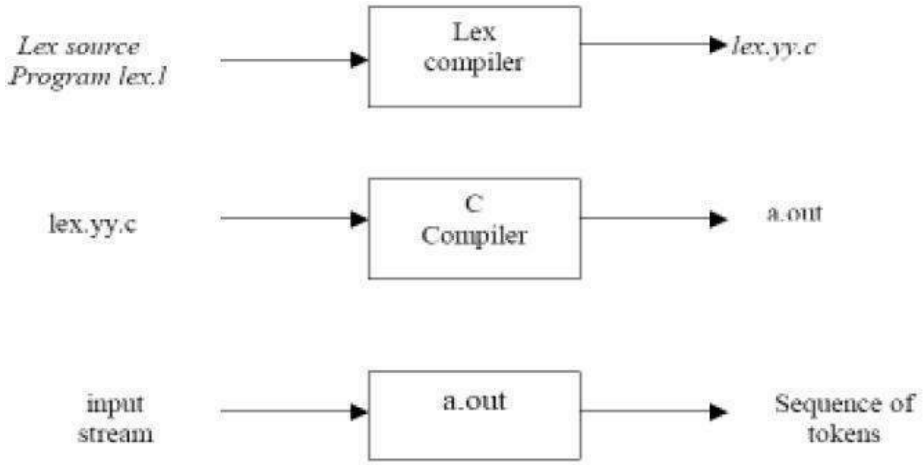
2. In addition, if it is necessary to return the forward pointer one position, then we shall additionally place a * near that accepting state.

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.

Creating a lexical analyzer with Lex



Lex specifications:

A Lex program (the .l file) consists of three parts:

declarations
%%
translation rules
%%
auxiliary procedures

1. The *declarations* section includes declarations of variables,manifest constants(A manifest constant is an identifier that is declared to represent a constant e.g. *# define PIE 3.14*), and regular definitions.
2. The *translation rules* of a Lex program are statements of the form :

   p1 *{action 1}*
   p2 *{action 2}*
   p3 *{action 3}*
   ... ...
   ... ...

   where each *p* is a regular expression and each *action* is a program fragment describing what action the lexical analyzer should take when a pattern *p* matches a lexeme. In Lex the actions are written in C.

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

Note: You can refer to a sample lex program given in page no. 109 of chapter 3 of the book: *Compilers: Principles, Techniques, and Tools* by Aho, Sethi & Ullman for more clarity.

.

SYNTAX ANALYSIS

ROLE OF THE PARSER

Parser obtains a string of tokens from the lexical analyzer and verifies that it can be generated by the language for the source program. The parser should report any syntax errors in an intelligible fashion. The two types of parsers employed are:

1.Top down parser: which build parse trees from top(root) to bottom(leaves)
2.Bottom up parser: which build parse trees from leaves and work up the root.

Therefore there are two types of parsing methods– top-down parsing and bottom-up parsing

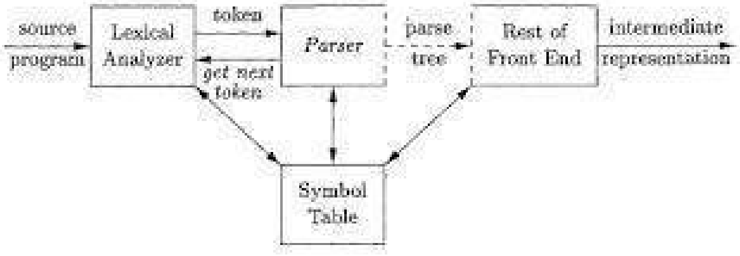


Figure 4.1: Position of parser in compiler model

Context free Grammars(CFG)

CFG is used to specify the syntax of a language. A grammar naturally describes the hierarchical structure of most programming language constructs.

Formal Definition of Grammars

A context-free grammar has four components:

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 non-terminal represents a set of strings of terminals, in a manner we shall describe.
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 and1or 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 non terminals as the *start* symbol.

Production is for a non terminal if the non terminal is the head of the production. A string of terminals is a sequence of zero or more terminals. The string of zero terminals, written as ***E*,** is called the empty string.

Derivations

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

Leftmost and Rightmost Derivation of a String

- **Leftmost derivation** − A leftmost derivation is obtained by applying production to the leftmost variable in each step.
- **Rightmost derivation** − A rightmost derivation is obtained by applying production to the rightmost variable in each step.
- **Example**

Let any set of production rules in a CFG be

X → X+X | X*X |X| a

over an alphabet {a}.

The leftmost derivation for the string **"a+a*a"** is

X → X+X → a+X → a + X*X → a+a*X → a+a*a

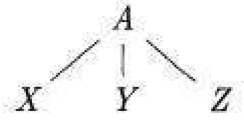
The rightmost derivation for the above string **"a+a*a"** is

X → X*X → X*a → X+X*a → X+a*a → a+a*a

Derivation or Yield of a Tree

The derivation or the yield of a parse tree is the final string obtained by concatenating the labels of the leaves of the tree from left to right, ignoring the Nulls. However, if all the leaves are Null, derivation is Null.

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:



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 e.
3. Each interior node is labeled by a nonterminal

If A is the nonterminal labeling some interior node and X I , Xz, . . . ,Xn are the labels of the children of that node from left to right, then there must be a production A → X1X2 . . Xn . Here, X1, X2,. . . , Xn, each stand for a symbol that is either a terminal or a nonterminal. As a special case, if A → **c** is a production, then a node labeled A may have a single child labeled ***E***

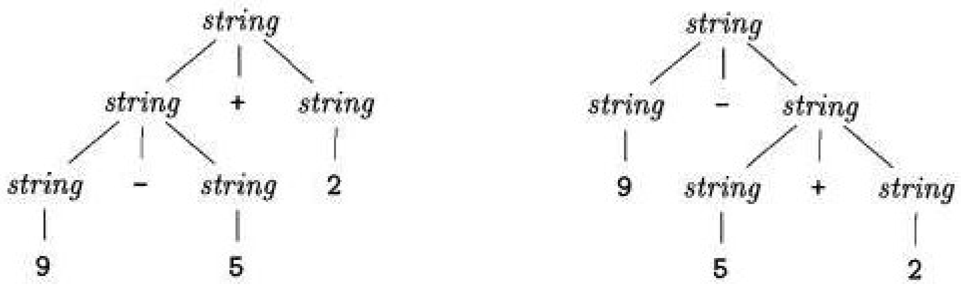
Ambiguity

A grammar can have more than one parse tree generating a given string of terminals. Such a grammar is said to be *ambiguous.* To show that a grammar is ambiguous, all we need to do is find a terminal string that is the yield of more than one parse tree. 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.

Example :: Suppose we used a single nonterminal *string* and did not distinguish between digits and lists,

$$string \rightarrow string + string \mid string - string \mid 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$

Fig. shows that an expression like 9-5+2 has more than one parse tree with this grammar. The two trees for 9-5+2 correspond to the two ways of parenthesizing the expression: (9-5) +2 and 9- (5+2) . This second parenthesization gives the expression the unexpected value 2 rather than the customary value 6.



Two parse trees for 9-5+2

Verifying the language generated by a grammar

The set of all strings that can be derived from a grammar is said to be the language generated from that grammar. A language generated by a grammar **G** is a subset formally defined by

L(G)={W|W ∈ ∑*, S ⇒G **W**}

If **L(G1) = L(G2)**, the Grammar **G1** is equivalent to the Grammar **G2**.

Example

If there is a grammar

G: N = {S, A, B} T = {a, b} P = {S → AB, A → a, B → b}

Here **S** produces **AB**, and we can replace **A** by **a**, and **B** by **b**. Here, the only accepted string is **ab**, i.e.,

L(G) = {ab}

Writing a grammar

A *grammar* consists of a number of *productions*. Each production has an abstract symbol called a *nonterminal* as its *left-hand side*, and a sequence of one or more nonterminal and *terminal* symbols as its *right-hand side*. For each grammar, the terminal symbols are drawn from a specified *alphabet*.

Starting from a sentence consisting of a single distinguished nonterminal, called the *goal symbol*, a given context-free grammar specifies a *language*, namely, the set of possible sequences of terminal symbols that can result from repeatedly replacing any nonterminal in the sequence with a right-hand side of a production for which the nonterminal is the left-hand side.

There are **four categories** in writing a grammar :

1. Lexical Vs Syntax Analysis
2. Eliminating ambiguous grammar.
3. Eliminating left-recursion
4. Left-factoring.

Each parsing method can handle grammars only of a certain form hence, the initial grammar may have to be rewritten to make it parsable

1. Lexical Vs Syntax Analysis

Reasons for using the regular expression to define the lexical syntax of a language

a) Regular expressions provide a more concise and easier to understand notation for tokens than grammars.

b) The lexical rules of a language are simple and to describe them, we donot need notation as powerful as grammars.

c) Efficient lexical analyzers can be constructed automatically from RE than from grammars.

d) Separating the syntactic structure of a language into lexical and nonlexical parts provides a convenient way of modularizing the front end into two manageable-sized components.

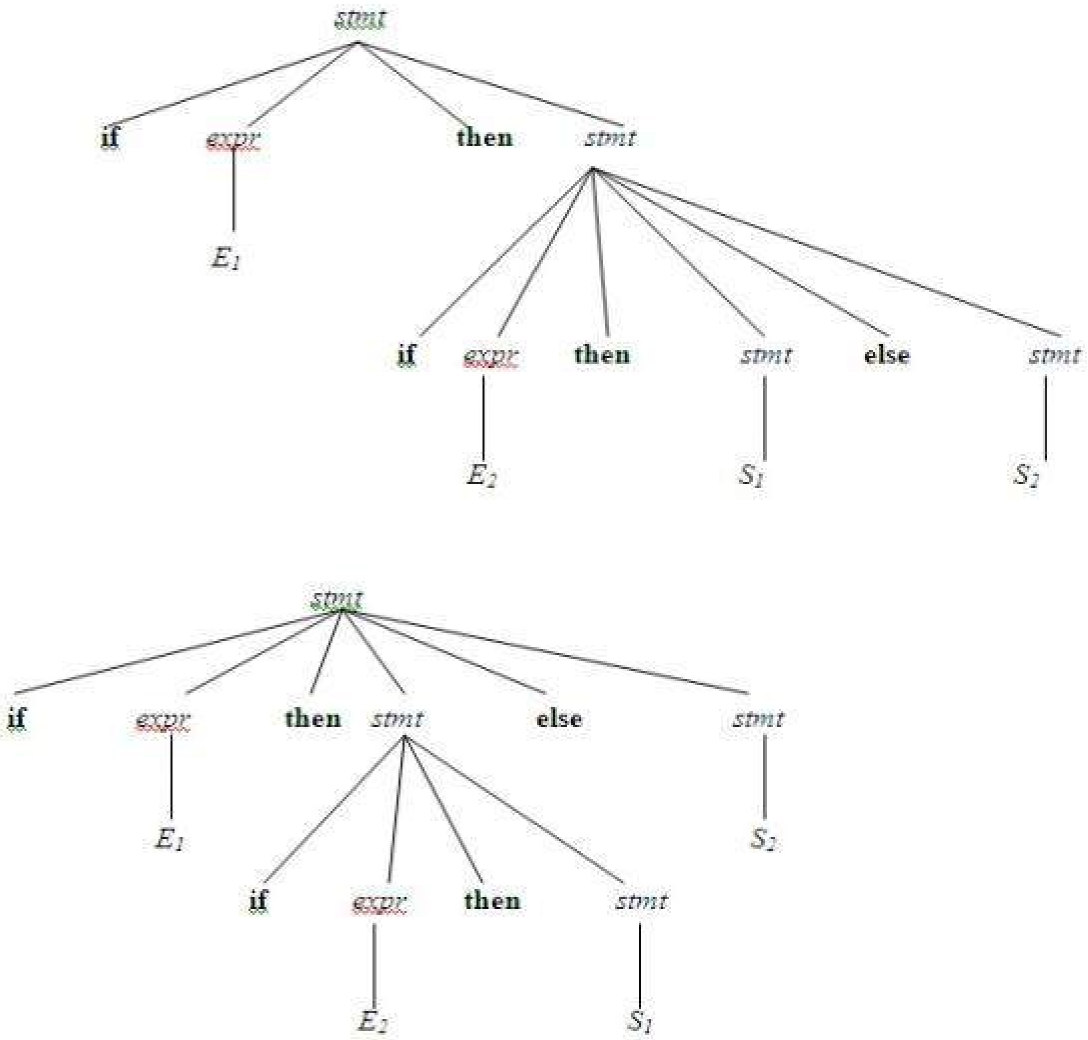
2. Eliminating ambiguous grammar.

Ambiguity of the grammar that produces more than one parse tree for leftmost or rightmost derivation can be eliminated by re-writing the grammar.

Consider this example,

G: *stmt*→**if** *expr* **then** *stmt*
|**if** *expr* **then** *stmt* **else** *stmt*
|**other**

This grammar is ambiguous since the string **if E1 then if E2 then S1 else S2 has the** following two parse trees for leftmost derivation



Two parse trees for an ambiguous sentence

The general rule is “Match each else with the closest unmatched then.This disambiguating rule can be used directly in the grammar,

To eliminate ambiguity, the following grammar may be used:

stmt→matched* | *unmatchedstmt

matched→**if** *expr stmt* **then** *matched* **else** *matchedstmt* | **other**

unmatched→ **if** *expr* **then** *stmt* | **if** *expr* **then** *matched* **else** *unmatchedstmt*

3. Eliminating left-recursion

Because we try to generate a leftmost derivation by scanning the input from left to right, grammars of the form A → A x may cause endless recursion.Such grammars are called left-recursive and they must be transformed if we want to use a top-down parser.

- A grammar is left recursive if for a non-terminal A, there is a derivation $A \Rightarrow^+ A\alpha$

- **To eliminate direct left recursion replace**

1) $A \rightarrow A\alpha \mid \beta$ **with** $A' \rightarrow \alpha A' \mid \varepsilon$

2) $A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \ldots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$

with

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

$B \rightarrow \alpha_1 B \mid \alpha_2 B \mid \ldots \mid \alpha_m B \mid \varepsilon$

4. 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 can rewrite the A-productions to defer the decision until we have seen enough of the input to make the right choice.

- Consider S → if E then S else S | if E then S
    - Which of the two productions should we use to expand non-terminal S when the next token is if?

      We can solve this problem by factoring out the common part in these rules.

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

becomes

$A \rightarrow \alpha B \mid \gamma$

$B \rightarrow \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$

Consider the grammar , G : S → iEtS | iEtSeS | a

E → b

Left factored, this grammar becomes

S → iEtSS' | a

S' → eS |ε

E → b

PARSING

It is the process of analyzing a continuous stream of input in order to determine its grammatical structure with respect to a given formal grammar.

Types of parsing:

1. Top down parsing
2. Bottom up parsing

Top-down parsing : A parser can start with the start symbol and try to transform it to the input string. **Example :** LL Parsers.

Bottom-up parsing : A parser can start with input and attempt to rewrite it into the start symbol.
Example : LR Parsers.

TOP-DOWN PARSING

It can be viewed as an attempt to find a left-most derivation for an input string or an attempt to construct a parse tree for the input starting from the root to the leaves.

Types of TOP-DOWN PARSING

1. Recursive descent parsing
2. Predictive parsing

RECURSIVE DESCENT PARSING

Recursive descent is a top-down parsing technique that constructs the parse tree from the top and the input is read from left to right. It uses procedures for every non-terminal entity. This parsing technique recursively parses the input to make a parse tree, which may or may not require back-tracking. But the grammar associated with it (if not left factored) cannot avoid back-tracking. A form of recursive-descent parsing that does not require any back-tracking is known as **predictive parsing**.

This parsing technique is regarded recursive as it uses context-free grammar which is recursive in nature.

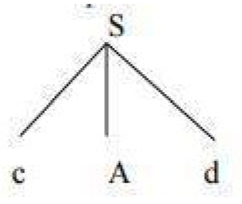
This parsing method may involve backtracking.
Example for :backtracking

Consider the grammar G : S → cAd
A→ab|a

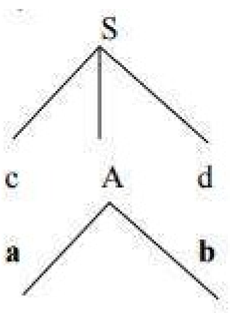
and the input string w=cad.

The parse tree can be constructed using the following top-down approach :

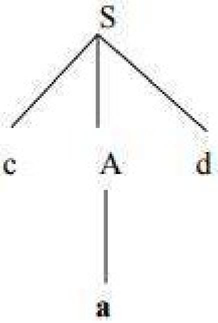
Step1:
Initially create a tree with single node labeled S. An input pointer points to 'c', the first symbol of w. Expand the tree with the production of S.



Step2:
The leftmost leaf 'c' matches the first symbol of w, so advance the input pointer to the second symbol of w 'a' and consider the next leaf 'A'. Expand A using the first alternative.



Step3:
The second symbol ‘a’ of w also matches with second leaf of tree. So advance the input pointer to third symbol of w ‘d’**.** But the third leaf of tree is b which does not match with the input symbol **d.** Hence discard the chosen production and reset the pointer to second **backtracking.**
Step4: Now try the second alternative for A.



Now we can halt and announce the successful completion of parsing.

Predictive parsing

It is possible to build a nonrecursive predictive parser by maintaining a stack explicitly, rather than implicitly via recursive calls. The key problem during predictive parsing is that of determining the production to be applied for a nonterminal . The nonrecursive parser in figure looks up the production to be applied in parsing table. In what follows, we shall see how the table can be constructed directly from certain grammars.

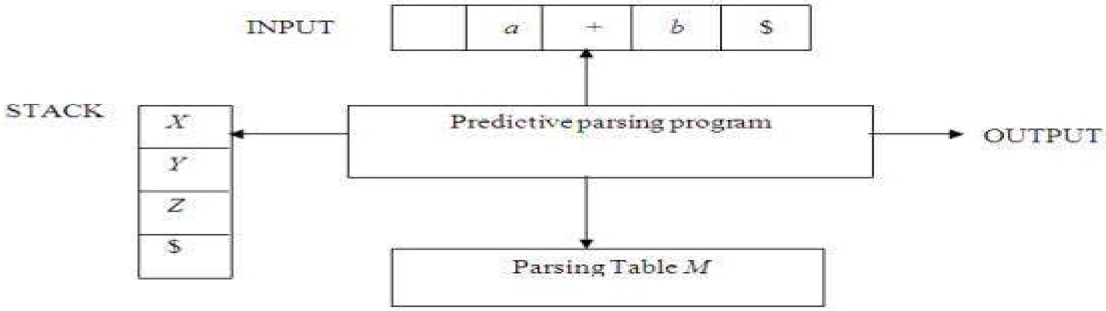


Fig. 2.4 Model of a nonrecursive predictive parser

A table-driven predictive parser has an input buffer, a stack, a parsing table, and an output stream. The input buffer contains the string to be parsed, followed by $, a symbol used as a right endmarker to indicate the end of the input string. The stack contains a sequence of grammar symbols with $ on the bottom, indicating the bottom of the stack. Initially, the stack contains the start symbol of the grammar on top of $. 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 considers X, the symbol on the 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, for example, M[X,a]={X- >UVW}, the parser replaces X on top of the stack by WVU(with U on top). As output, we shall assume that the parser just prints the production used; any other code could be executed here. If M[X,a]=error, the parser calls an error recovery routine

Implementation of predictive parser:

1. Elimination of left recursion, left factoring and ambiguous grammar.
2. Construct FIRST() and FOLLOW() for all non-terminals.
3. Construct predictive parsing table.
4. Parse the given input string using stack and parsing table

Algorithm for Nonrecursive predictive parsing.

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 is in a configuration in which it 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 shown in Fig.

```
 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 of $ then
      if X=a then
         pop X from the stack and advance ip else error()
      else
      if M[X,a]=X->Y1Y2...Yk then begin pop X from the stack;
         push Yk,Yk-1...Y1 onto the stack, with Y1 on top; output the production X-> Y1Y2...Yk
      end
      else error()
   until X=$ /* stack is empty */
```

FIRST AND FOLLOW

The construction of a predictive parsing table is aided by two functions associated with a grammar:

1. FIRST
2. FOLLOW

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

Rules for FIRST **( ):**

1. If X is terminal, then FIRST(X) is {X}.
2. If X → ε is a production, then add ε to FIRST(X).
3. If X is non-terminal and X → aα is a production then add a to FIRST(X).
4. If X is non-terminal and X → Y1 Y2…Yk is a production, then place a in FIRST(X) if for some i, a is in FIRST(Yi), and ε is in all of FIRST(Y1),…,FIRST(Yi-1);that is, Y1,….Yi-1=> ε. If ε is in FIRST(Yj) for all j=1,2,..,k, then add ε to FIRST(X).

Rules for FOLLOW ( ):

1. If S is a start symbol, then FOLLOW(S) contains $.
2. If there is a production A → αBβ, then everything in FIRST(β) except ε is placed in follow(B).
3. If there is a production A → αB, or a production A → αBβ where FIRST(β) contains ε, then everything in FOLLOW(A) is in FOLLOW(B).

Algorithm for construction of predictive parsing table:

Input : Grammar G
Output : Parsing table M
Method :

1. For each production A → α of the grammar, do steps 2 and 3.
2. For each terminal a in FIRST(α), add A → α to M[A, a].
3. If ε is in FIRST(α), add A → α to M[A, b] for each terminal b in FOLLOW(A). If ε is in FIRST(α) and $ is in FOLLOW(A) , add A → α to M[A, $].
4. Make each undefined entry of M be error.

Example:

Consider the following grammar :

E→E+T|T
T→T*F|F
F→(E)|id

After eliminating left-recursion the grammar is

E →TE’
E’ → +TE’ | ε
T →FT’
T’ → *FT’ | ε
F → (E)|id

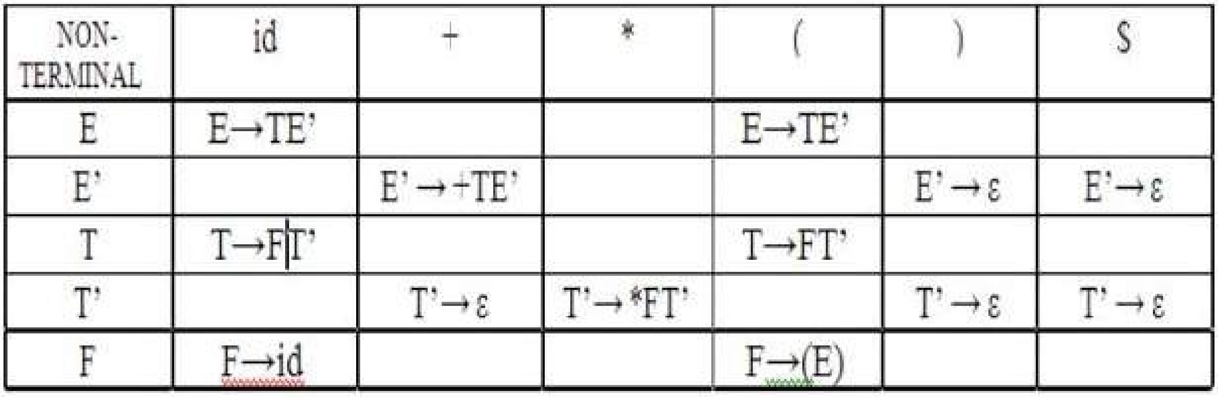
First( ) :

FIRST(E) = { (, id}
FIRST(E’) ={+ , ε }
FIRST(T) = { (, id}
FIRST(T’) = {*, ε }
FIRST(F) = { (, id }

Follow( ):

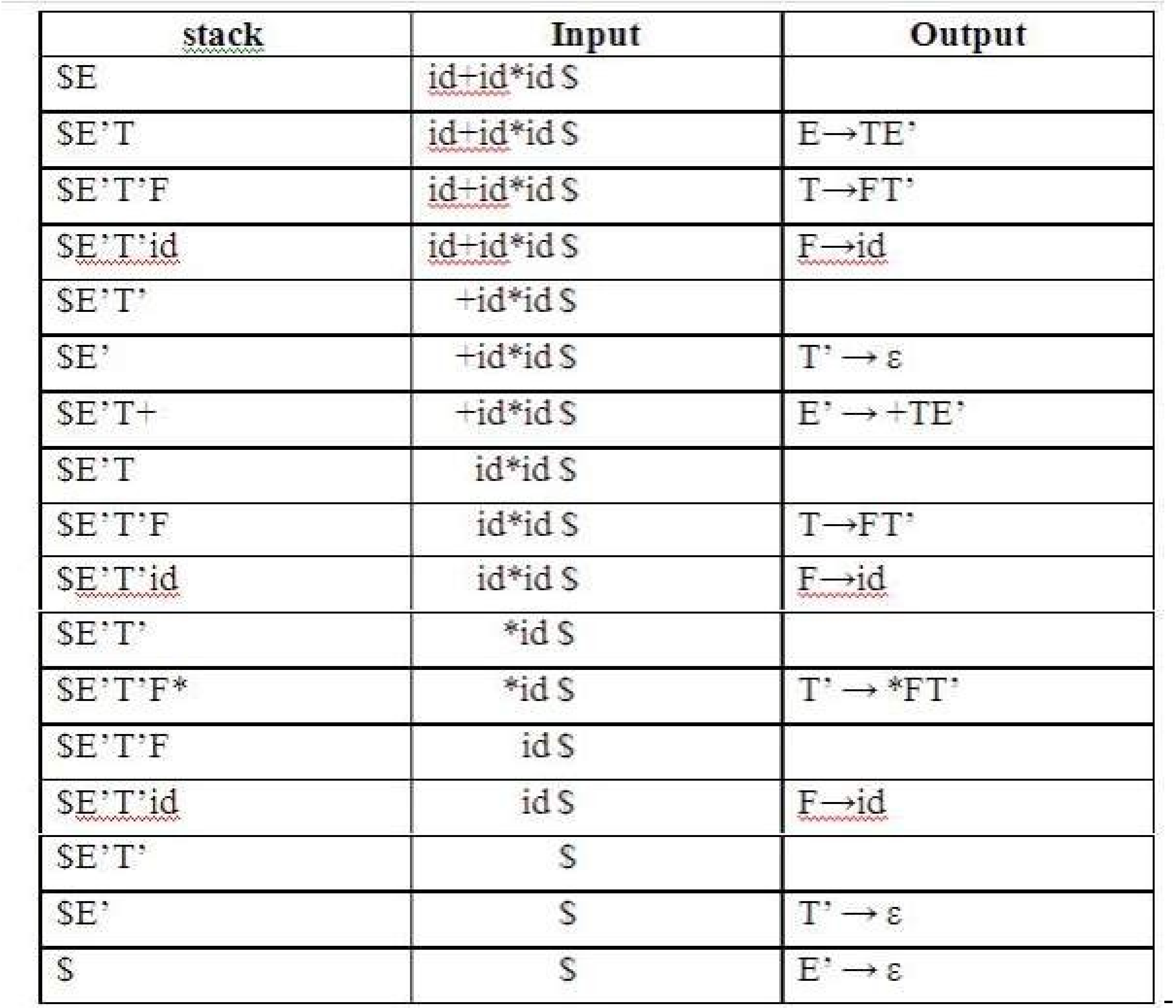
FOLLOW(E) = { $,) }
FOLLOW(E’) = { $,) }
FOLLOW(T) = { +, $,) }
FOLLOW(T’) = { +, $,) }
FOLLOW(F) = {+, * , $,) }

Predictive parsing Table



NON-TERMINAL	id	+	*	(	)	$
E	E→TE’			E→TE’		
E’		E’→+TE’			E’→ε	E’→ε
T	T→FT’			T→FT’		
T’		T’→ε	T’→*FT’		T’→ε	T’→ε
F	F→id			F→(E)		

Stack Implementation



stack	Input	Output
$E	id+id*id $	
$E’T	id+id*id $	E→TE’
$E’T’F	id+id*id $	T→FT’
$E’T’id	id+id*id $	F→id
$E’T’	+id*id $	
$E’	+id*id $	T’ → ε
$E’T+	+id*id $	E’ → +TE’
$E’T	id*id $	
$E’T’F	id*id $	T→FT’
$E’T’id	id*id $	F→id
$E’T’	*id $	
$E’T’F*	*id $	T’ → *FT’
$E’T’F	id $	
$E’T’id	id $	F→id
$E’T’	$	
$E’	$	T’ → ε
$	$	E’ → ε

LL(1) GRAMMAR

The parsing table algorithm can be applied to any grammar G to produce a parsing table M. For some Grammars, for example if G is left recursive or ambiguous, then M will have at least one multiply-defined entry. A grammar whose parsing table has no multiply defined entries is said to be LL(1). It can be shown that the above algorithm can be used to produce for every LL(1) grammar G, a parsing table M that parses all and only the sentences of G. LL(1) grammars have several distinctive properties. No ambiguous or left recursive grammar can be LL(1). There remains a question of what should be done in case of multiply defined entries. One easy solution is to eliminate all left recursion and left factoring, hoping to produce a grammar which will produce no multiply defined entries in the parse tables. Unfortunately there are some grammars which will give an LL(1) grammar after any kind of alteration. In general, there are no universal rules to convert multiply defined entries into single valued entries without affecting the language recognized by the parser.

The main difficulty in using predictive parsing is in writing a grammar for the source language such that a predictive parser can be constructed from the grammar. Although left recursion elimination and left factoring are easy to do, they make the resulting grammar hard to read and difficult to use the translation purposes. To alleviate some of this difficulty, a common organization for a parser in a compiler is to use a predictive parser for control constructs and to use operator precedence for expressions.however, if an lr parser generator is available, one can get all the benefits of predictive parsing and operator precedence automatically.

ERROR RECOVERY IN PREDICTIVE PARSING

The stack of a nonrecursive predictive parser makes explicit the terminals and nonterminals that the parser hopes to match with the remainder of the input. We shall therefore refer to symbols on the parser stack in the following discussion. An error is detected during predictive parsing when the terminal on top of the stack does not match the next input symbol or when nonterminal A is on top of the stack, a is the next input symbol, and the parsing table entry M[A,a] is empty.

Consider error recovery predictive parsing using the following two methods

Panic-Mode recovery
Phrase Level recovery

Panic-mode error recovery is based on the idea of skipping symbols on the input until a token in a selected set of synchronizing tokens appears. Its effectiveness depends on the choice of synchronizing set. The sets should be chosen so that the parser recovers quickly from errors that are likely to occur in practice.

As a starting point, we can place all symbols in FOLLOW(A) into the synchronizing set for nonterminal A. If we skip tokens until an element of FOLLOW(A) is seen and pop A from the stack, it is likely that parsing can continue.

It is not enough to use FOLLOW(A) as the synchronizingset for A. Fo example , if semicolons terminate statements, as in C, then keywords that begin statements may not appear in the FOLLOW set of the nonterminal generating expressions. A missing semicolon after an assignment may therefore result in the keyword beginning the next statement being skipped. Often, there is a hierarchica structure on constructs in a language; e.g., expressions appear within statement, which appear within bblocks,and so on. We can add to the synchronizing set of a lower construct the symbols that begin higher constructs. For example, we might add keywords that begin statements to the synchronizing sets for the nonterminals generaitn expressions.

If we add symbols in FIRST(A) to the synchronizing set for nonterminal A, then it may be possible to resume parsing according to A if a symbol in FIRST(A) appears in the input.

If a nonterminal can generate the empty string, then the production deriving e can be used as a default. Doing so may postpone some error detection, but cannot cause an error to be missed. This approach reduces the number of nonterminals that have to be considered during error recovery.

If a terminal on top of the stack cannot be matched, a simple idea is to pop the terminal, issue a message saying that the terminal was inserted, and continue parsing. In effect, this approach takes the synchronizing set of a token to consist of all other tokens.

Phrase Level recovery

This involves, defining the blank entries in the table with pointers to some error routines which may

Change, delete or insert symbols in the input or
May also pop symbols from the stack

BOTTOM-UP PARSING

Constructing a parse tree for an input string beginning at the leaves and going towards the root is called bottom-up parsing. A general type of bottom-up parser is a shift-reduce parser.

2.6.1 SHIFT-REDUCE PARSING

Shift-reduce parsing is a type of bottom-up parsing that attempts to construct a parse tree for an input string beginning at the leaves (the bottom) and working up towards the root (the top).

Example:

Consider the grammar: S → aABe
A → Abc | b
B → d

The sentence to be recognized is abbcde.

REDUCTION (LEFTMOST)	**RIGHTMOST DERIVATION**
abbcde (A → b)	S → aABe
aAbcde(A → Abc)	→ aAde
aAde (B → d)	→ aAbcde
aABe (S → aABe)	→ abbcde
S	

The reductions trace out the right-most derivation in reverse.

Handles: A handle of a string is a substring that matches the right side of a production, and whose reduction to the non-terminal on the left side of the production represents one step along the reverse of a rightmost derivation.

Example:

Consider the grammar:

E→E+E
E→E*E
E→(E)
E→id

And the input string id1+id2*id3

The rightmost derivation is :

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

In the above derivation the underlined substrings are called handles.

Handle pruning:

A rightmost derivation in reverse can be obtained by "handle pruning". (i.e.) if w is a sentence or string of the grammar at hand, then w = γn, where γn is the nth right sentential form of some rightmost derivation.

Actions in shift-reduce parser:

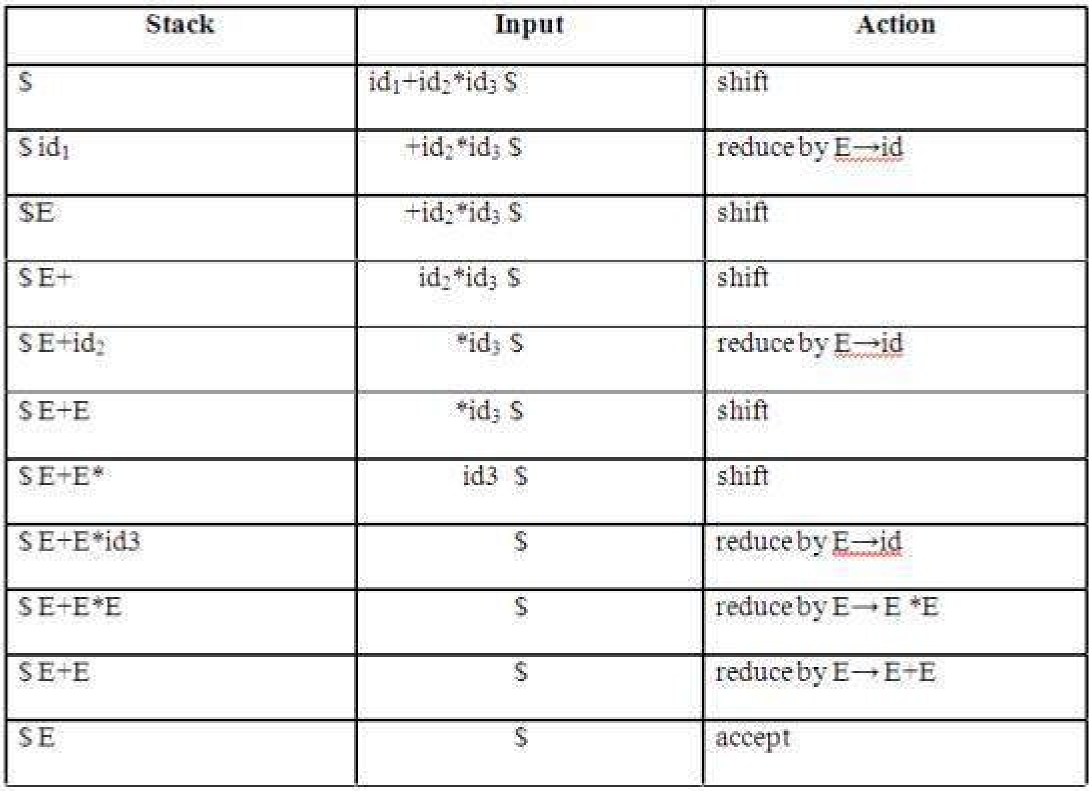
- shift - The next input symbol is shifted onto the top of the stack.
- reduce - The parser replaces the handle within a stack with a non-terminal.
- accept - The parser announces successful completion of parsing.
- error - The parser discovers that a syntax error has occurred and calls an error recovery routine.

Conflicts in shift-reduce parsing:

There are two conflicts that occur in shift-reduce parsing:

1. Shift-reduce conflict: The parser cannot decide whether to shift or to reduce.
2. Reduce-reduce conflict: The parser cannot decide which of several reductions to make.

Stack implementation of shift-reduce parsing :



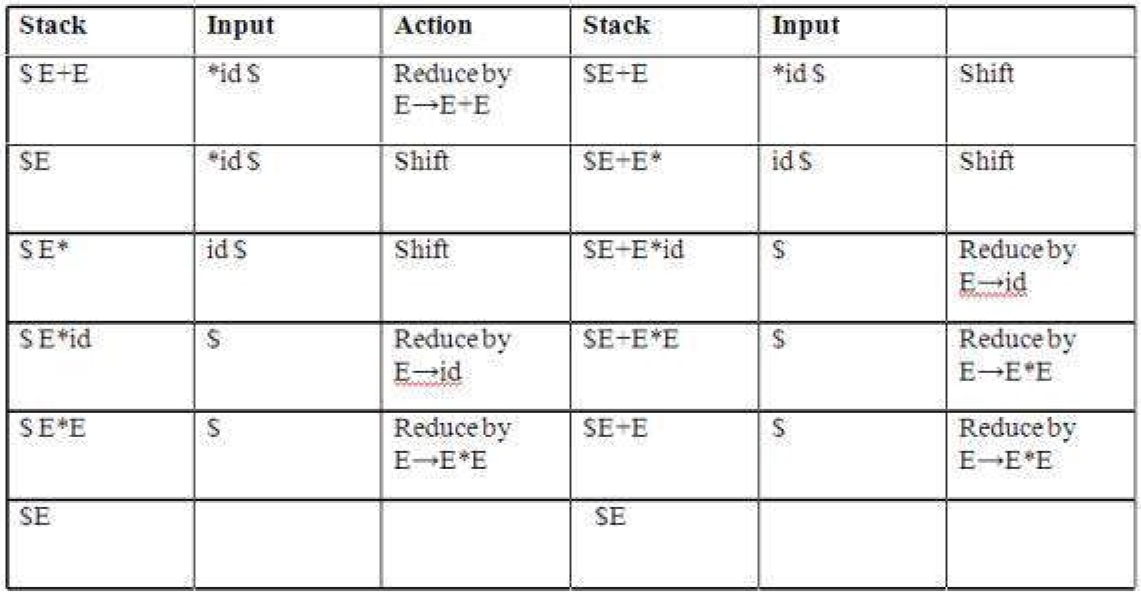
Stack	Input	Action
$	id_1+id_2*id_3 $	shift
$ id_1	+id_2*id_3 $	reduce by E→id
$E	+id_2*id_3 $	shift
$ E+	id_2*id_3 $	shift
$ E+id_2	*id_3 $	reduce by E→id
$ E+E	*id_3 $	shift
$ E+E*	id3 $	shift
$ E+E*id3	$	reduce by E→id
$ E+E*E	$	reduce by E→ E *E
$ E+E	$	reduce by E→ E+E
$ E	$	accept

1. Shift-reduce conflict:

Example:

Consider the grammar:

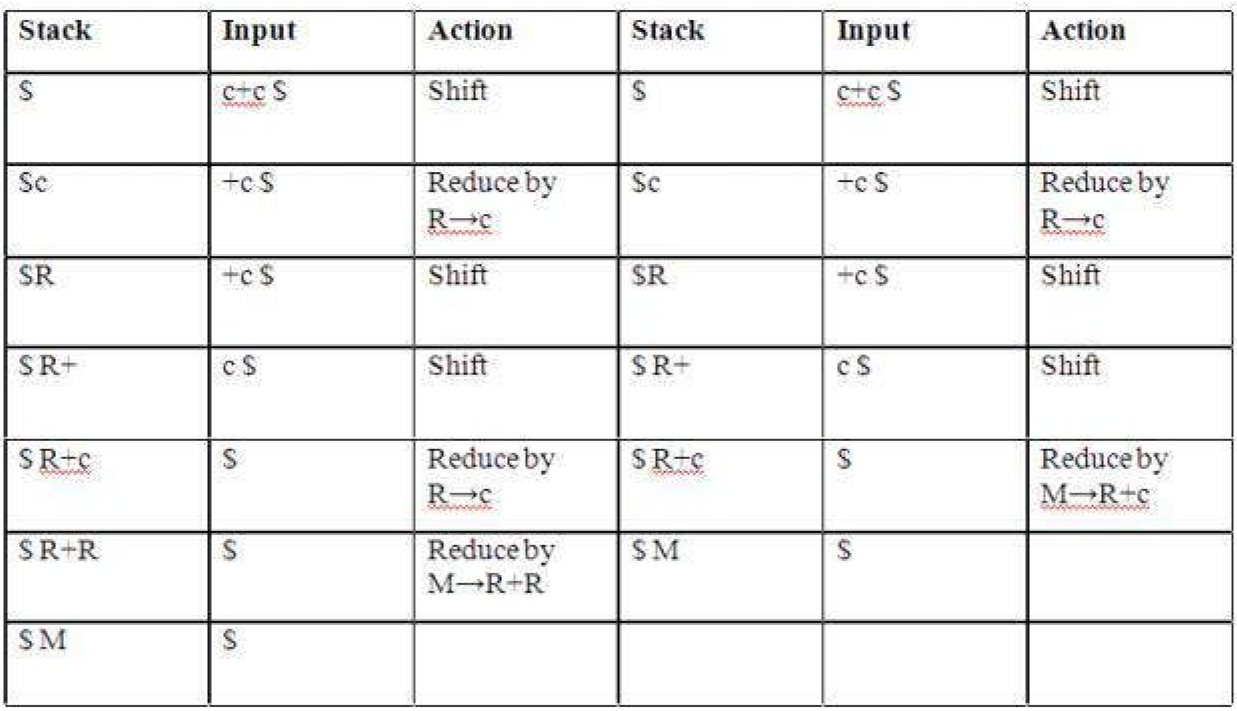
E→E+E | E*E | id and input id+id*id



Stack	Input	Action	Stack	Input	
$ E+E	*id $	Reduce by E→E+E	$E+E	*id $	Shift
$E	*id $	Shift	$E+E*	id $	Shift
$ E*	id $	Shift	$E+E*id	$	Reduce by E→id
$ E*id	$	Reduce by E→id	$E+E*E	$	Reduce by E→E*E
$ E*E	$	Reduce by E→E*E	$E+E	$	Reduce by E→E*E
$E			$E		

2. Reduce-reduce conflict:

Consider the grammar: M→R+R|R+c|R
R→c
input c+c



Stack	Input	Action	Stack	Input	Action
$	c+c $	Shift	$	c+c $	Shift
$c	+c $	Reduce by R→c	$c	+c $	Reduce by R→c
$R	+c $	Shift	$R	+c $	Shift
$ R+	c $	Shift	$ R+	c $	Shift
$ R+c	$	Reduce by R→c	$ R+c	$	Reduce by M→R+c
$ R+R	$	Reduce by M→R+R	$ M	$	
$ M	$				

INTRODUCTION TO LR PARSERS

An efficient bottom-up syntax analysis technique that can be used CFG is called LR(k) parsing. The ‘L’ is for left-to-right scanning of the input, the ‘R’ for constructing a rightmost derivation in reverse, and the ‘k’ for the number of input symbols. When ‘k’ is omitted, it is assumed to be 1.

Advantages of LR parsing:

1. It recognizes virtually all programming language constructs for which CFG can be written.
2. It is an efficient non-backtracking shift-reduce parsing method.
3. A grammar that can be parsed using LR method is a proper superset of a grammar that can be parsed with predictive parser
4. It detects a syntactic error as soon as possible.

Drawbacks of LR method:

It is too much of work to construct a LR parser by hand for a programming language grammar. A specialized tool, called a LR parser generator, is needed. Example: YACC.

Types of LR parsing method:

1. SLR- Simple LR

Easiest to implement, least powerful.

2. CLR- Canonical LR

Most powerful, most expensive.

3. LALR- Look-Ahead LR

Intermediate in size and cost between the other two methods.

The LR parsing algorithm:

The schematic form of an LR parser is as follows:

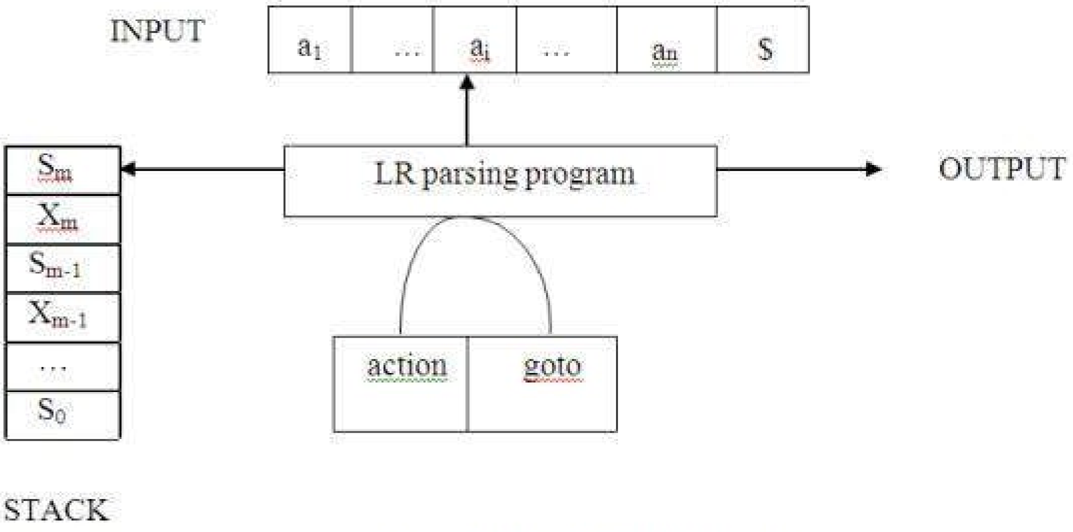


Fig. 2.5 Model of an LR parser

It consists of an input, an output, a stack, a driver program, and a pa parts (action and goto).

a. The driver program is the same for all LR parser.
b. The parsing program reads characters from an input buffer one at a time.
c. The program uses a stack to store a string of the form s0X1s1X2s2…Xmsm, where sm is on top. Each Xi is a grammar symbol and each si is a state.
d. The parsing table consists of two parts : action and goto functions.

Action : The parsing program determines sm, the state currently on top of stack, and ai, the current input symbol. It then consults action[sm,ai] in the action table which can have one of four values:

1. shift s, where s is a state,
2. reduce by a grammar production A → β,
3. accept,
4. Error.

Goto : The function goto takes a state and grammar symbol as arguments and produces a state.

LR Parsing algorithm:

Input: An input string w and an LR parsing table with functions action and goto for grammar G. Output: If w is in L(G), a bottom-up-parse for w; otherwise, an error indication.

Method: Initially, the parser has s0 on its stack, where s0 is the initial state, and w$ in the input buffer.

The parser then executes the following program:

```
set ip to point to the first input symbol of w$; repeat forever begin
    let s be the state on top of the stack and a the symbol pointed to by ip;
if action[s, a] = shift s' then begin
    push a then s' on top of the stack; advance ip to the next input symbol end
    else if action[s, a] = reduce $A \to \beta$ then begin pop 2* | $\beta$ | symbols off the stack;
    let s' be the state now on top of the stack; push A then goto[s', A] on top of the stack; output the
    production $A \to \beta$
    end
    else if action[s, a] = accept then
    return
    else error( )
    end
```

CONSTRUCTING SLR(1) PARSING TABLE

To perform SLR parsing, take grammar as input and do the following:

1. Find LR(0) items.
2. Completing the closure.
3. Compute goto(I,X), where, I is set of items and X is grammar symbol.

LR(0) items:

An LR(0) item of a grammar G is a production of G with a dot at some position of the right side. For example, production $A \to XYZ$ yields the four items :

$A \to .XYZ$
$A \to X . YZ$
$A \to XY . Z$
$A \to XYZ .$

Closure operation:

If I is a set of items for a grammar G, then closure(I) is the set of items constructed from I by the two rules:

1. Initially, every item in I is added to closure(I).
2. If $A \to \alpha . B\beta$ is in closure(I) and $B \to \gamma$ is a production, then add the item $B \to . \gamma$ to I , if it is not already there. We apply this rule until no more new items can be added to closure(I).

Goto operation:

Goto(I, X) is defined to be the closure of the set of all items $[A \to \alpha X . \beta]$ such that $[A \to \alpha . X\beta]$ is in I.

Steps to construct SLR parsing table for grammar G are:

1. Augment G and produce G'
2. Construct the canonical collection of set of items C for G'
3. Construct the parsing action function action and goto using the following algorithm that requires FOLLOW(A) for each non-terminal of grammar.

Algorithm for construction of SLR parsing table:

Input : An augmented grammar G'

Output : The SLR parsing table functions action and goto for G'

Method :

1. Construct C = {I0, I1, …. In}, the collection of sets of LR(0) items for G'.
2. State i is constructed from Ii.. The parsing functions for state i are determined as follows:
   (a) If [A→α·aβ] is in Ii and goto(Ii,a) = Ij, then set action[i,a] to "shift j". Here a must be terminal.
   (b) If [A→α·] is in Ii , then set action[i,a] to "reduce A→α" for all a in FOLLOW(A).
   (c) If [S'→S.] is in Ii, then set action[i,$] to "accept".
   If any conflicting actions are generated by the above rules, we say grammar is not SLR(1).
3. The goto transitions for state i are constructed for all non-term
   If goto(Ii,A) = Ij, then goto[i,A] = j.
4. All entries not defined by rules (2) and (3) are made "error"
5. The initial state of the parser is the one constructed from the [S'→.S].

Example on SLR (1) Grammar

S → E
E → E + T | T
T → T * F | F
F → id

Add Augment Production and insert '•' symbol at the first position for every production in G

S` → •E
E → •E + T
E → •T
T → •T * F
T → •F
F → •id

I0 State:

Add Augment production to the I0 State and Compute the Closure

I0 = Closure (S` → •E)

Add all productions starting with E in to I0 State because "." is followed by the non-terminal. So, the I0 State becomes

I0 = S` → •E
E → •E + T
E → •T

Add all productions starting with T and F in modified I0 State because "." is followed by the non-terminal. So, the I0 State becomes.

I0= S` → •E
E → •E + T
E → •T
T → •T * F
T → •F
F → •id

I1= Go to (I0, E) = closure (S` → E•, E → E• + T)
I2= Go to (I0, T) = closure (E → T•T, T• → * F)
I3= Go to (I0, F) = Closure (T → F•) = T → F•
I4= Go to (I0, id) = closure (F → id•) = F → id•
I5= Go to (I1, +) = Closure (E → E +•T)

Add all productions starting with T and F in I5 State because "." is followed by the non-terminal. So, the I5 State becomes

I5 = E → E +•T
T → •T * F
T → •F
F → •id

Go to (I5, F) = Closure (T → F•) = (same as I3)
Go to (I5, id) = Closure (F → id•) = (same as I4)

I6= Go to (I2, *) = Closure (T → T * •F)

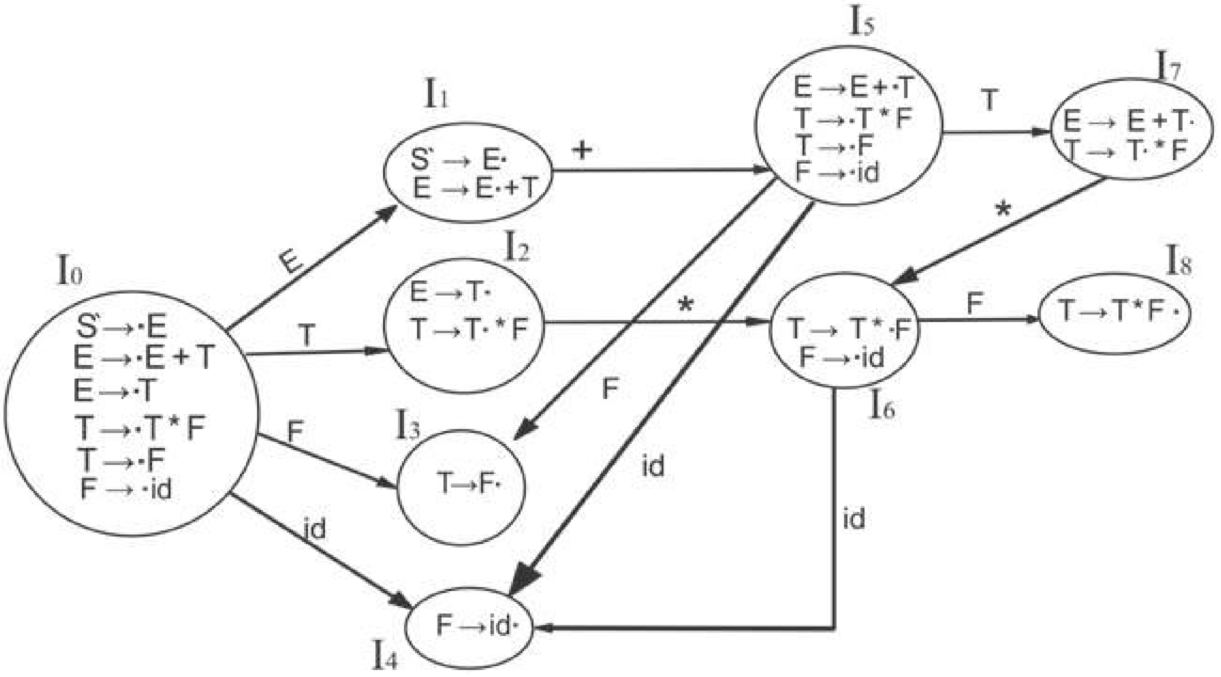
Add all productions starting with F in I6 State because "." is followed by the non-terminal. So, the I6 State becomes

I6 = T → T * •F
F → •id

Go to (I6, id) = Closure (F → id•) = (same as I4)

I7= Go to (I5, T) = Closure (E → E + T•) = E → E + T•
I8= Go to (I6, F) = Closure (T → T * F•) = T → T * F•

Drawing DFA



SLR (1) Table

States	Action				Go to		
	id	+	*	$	**E**	**T**	**F**
I_0	S_4				1	2	3
I_1		S_5		Accept			
I_2		R_2	S_6	R2			
I_3		R_4	R_4	R4			
I_4		R_5	R5	R5			
I_5	S4					7	3
I_6	S4						8
I_7		R1	S6	R1			
I_8		R3	R3	R3			

Explanation:

First (E) = First (E + T) ∪ First (T)
First (T) = First (T * F) ∪ First (F)
First (F) = {id}
First (T) = {id}
First (E) = {id}
Follow (E) = First (+T) ∪ {$} = {+, $}
Follow (T) = First (*F) ∪ First (F)
= {*, +, $}
Follow (F) = {*, +, $}

1) I1 contains the final item which drives S → E• and follow (S) = {$}, so action {I1, $} = Accept

2) I2 contains the final item which drives E → T• and follow (E) = {+, $}, so action {I2, +} = R2, action {I2, $} = R2

3) I3 contains the final item which drives T → F• and follow (T) = {+, *, $}, so action {I3, +} = R4, action {I3, *} = R4, action {I3, $} = R4

4) I4 contains the final item which drives F → id• and follow (F) = {+, *, $}, so action {I4, +} = R5, action {I4, *} = R5, action {I4, $} = R5

5) I7 contains the final item which drives E → E + T• and follow (E) = {+, $}, so action {I7, +} = R1, action {I7, $} = R1

6) I8 contains the final item which drives T → T * F• and follow (T) = {+, *, $}, so action {I8, +} = R3, action {I8, *} = R3, action {I8, $} = R3.

CANONICAL LR PARSING

CLR refers to canonical look ahead. CLR parsing use the canonical collection of LR (1) items to build the CLR (1) parsing table. CLR (1) parsing table produces the more number of states as compare to the SLR (1) parsing.

In the CLR (1), we place the reduce node only in the look ahead

symbols.Various steps involved in the CLR (1) Parsing:

1) For the given input string write a context free grammar
2) Check the ambiguity of the grammar
3) Add Augment production in the given grammar
4) Create Canonical collection of LR (0) items
5) Draw a data flow diagram (DFA)
6) Construct a CLR (1) parsing table

In the SLR method we were working with LR(0)) items. In CLR parsing we will be using LR(1) items. LR(k) item is defined to be an item using lookaheads of length k. So ,the LR(1) item is comprised of two parts : the LR(0) item and the lookahead associated with the item. The look ahead is used to determine that where we place the final item. The look ahead always add $ symbol for the argument production.

LR(1) parsers are more powerful parser.

for LR(1) items we modify the Closure and GOTO function.

Closure Operation

```
Closure(I)
repeat
   for (each item [ A -> ?.B?, a ] in I )
      for (each production B -> ? in G')
      for (each terminal b in FIRST(?a))
         add [ B -> .? , b ] to set I;
until no more items are added to I;
return I;
```

Goto Operation

```
Goto(I, X)
Initialise J to be the empty set;
for ( each item A -> ?.X?, a ] in I )
    Add item A -> ?X.?, a ] to se J;  /* move the dot one step */
return Closure(J);    /* apply closure to the set */
```

LR(1) items

```
Void items(G’)
Initialise C to { closure ({[S’ -> .S, $]})};
Repeat
    For (each set of items I in C)
        For (each grammar symbol X)
            if( GOTO(I, X) is not empty and not in C)
                Add GOTO(I, X) to C;
Until no new set of items are added to C;
```

ALGORITHM FOR CONSTRUCTION OF THE CANONICAL LR PARSING TABLE

Input: grammar G'
Output: canonical LR parsing table functions action and go to

1. Construct C = {I0, I1 , ..., In} the collection of sets of LR(1) items for G'.State i is constructed from Ii.
2. if [*A* -> a.ab, b>] is in Ii and go to(Ii, a) = Ij, then set action[i, a] to "shift j". Herea must be a terminal.
3. if [*A* -> a., a] is in Ii, then set action[i, a] to "reduce *A* -> a" for all a in FOLLOW(*A*). Here *A* may *not* be *S'*.
4. if [*S'* -> *S*.] is in Ii, then set action[i, $] to "accept"
5. If any conflicting actions are generated by these rules, the grammar is not LR(1) and the algorithm fails to produce a parser.
6. The go to transitions for state i are constructed for all *non terminal*s *A* using therule: If go to(Ii, A)= Ij, then go to[i, *A*] = j.
7. All entries not defined by rules 2 and 3 are made "error".
8. The inital state of the parser is the one constructed from the set of items containing [*S'* -> .S, $].

Example,

Consider the following grammar,

S‟->S
S->CC
C->cC
C->d

Sets of LR(1) items

I0: S‟->.S,$
S->.CC,$
C->.Cc,c/d
C->.d,c/d

I1: S‟->S.,$
I2: S->C.C,$
C->.Cc,$
C->.d,$

I3: C->c.C,c/d C-
>.Cc,c/dC-
>.d,c/d

I4: C->d.,c/d

I5: S->CC.,$

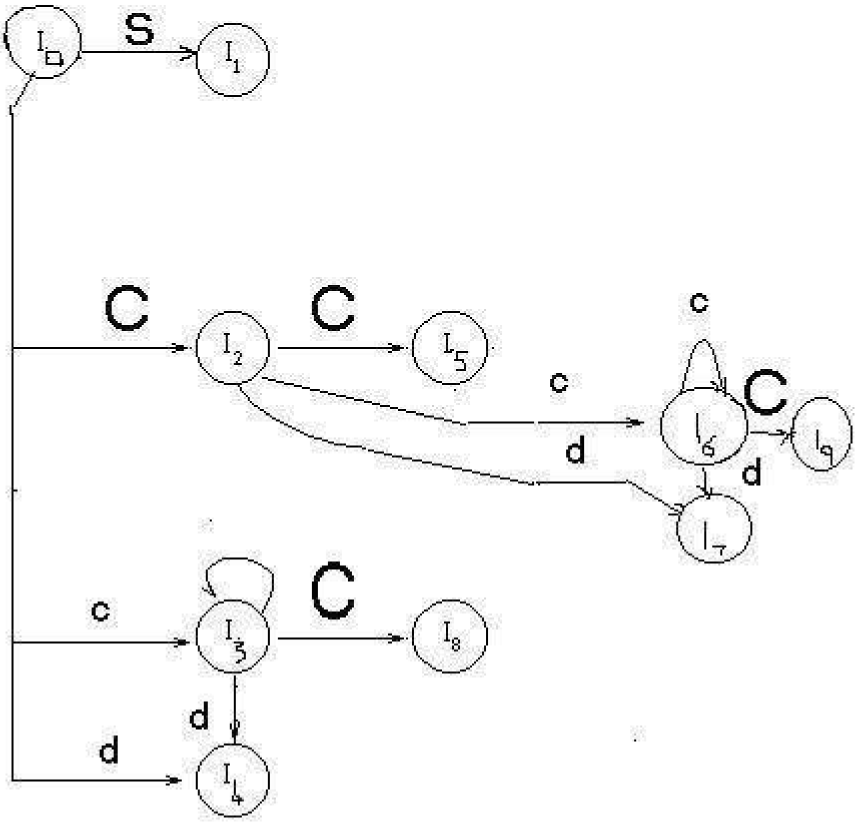
I6: C->c.C,$
C->.cC,$
C->.d,$

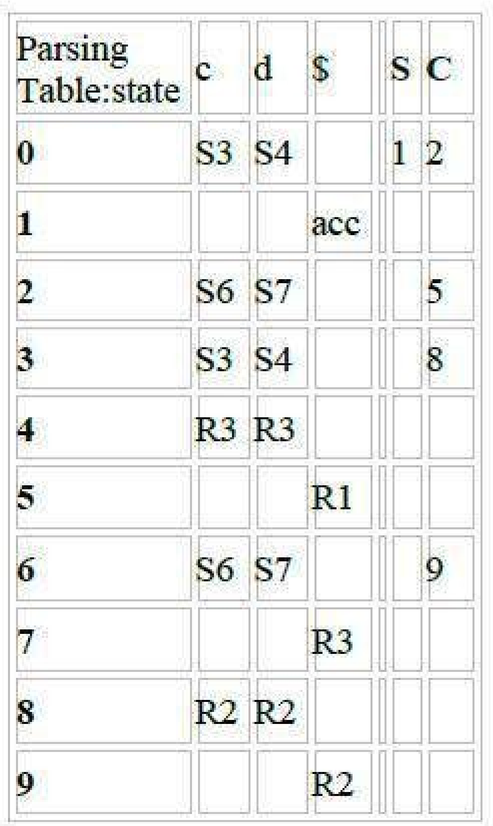
I7: C->d.,$

I8: C->cC.,c/d

I9: C->cC.,$

Here is what the corresponding DFA looks like





Parsing Table:state	c	d	$		S	C
0	S3	S4			1	2
1			acc			
2	S6	S7				5
3	S3	S4				8
4	R3	R3				
5			R1			
6	S6	S7				9
7			R3			
8	R2	R2				
9			R2			