BAPATLA ENGINEERING COLLEGE :: BAPATLA (AUTONOMOUS)

Compiler Design (18IT502)

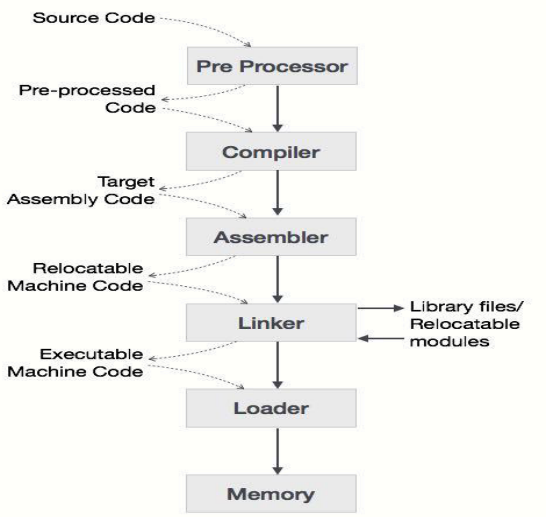
Lecture Notes

Department of Information Technology
BAPATLA ENGINEERING COLLEGE :: BAPATLA (AUTONOMOUS)
Affiliated to Acharya Nagarjuna University
Bapatla-522102, Guntur(District), AP.

UNIT-III

1. Introduction to Compiling

We have learnt that any computer system is made of hardware and software. The hardware understands a language, which humans cannot understand. So we write programs in high-level language, which is easier for us to understand and remember. These programs are then fed into a series of tools and OS components to get the desired code that can be used by the machine. This is known as 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.

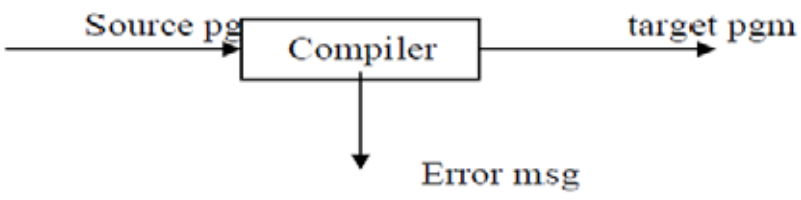
2. *File inclusion:* A preprocessor may include header files into the program text.

3. *Rational preprocessor:* these preprocessors augment older languages with more modern flow-ofcontrol and data structuring facilities.

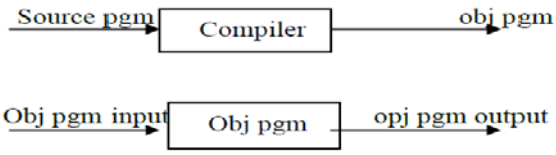
4. *Language Extensions:* These preprocessor attempts to add capabilities to the language by certain amounts to build-in macro.

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 translated into an object program. Then the results object program is loaded into a memory executed.

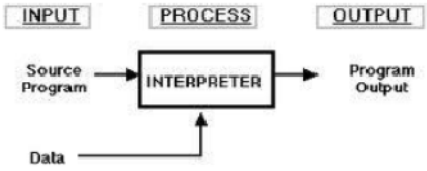


Assembler:

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. The input to an assembler program is called source program, the output is a machine language translation (object program).

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.
- 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 the program is *slower*.
- *Memory* consumption is more.

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, hereby 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 over come 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.

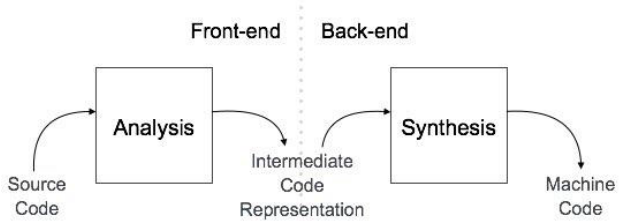
Translator:
A translator is a program that takes as input a program written in one language and produces as output a program in another language. Beside program translation, the translator performs another very important role, the error-detection. Any violation of d HLL specification would be detected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

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 .Common Lisp compilers
9. ECMAScript interpreters
10. Fortran compilers
11 .Java compilers
12. Pascal compilers
13. PL/I compilers
14. Python compilers
15. Smalltalk compilers

THE PHASES OF A COMPILER

A compiler can broadly be divided into two phases based on the way they compile.

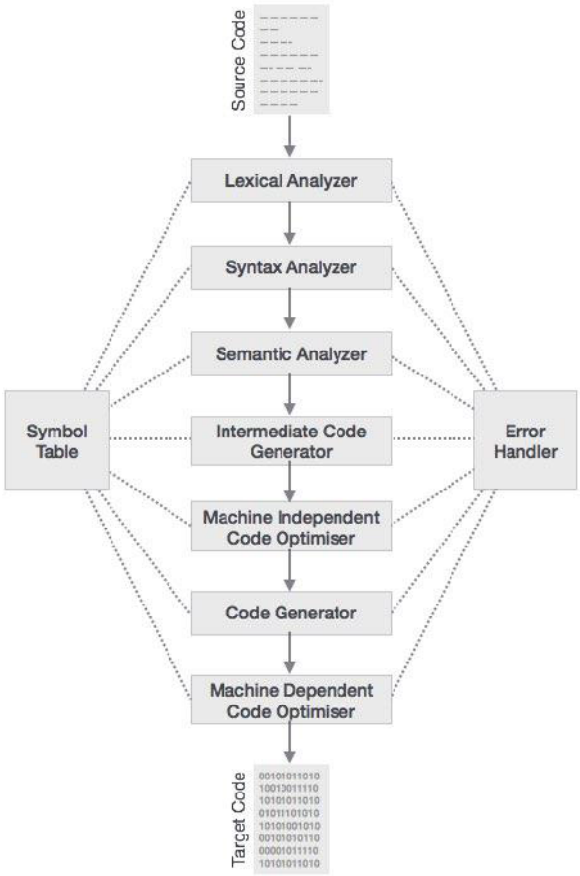
Analysis Phase:
Known as the front-end of the compiler, the **analysis** phase of the compiler reads the source program, divides it into core parts, and then checks for lexical, grammar, and syntax errors. The analysis phase generates an intermediate representation of the source program and symbol table, which should be fed to the Synthesis phase as input.



Synthesis Phase:
Known as the back-end of the compiler, the **synthesis** phase generates the target program with the help of intermediate source code representation and symbol table.
A compiler can have many phases and passes.
 Pass : A pass refers to the traversal of a compiler through the entire program.

 Phase : A phase of a compiler is a distinguishable stage, which takes input from the previous stage, processes and yields output that can be used as input for the next stage. A pass can have more than one phase.

The compilation process is a sequence of various phases. Each phase takes input from its previous stage, has its own representation of source program, and feeds its output to the next phase of the compiler. Let us understand the phases of a compiler.



Lexical Analysis:

LA or Scanner reads the source program one character at a time, separates the source program into a sequence of atomic units called **tokens.** The usual tokens are keywords such as WHILE, FOR, DO or IF, identifiers such as X or NUM, operator symbols such as <,<=,+,>,>= and punctuation symbols such as parentheses or commas. The output of the lexical analyzer is a stream of tokens, which is passed to the next phase.

Syntax Analysis:

The second phase is called Syntax analysis or parser. In this phase expressions, statements, declarations etc… are identified by using the results of lexical analysis. It takes the token produced by lexical analysis as input and generates a parse tree (or syntax tree). In this phase, token arrangements are checked against the source code grammar, i.e., the parser checks if the expression made by the tokens is syntactically correct.

Semantic Analysis:

Semantic analysis checks whether the parse tree constructed follows the rules of language. For example, assignment of values is between compatible data types, and adding string to an integer. Also, the semantic analyzer keeps track of identifiers, their types and expressions; whether identifiers are declared before use or not, etc. The semantic analyzer produces an annotated syntax tree as an output.

Intermediate Code Generations:

After semantic analysis, the compiler generates an intermediate code of the source code for the target machine. It represents a program for some abstract machine. It is in between the high-level language and the machine language. This intermediate code should be generated in such a way that it makes it easier to be translated into the target machine code. This phase bridges the analysis and synthesis phases of translation.

Code Optimization:

The next phase does code optimization of the intermediate code. Optimization can be assumed as something that removes unnecessary code lines, and arranges the sequence of statements in order to speed up the program execution without wasting resources (CPU, memory). 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.

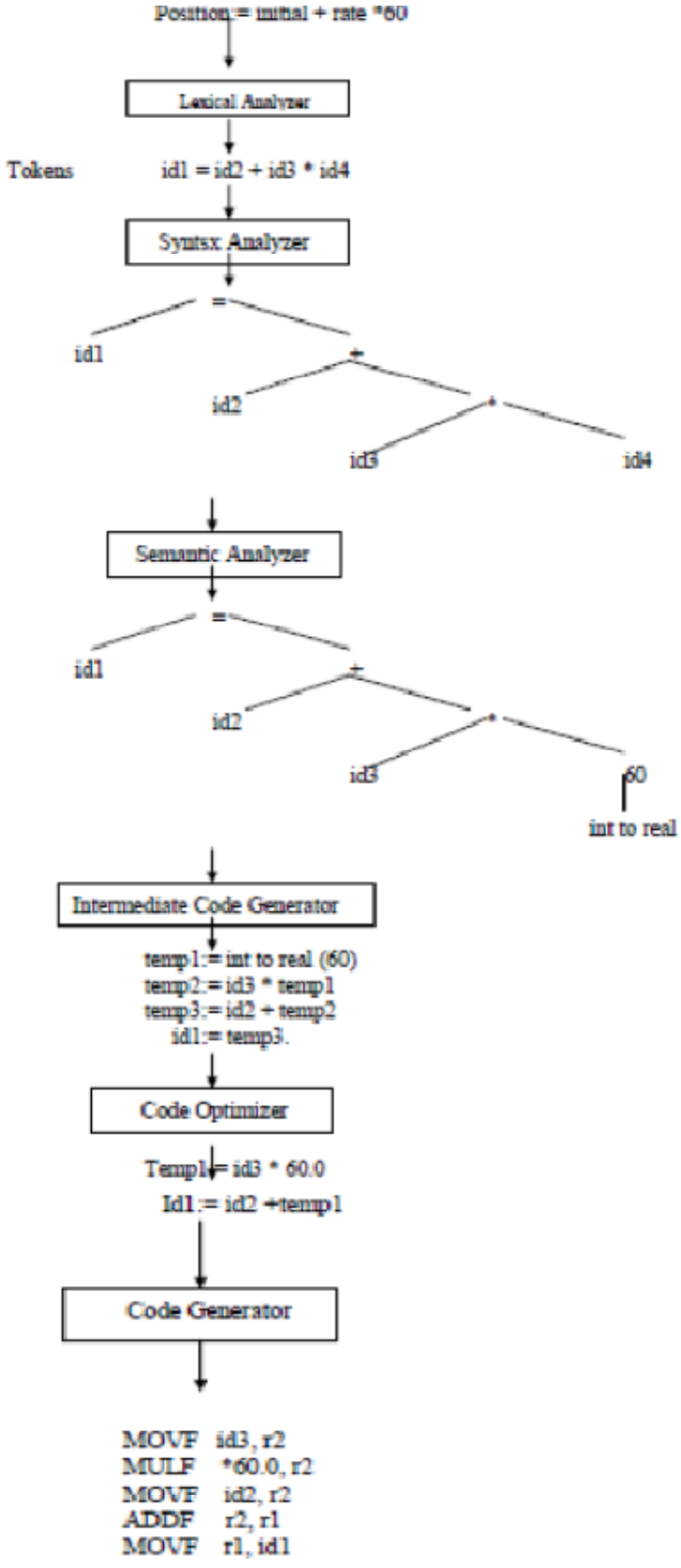
Table Management (or) Book-keeping:

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'. It is a data-structure maintained throughout all the phases of a compiler. All the identifiers' names along with their types are stored here. The symbol table makes it easier for the compiler to quickly search the identifier record and retrieve it. The symbol table is also used for scope management.

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.

Example:

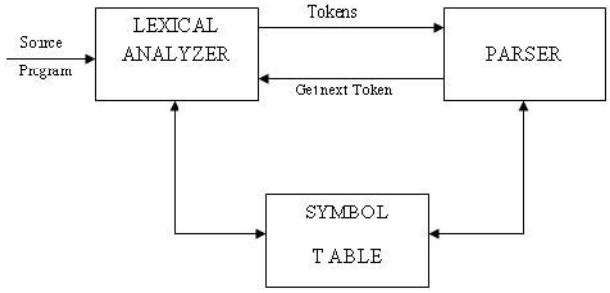

Position := initial + rate *60
Lexical Analyzer
Tokens
id1 = id2 + id3 * id4
Syntax Analyzer
=
id1
+
id2
*
id3
id4
Semantic Analyzer
=
id1
+
id2
*
id3
60
int to real
Intermediate Code Generator
temp1:= int to real (60)
temp2:= id3 * temp1
temp3:= id2 + temp2
id1:= temp3.
Code Optimizer
Temp1:= id3 * 60.0
Id1:= id2 +temp1
Code Generator
MOVF id3, r2
MULF *60.0, r2
MOVF id2, r2
ADDF r2, r1
MOVF r1, id1

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.

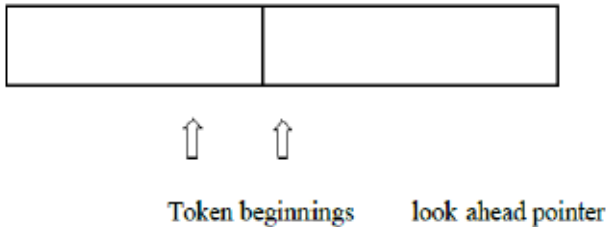
INPUT BUFFERING

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 haves(halfs) 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. 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, if the 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 grouped into 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.

```
if forward at end of first half then begin
        Reload second half;
        forward := forward+1
end
else if forward at end of second half then begin
        Reload first half;
        Move forward to beginning of first half
end
else forward := forward+1;
        Code to advance forward pointer
```

The above code requires two tests for each advance of the forward pointer. We can reduce the two to one if we extend each buffer half to hold a ***sentinel*** character at the end. The ***sentinel*** is a special character that cannot be part of the source program. A natural choice is **eof**;

: : : E: : =: : M: *: **eof**	C : * : * : 2 : **eof**: : : : **eof**

Most of the time the code performs only one test to see whether *forward* points to an **eof.**

```
forward := forward+1;
if forward ↑= eof then begin
        if forward at end of first half then begin
                Reload second half;
                forward := forward+1
        end
        else if forward at end of second half then begin
                Reload first half;
                Move forward to beginning of first half
        end
        else terminate lexical analysis
end
        Lookahead code with sentinels
```

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.

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

LEXICAL ERRORS

Lexical errors are the errors thrown by your lexer when unable to continue. Which means that there is no way to recognize 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:

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

REGULAR EXPRESSIONS

Regular expression is a formula that describes a possible set of string. Component of regular expression..

X	**the character x**
.	**any character, usually accept a new line**
[x y z]	**any of the characters x, y, z, …..**
R?	**a R or nothing (=optionally as R)**
R*	**zero or more occurrences…..**
R+	**one or more occurrences ……**
R1R2	**an R1 followed by an R2**
R1|R2	**either an R1 or an R2.**

A token is either a single string or one of a collection of strings of a certain type. If we view the set of strings in each token class as an language, we can use the regular-expression notation to describe tokens. Consider an identifier, which is defined to be a letter followed by zero or more letters or digits.

In regular expression notation we would write.

Identifier = letter (letter | digit)*

Here are the rules that define the regular expression over alphabet.

- is a regular expression denoting { € }, that is, the language containing only the empty string.
- For each 'a' in Σ, is a regular expression denoting { a }, the language with only one string consisting of the single symbol 'a' .
- If R and S are regular expressions, then

(R) | (S) means L(r) U L(s)
R.S means L(r).L(s)
R* denotes L(r*)

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)*

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 examines the input string and finds a prefix that is a lexeme matching one of the patterns.

Stmt →if expr then stmt
| If expr then stmt else stmt
| ϵ

Expr →term relop term
| term

Term →id
|number

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
WS	--	-
if	**if**	-
then	**then**	-
else	**else**	-
id	**id**	Pointer to table entry
num	**num**	Pointer to table entry

<	**relop**	LT
<=	**relop**	LE
=	**relop**	EQ
< >	**relop**	NE
>	**relop**	GT
>=	**relop**	GE

TRANSITION DIAGRAM:

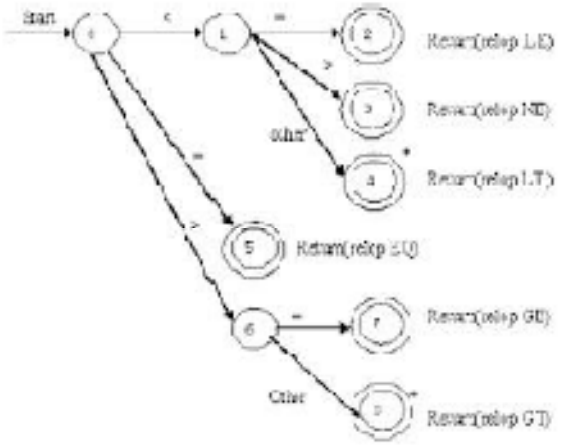
Transition Diagram has a collection of nodes or circles, called states. Each state represents a condition that could occur during the process of scanning the input looking for a lexeme that matches one of several patterns.

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.

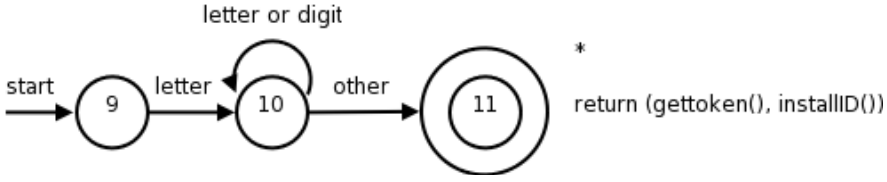
Some important conventions about transition diagrams are

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.



Transition diagram of Relational operators

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.

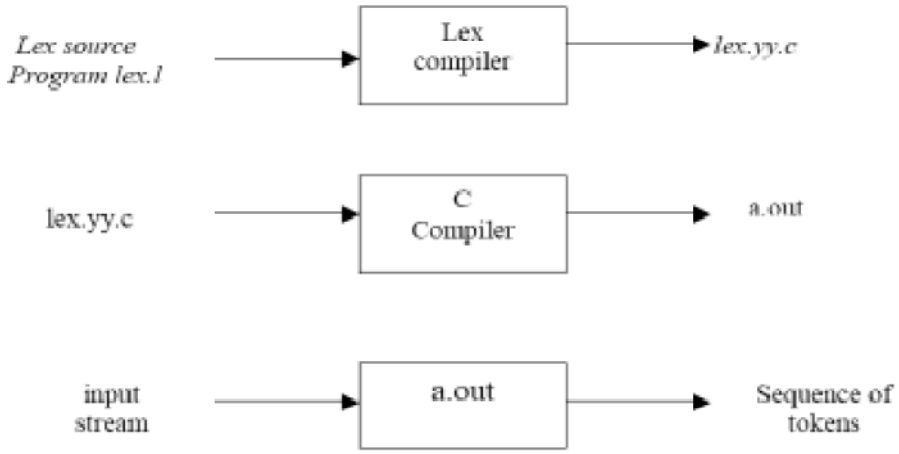


The above TD for an identifier, defined to be a letter followed by any no of letters or digits. A sequence of transition diagram can be converted into program to look for the tokens specified by the diagrams. Each state gets a segment of code.

```
case 9: c=nextchar( );
        if (isletter( c )) state=10;
        else state= fail();
        break;
case 10: c=nextchar();
         if( isletter( c )) state=10;
         else if(isdidit( c )) state=10;
         else state=Fail();
         break;
case 11: retract(1);
         install_id();
         return (gettoken( ));
```

A LANGUAGE FOR SPECIFYING LEXICAL ANLYZERS (OR) LEXICAL ANALYZER GENERATOR (OR) LEX TOOL

We describe a particular tool, called Lex, that has been widely used ti specify lexical analyzers for a variety of languages. We refer to the tool as the Lex compiler, and to its specification as the Lex language. 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. The program lex.yy.c consists of a tabular representation of a transition diagram constructed from the regular expressions of lex.l, together with a standard routine that uses the table to recognize lexemes. 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 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}*
……
……
Pn {*action n*}

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.

Lex program for tokens:

```
%{
        / *definitions of manifest constants  LT,LE,EQ,NE,GT,GE,IF,THEN,ID,
        NUMBER,RELOP */
%}

                /* regular definitions */
                Delim           [     \t \n]
                WS              {delim}+
                Letter          [A-Za-z]
                Digit           [0-9]
                Id              {letter}({letter}|{digit})*
                Number                  {digit}+(\.{digit}+)?(E[+/-]?{digit}+)?

%%              //    TRANSLATION RULES

                {WS}            {/ *no action and no return* /}
                If              {return (IF);}
                Then            {return(THEN);}
                Else            {return(ELSE);}
                {id}            {yylval  =install_id();return(ID);}
                {number}        {yylval = install_num();return(NUMBER);}

                " < "    { yylval  = LT ; return( relop);}
                " <= "   { yylval  = LE ; return( relop);}
                " ="     { yylval  = EQ ; return( relop);}
                " <>"    { yylval  = NE ; return( relop);}
                ">"      { yylval  = GT ; return( relop);}
                ">="     { yylval  = GE ; return( relop);}
%%
```

```
                                    //  AUXILLARY PROCEDURES
Install_id(){
        /* procedure to install the lexeme,whose first
        character is pointed to by yytext and whose length is
        yyleng, into symbol table and return a pointer thereto*/
}
Install_num(){
        /* similar procedure to install a lexeme that is a number */
}
```

SYNTAX ANALYSIS

ROLE OF THE PARSER

Parser for any grammar is program that takes as input string w (obtain set of strings tokens from the lexical analyzer) and produces as output either a parse tree for w , if w is a valid sentences of grammar or error message indicating that w is not a valid sentences of given grammar. The goal of the parser is to determine the syntactic validity of a source string is valid; a tree is built for use by the subsequent phases of the computer. The tree reflects the sequence of derivations or reduction used during the parser. Hence, it is called parse tree. If string is invalid, the parse has to issue diagnostic message identifying the nature and cause of the errors in string. Every elementary sub tree in the parse tree corresponds to a production of the grammar.

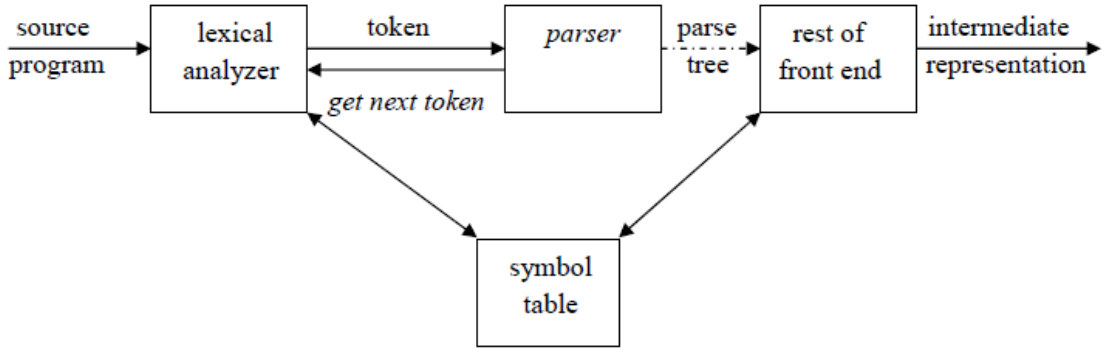
There are two ways of identifying an elementary sub tree:

1. By deriving a string from a non-terminal or
2. By reducing a string of symbol to a non-terminal.

The two types of parsers employed are:

a. Top down parser: which build parse trees from top(root) to bottom(leaves)

b. Bottom up parser: which build parse trees from leaves and work up the root.



CONTEXT FREE GRAMMARS

Inherently recursive structures of a programming language are defined by a context-free Grammar. In a context-free grammar, we have four triples G=(V,T,P,S).

Here, V is finite set of terminals (in our case, this will be the set of tokens)

T is a finite set of non-terminals (syntactic-variables)

P is a finite set of productions rules in the following form

$A \rightarrow \alpha$ where A is a non-terminal and α is a string of terminals and non-terminals (Including the empty string)

S is a start symbol (one of the non-terminal symbol)

L(G) is the language of G (the language generated by G) which is a set of sentences. A sentence of L(G) is a string of terminal symbols of G. If S is the start symbol of G then ω is a sentence of L(G) iff $S \rightarrow \omega$ where ω is a string of terminals of G. If G is a context free grammar, L(G) is a context-free language. Two grammars G1 and G2 are equivalent, if they produce same grammar.

Consider the production of the form $S \rightarrow \alpha$, If α contains non-terminals, it is called as a Sentential form of G. If α does not contain non-terminals, it is called as a sentence of G.

DERIVATIONS

In general a derivation step is

$\alpha A\beta \rightarrow \alpha\gamma\beta$ is sentential form and if there is a production rule $A \rightarrow \gamma$ in our grammar. where α and β are arbitrary strings of terminal and non-terminal symbols $\alpha 1 \rightarrow \alpha 2 \rightarrow \ldots \rightarrow \alpha n$ (αn derives from $\alpha 1$ or $\alpha 1$ derives αn).

There are two types of derivation

1 At each derivation step, we can choose any of the non-terminals in the sentential form of G for the replacement.

2 If we always choose the left-most non-terminal in each derivation step, this derivation is called as left-most derivation.

Example:

E → E + E | E – E | E * E | E / E | - E

E → (E)

E → id

Leftmost derivation :

E → E + E→ E * E+E →id* E+E→id*id+E→id*id+id

The string is derive from the grammar w= id*id+id, which is consists of all terminal symbols

Rightmost derivation:

E → E + E→ E+E * E→E+ E*id→E+id*id→id+id*id

Given grammar G : E → E+E | E*E | (E) | - E | id

Sentence to be derived : – (id+id)

LEFTMOST DERIVATION	**RIGHTMOST DERIVATION**
E → - E	E → - E
E → - (E)	E → - (E)
E → - (E+E)	E → - (E+E)
E → - (id+E)	E → - (E+id)
E → - (id+id)	E → - (id+id)

• String that appear in leftmost derivation are called left sentinel forms.

• String that appear in rightmost derivation are called right sentinel forms.

Sentinels:

• Given a grammar G with start symbol S, if S → α, where α may contain non terminals or terminals, and then α is called the sentinel form of G.

Yield or frontier of tree:

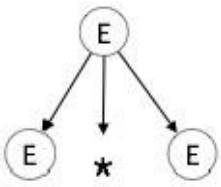
• Each interior node of a parse tree is a non-terminal. The children of node can be a terminal or non-terminal of the sentinel forms that are read from left to right. The sentinel form in the parse tree is called yield or frontier of the tree.

PARSE TREE

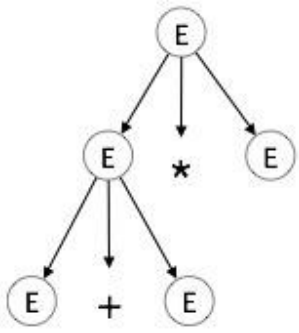
• Inner nodes of a parse tree are non-terminal symbols.

• The leaves of a parse tree are terminal symbols.

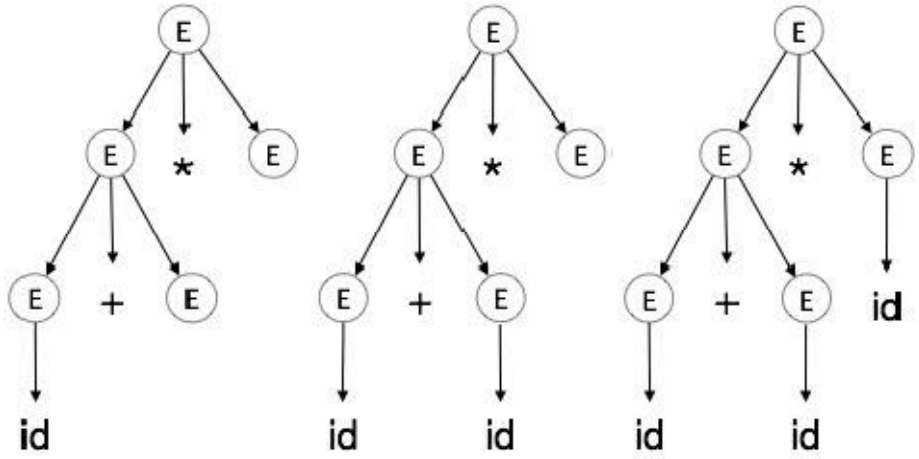
• A parse tree can be seen as a graphical representation of a derivation.



E → E * E



E → E + E * E



E → id + E * E **E → id + id * E** **E → id + id * id**

Ambiguity:

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

Example : Given grammar G : E → E+E | E*E | (E) | - E | id

The sentence id+id*id has the following two distinct leftmost derivations:

E → E+ E	E → E* E
E → id + E	E → E + E * E
E → id + E * E	E → id + E * E
E → id + id * E	E → id + id * E
E → id + id * id	E → id + id * id

The two corresponding parse trees are:

Example:

To disambiguate the grammar E → E+E | E*E | E^E | id | (E), we can use precedence of operators as follows:

^ (right to left)
/,* (left to right)
-,+ (left to right)

Eliminating Left Recursion:

A grammar is said to be left recursive if it has a non-terminal A such that there is a derivation A=>Aα for some string α. Top-down parsing methods cannot handle left-recursive grammars.

Hence, left recursion can be eliminated as follows:

If there is a production A → Aα | β it can be replaced with a sequence of two productions

A → βA' **A' → αA' | ε**

Without changing the set of strings derivable from A.

Example: Consider the following grammar for arithmetic expressions:

E → E+T | T
T → T*F | F
F → (E) | id

First eliminate the left recursion for E as

E → TE'
E' → +TE' | ε

Then eliminate for T as

T → FT'
T'→ *FT' | ε

Thus the obtained grammar after eliminating left recursion is

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

Algorithm to eliminate left recursion:

1. Arrange the non-terminals in some order A1, A2 . . . An.
2. for i := 1 to n do begin
   for j := 1 to i-1 do begin
   replace each production of the form Ai → Aj γ
   by the productions Ai → δ1 γ | δ2γ | . . . | δk γ
   where Aj → δ1 | δ2 | . . . | δk are all the current Aj-productions;
   end
   eliminate the immediate left recursion among the Ai-productions

end

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.

If there is any production A → αβ1 | αβ2 , it can be rewritten as

A → αA'
A' → β1 | β2

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

Left factored, this grammar becomes

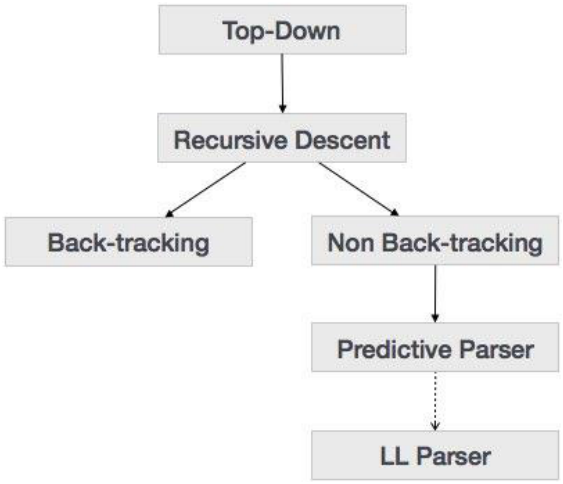
S → iEtSS' | a
S' → eS | ε
E → b

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



1. RECURSIVE DESCENT PARSING

- Recursive descent parsing is one of the top-down parsing techniques that uses a set of recursive procedures to scan its input.
- This parsing method may involve backtracking, that is, making repeated scans of the input.

Example for backtracking:

Consider the grammar G : S → cAd

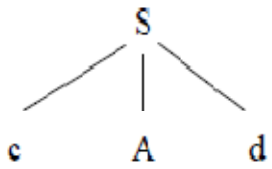
A → ab | a

and the input string w=cad.

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

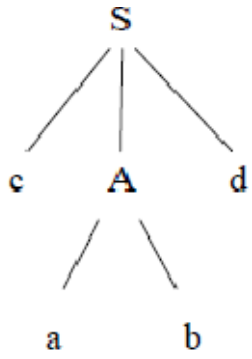
Step1:

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.

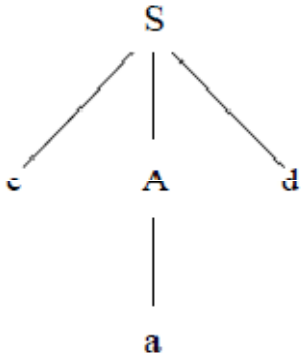


Step2:

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 position. This is called backtracking.
Step4:
Now try the second alternative for A.



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

Example for recursive decent parsing:

A left-recursive grammar can cause a recursive-descent parser to go into an infinite loop.
Hence, elimination of left-recursion must be done before parsing.
Consider the grammar for arithmetic expressions

E → E+T | T
T → T*F | F
F → (E) | id

After eliminating the left-recursion the grammar becomes,

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

Now we can write the procedure for grammar as follows:
Recursive procedure:

```
Procedure E()
begin
        T( );
        EPRIME( );
end
Procedure EPRIME( )
begin
        If input_symbol='+' then
        ADVANCE( );
        T( );
        EPRIME( );
end
Procedure T( )
begin
        F( );
        TPRIME( );
end
```

```
Procedure TPRIME( )
begin
        If input_symbol='*' then
        ADVANCE( );
        F( );
        TPRIME( );
end
Procedure F( )
        If input-symbol='id' then
                ADVANCE( );
        else if input-symbol='(' then
        begin
                ADVANCE( );
                E( );
                else if input-symbol=')' then
                ADVANCE( );
                else ERROR()
        end
else ERROR( );
```

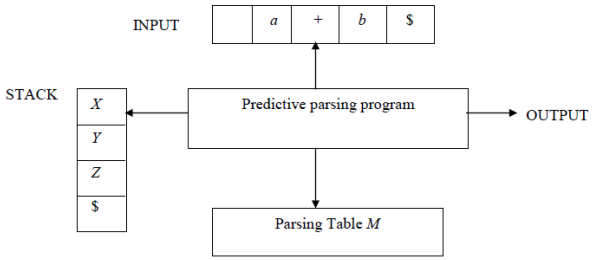
Stack implementation:

PROCEDURE	INPUT STRING
E( )	id+id*id
T( )	id+id*id
F( )	id+id*id
ADVANCE( )	id+id*id
TPRIME( )	id+id*id
EPRIME( )	id+id*id
ADVANCE( )	id+id*id
T( )	id+id*id
F( )	id+id*id
ADVANCE( )	id+id*id
TPRIME( )	id+id*id
ADVANCE( )	id+id*id
F( )	id+id*id
ADVANCE( )	id+id*id
TPRIME( )	id+id*id

PREDICTIVE PARSING

- ✓ Predictive parsing is a special case of recursive descent parsing where no backtracking is required.
- ✓ The key problem of predictive parsing is to determine the production to be applied for a non-terminal in case of alternatives.

Non-recursive predictive parser:



The table-driven predictive parser has an input buffer, stack, a parsing table and an output stream.

Input buffer:

It consists of strings to be parsed, followed by $ to indicate the end of the input string.

Stack:

It contains a sequence of grammar symbols preceded by $ to indicate the bottom of the stack. Initially, the stack contains the start symbol on top of $.

Parsing table:

It is a two-dimensional array M[A, a], where 'A' is a non-terminal and 'a' is a terminal.

Predictive parsing program:

The parser is controlled by a program that considers X, the symbol on top of stack, and a, the current input symbol. These two symbols determine the parser action. 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 non-terminal , the program consults entry M[X, a] of the parsing table M. This entry will either be an X-production of the grammar or an error entry.

If M[X, a] = {X → UVW},the parser replaces X on top of the stack by UVW

If M[X, a] = error, the parser calls an error recovery routine.

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 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 as follows:

```
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 →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 → Y1 Y2 . . . Yk
        end
        else error()
until X = $
```

Predictive parsing table construction:

The construction of a predictive parser is aided by two functions associated with a grammar G :

1. FIRST
2. FOLLOW

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 entries are single entries. So each location has not more than one entry. This type of grammar is called LL(1) grammar.

Consider this following grammar:

S → iEtS | iEtSeS | a

E → b

After eliminating left factoring, we have

S → iEtSS' | a

S'→ eS | ε

E → b

To construct a parsing table, we need FIRST() and FOLLOW() for all the non-terminals.

FIRST(S) = { i, a }
FIRST(S') = {e, ε }
FIRST(E) = { b}
FOLLOW(S) = { $,e }
FOLLOW(S') = { $,e }
FOLLOW(E) = {t}

Parsing table:

NON-TERMINAL	a	b	e	i	t	$
S	S → a			S → iEtSS'		
S'			S' → eS S' → ε			S' → ε
E		E → b				

Since there are more than one production, the grammar is not LL(1) grammar.

Actions performed in predictive parsing:

1. Shift
2. Reduce
3. Accept
4. Error

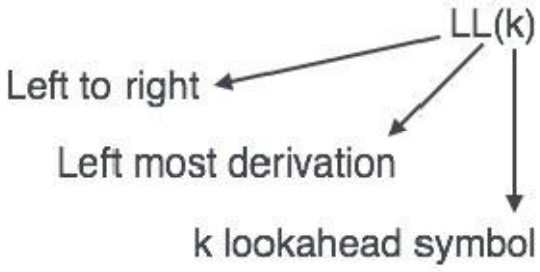
Implementation of predictive parser:

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.

LL Parser

An LL Parser accepts LL grammar. LL grammar is a subset of context-free grammar but with some restrictions to get the simplified version, in order to achieve easy implementation. LL grammar can be implemented by means of both algorithms, namely, recursive-descent or table-driven.

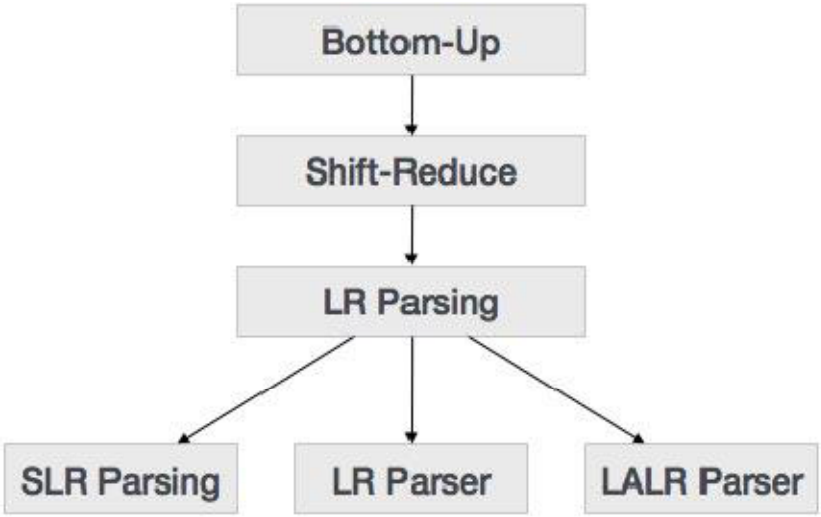
LL parser is denoted as LL(k). The first L in LL(k) is parsing the input from left to right, the second L in LL(k) stands for left-most derivation and k itself represents the number of look aheads. Generally k = 1, so LL(k) may also be written as LL(1).



SYNTAX ANALYSIS

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. Bottom-up parsing starts from the leaf nodes of a tree and works in upward direction till it reaches the root node. Here, we start from a sentence and then apply production rules in reverse manner in order to reach the start symbol. The image given below depicts the bottom-up parsers available.



A general type of bottom-up parser is a shift-reduce parser.

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	S → aABe
aAbcde (A→b)	→ aAde
aAde (A→Abc)	→ aAbcde
aABe (B→d)	→ 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 $id_1+id_2*id_3$

The rightmost derivation is :

E → **E+E**
→ E+**E*E**
→ E+E ***id**$_3$**
→ E+**id**$_2$*id$_3$
→ **id**$_1$+id$_2$*id$_3$

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 $w = \gamma_n$, where γ_n is the n^{th} right-sentinel form of some rightmost derivation.

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

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 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

1. Shift-reduce conflict:

Example:

Consider the grammar:

E→E+E | E*E | id and input id+id*id

Stack	Input	Action	Stack	Input	Action
$ 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

and input c+c

R→ 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	$				

OPERATOR-PRECEDENCE PARSING

Precedence Relations

Bottom-up parsers for a large class of context-free grammars can be easily developed using operator grammars. Operator grammars have the property that no production right side is empty or has two adjacent non terminals. This property enables the implementation of efficient operator-precedence parsers. This parser relies on the following three precedence relations:

Relation	Meaning
a <· b	a yields precedence to b
a =· b	a has the same precedence as b
a ·> b	a takes precedence over b

These operator precedence relations allow to delimit the handles in the right sentential forms: <· marks the left end, =· appears in the interior of the handle, and ·> marks the right end.

	id	+	*	$
id		·>	·>	·>
+	<·	·>	<·	·>
*	<·	·>	·>	·>
$	<·	<·	<·	·>

Example: The input string:

id1 + id2 * id3

after inserting precedence relations becomes

$ <· id1 ·> + <· id2 ·> * <· id3 ·> $

Having precedence relations allows to identify handles as follows:

- scan the string from left until seeing ·>
- scan backwards the string from right to left until seeing <·
- everything between the two relations <· and ·> forms the handle

OPERATOR PRECEDENCE PARSING ALGORITHM

Initialize: 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 $ is on the top of the stack and ip points to $ then return
else
Let a be the top terminal on the stack, and b the symbol pointed to by ip
if a <· b or a =· b then
push b onto the stack
advance ip to the next input symbol
else if a ·> b then
repeat
pop the stack
until the top stack terminal is related by <· to the terminal most recently popped
else error()
end

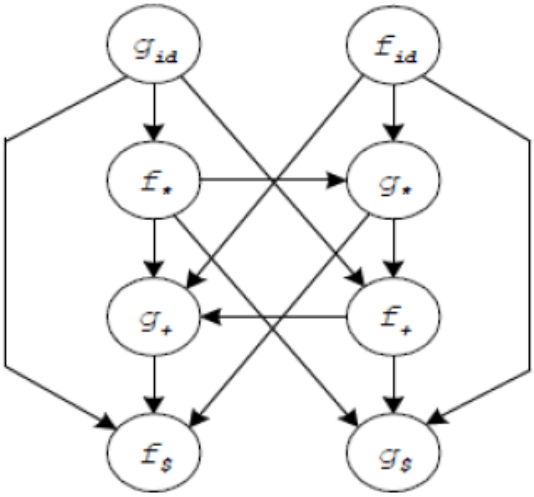
ALGORITHM FOR CONSTRUCTING PRECEDENCE FUNCTIONS

1. Create functions fa for each grammar terminal a and for the end of string symbol;
2. Partition the symbols in groups so that fa and gb are in the same group if a =· b (there can be symbols in the same group even if they are not connected by this relation)
3. Create a directed graph whose nodes are in the groups, next for each symbols a and b do: place an edge from the group of gb to the group of fa if a <· b, otherwise if a ·> b place an edge from the group of fa to that of gb;
4. If the constructed graph has a cycle then no precedence functions exist. When there are no cycles collect the length of the longest paths from the groups of fa and gb.

Example:

	id	+	*	$
id		·>	·>	·>
+	<·	·>	<·	·>
*	<·	·>	·>	·>
$	<·	<·	<·	·>

Consider the above table Using the algorithm leads to the following graph:



LR Parser

The LR parser is a non-recursive, shift-reduce, bottom-up parser. It uses a wide class of context-free grammar which makes it the most efficient syntax analysis technique. LR parsers are also known as LR(k) parsers, where L stands for left-to-right scanning of the input stream; R stands for the construction of right-most derivation in reverse, and k denotes the number of look ahead symbols to make decisions.

There are three widely used algorithms available for constructing an LR parser:

SLR(1) – Simple LR Parser:

- ✓ Works on smallest class of grammar
- ✓ Few number of states, hence very small table
- ✓ Simple and fast construction

LR(1) – LR Parser:

- ✓ Works on complete set of LR(1) Grammar
- ✓ Generates large table and large number of states
- ✓ Slow construction

LALR(1) – Look-Ahead LR Parser:

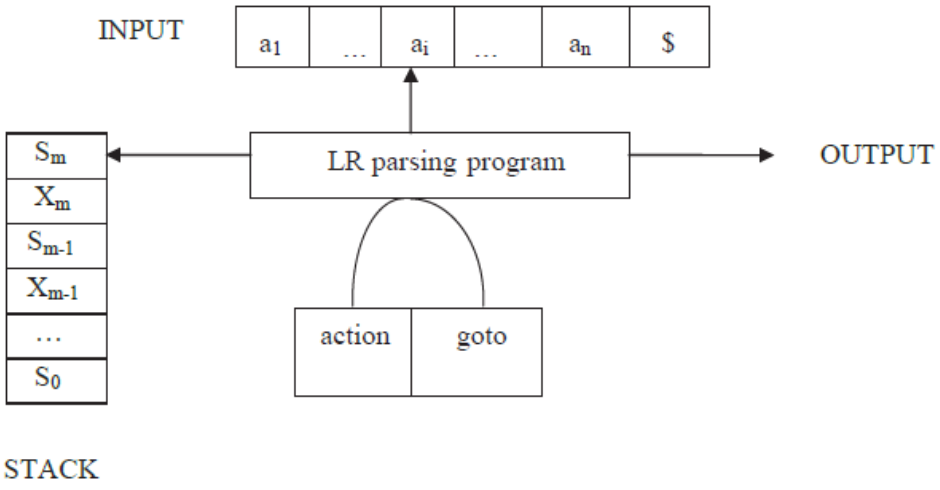
- ✓ Works on intermediate size of grammar
- ✓ Number of states are same as in SLR(1)

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:



It consists of : an input, an output, a stack, a driver program, and a parsing table that has two parts (*action* and *goto*).

- ➢ The driver program is the same for all LR parser.
- ➢ The parsing program reads characters from an input buffer one at a time.
- ➢ The program uses a stack to store a string of the form $s_0X_1s_1X_2s_2\ldots X_ms_m$, where s_m is on top. Each X_i is a grammar symbol and each s_i is a state.
- ➢ The parsing table consists of two parts : *action* and *goto* functions.

Action : The parsing program determines s_m, the state currently on top of stack, and a_i, the current input symbol. It then consults *action*$[s_m,a_i]$ 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 $A \rightarrow \beta$,
3. accept, and
4. error.

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

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 → XYZ yields the four items :

A → . XYZ
A → X . YZ
A → XY . Z
A → 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 → α . Bβ is in closure(I) and B → γ is a production, then add the item B → . γ 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→ αX . β] such that [A→ α . Xβ] 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 = $\{I_0, I_1, \ldots I_n\}$, the collection of sets of LR(0) items for G'.
2. State i is constructed from I_i. The parsing functions for state i are determined as follows:
   (a) If [A→α·aβ] is in I_i and goto(I_i,a) = I_j, then set *action*[i,a] to "shift j". Here a must be terminal.
   (b) If [A→α·] is in I_i, then set *action*[i,a] to "reduce A→α" for all a in FOLLOW(A).
   (c) If [S'→S.] is in I_i, 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-terminals A using the rule: If *goto*(I_i,A) = I_j, 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 set of items containing [S'→.S]

Example for SLR parsing:

Construct SLR parsing for the following grammar :

G : E → E + T | T
T → T * F | F
F → (E) | id

The given grammar is :

G : E → E + T ------ (1)
E → T ------ (2)
T → T * F ------ (3)
T → F ------ (4)
F → (E) ------ (5)
F → id ------ (6)

Step 1 : Convert given grammar into augmented grammar.

Augmented grammar :

E' → E
E → E + T
E → T
T → T * F
T → F
F → (E)
F → id

Step 2 : Find LR (0) items.

I_0 : E' → . E
E → . E + T
E → . T
T → . T * F
T → . F
F → . (E)
F → . id

GOTO (I_0 , E)
I_1 : E' → E .
E → E . + T

GOTO (I_4 , id)
I_5 : F → id .

GOTO (I_0 , T)
I_2 : E → T .
T → T . * F

GOTO (I_0 , F)
I_3 : T → F .

GOTO (I_0 , ( )
I_4 : F → (. E)
E → . E + T
E → . T
T → . T * F
T → . F
F → . (E)
F → . id

GOTO (I_0 , id)
I_5 : F → id .

GOTO (I_1 , +)
I_6 : E → E + . T
T → . T * F
T → . F
F → . (E)
F → . id

GOTO (I_2 , *)
I_7 : T → T * . F
F → . (E)
F → . id

GOTO (I_4 , E)
I_8 : F → (E .)
E → E . + T

GOTO (I_4 , T)
I_2 : E → T .
T → T . * F

GOTO (I_4 , F)
I_3 : T → F .

GOTO (I_6 , T)
I_9 : E → E + T .
T → T . * F

GOTO (I_6 , F)
I_3 : T → F .

GOTO (I_6 , ( )
I_4 : F → (. E)

GOTO (I_6 , id)
I_5 : F → id .

GOTO (I_7 , F)
I_{10} : T → T * F .

GOTO (I_7 , ( )
I_4 : F → (. E)
E → . E + T
E → . T
T → . T * F
T → . F
F → . (E)
F → . id

GOTO (I_7 , id)
I_5 : F → id .

GOTO (I_8 ,))
I_{11} : F → (E) .

GOTO (I_8 , +)
I_6 : E → E + . T
T → . T * F
T → . F
F → . (E)
F → . id

GOTO (I_9 , *)
I_7 : T → T * . F
F → . (E)
F → . id

GOTO (I_4 , ( )

I_4 : F → (. E)

E → . E + T

E → . T

T → . T * F

T → . F

F → . (E)

F → id

FOLLOW (E) = { $,) , +)

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

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

SLR parsing table:

	ACTION						GOTO		
	id	**+**	*****	**(**	**)**	**$**	**E**	**T**	**F**
I_0	s5			s4			1	2	3
I_1		s6				ACC			
I_2		r2	s7		r2	r2			
I_3		r4	r4		r4	r4			
I_4	s5			s4			8	2	3
I_5		r6	r6		r6	r6			
I_6	s5			s4				9	3
I_7	s5			s4					10
I_8		s6			s11				
I_9		r1	s7		r1	r1			
I_{10}		r3	r3		r3	r3			
I_{11}		r5	r5		r5	r5			

Blank entries are error entries.

Stack implementation:

Check whether the input **id + id * id** is valid or not.

STACK	INPUT	ACTION
0	id + id * id $	GOTO (I_0 , id) = s5 ; **shift**
0 id 5	+ id * id $	GOTO (I_5 , +) = r6 ; **reduce** by F→id
0 F 3	+ id * id $	GOTO (I_0 , F) = 3 GOTO (I_3 , +) = r4 ; **reduce** by T → F
0 T 2	+ id * id $	GOTO (I_0 , T) = 2 GOTO (I_2 , +) = r2 ; **reduce** by E → T
0 E 1	+ id * id $	GOTO (I_0 , E) = 1 GOTO (I_1 , +) = s6 ; **shift**
0 E 1 + 6	id * id $	GOTO (I_6 , id) = s5 ; **shift**
0 E 1 + 6 id 5	* id $	GOTO (I_5 , *) = r6 ; **reduce** by F → id
0 E 1 + 6 F 3	* id $	GOTO (I_6 , F) = 3 GOTO (I_3 , *) = r4 ; **reduce** by T → F
0 E 1 + 6 T 9	* id $	GOTO (I_6 , T) = 9 GOTO (I_9 , *) = s7 ; **shift**
0 E 1 + 6 T 9 * 7	id $	GOTO (I_7 , id) = s5 ; **shift**
0 E 1 + 6 T 9 * 7 id 5	$	GOTO (I_5 , $) = r6 ; **reduce** by F → id
0 E 1 + 6 T 9 * 7 F 10	$	GOTO (I_7 , F) = 10 GOTO (I_{10} , $) = r3 ; **reduce** by T → T * F
0 E 1 + 6 T 9	$	GOTO (I_6 , T) = 9 GOTO (I_9 , $) = r1 ; **reduce** by E → E + T
0 E 1	$	GOTO (I_0 , E) = 1 GOTO (I_1 , $) = **accept**

LL vs. LR

LL	LR
Does a leftmost derivation.	Does a rightmost derivation in reverse.
Starts with the root nonterminal on the stack.	Ends with the root nonterminal on the stack.
Ends when the stack is empty.	Starts with an empty stack.
Uses the stack for designating what is still to be expected.	Uses the stack for designating what is already seen.
Builds the parse tree top-down.	Builds the parse tree bottom-up.
Continuously pops a nonterminal off the stack, and pushes the corresponding right hand side.	Tries to recognize a right hand side on the stack, pops it, and pushes the corresponding nonterminal.
Expands the non-terminals.	Reduces the non-terminals.
Reads the terminals when it pops one off the stack.	Reads the terminals while it pushes them on the stack.
Pre-order traversal of the parse tree.	Post-order traversal of the parse tree.

CANONICAL LR PARSING

By splitting states when necessary, we can arrange to have each state of an LR parser indicate exactly which input symbols can follow a handle a for which there is a possible reduction to *A*. As the text points out, sometimes the FOLLOW sets give too much information and doesn't (can't) discriminate between different reductions.

The general form of an LR(k) item becomes [A -> a.b, s] where A -> ab is a production and s is a string of terminals. The first part (A -> a.b) is called the core and the second part is the look ahead. In LR(1) |s| is 1, so s is a single terminal.

A -> ab is the usual right hand side with a marker; any a in s is an incoming token in which we are interested. Completed items used to be reduced for every incoming token in FOLLOW(A), but now we will reduce only if the next input token is in the look ahead set s..if we get two productions A -> a and B -> a, we can tell them apart when a is a handle on the stack if the corresponding completed items have different look ahead parts. Furthermore, note that the look ahead has no effect for an item of the form [A -> a.b, a] if b is not e. Recall that our problem occurs for completed items, so what we have done now is to say that an item of the form [A -> a., a] calls for a reduction by A -> a only if the next input symbol is a.

More formally, an LR(1) item [A -> a.b, a] is valid for a viable prefix g if there is a derivation

S =>* s abw, where g = sa, and either a is the first symbol of w, or w is e and a is $.

ALGORITHM FOR CONSTRUCTION OF THE SETS OF LR(1) ITEMS

Input: grammar G'
Output: sets of LR(1) items that are the set of items valid for one or more viable prefixes of G'
Method:

```
closure(I)
begin
        repeat
                for each item [A -> a.Bb, a] in I,
                each production B -> g in G',
                and each terminal b in FIRST(ba)
                such that [B -> .g, b] is not in I do
                add [B -> .g, b] to I;
        until no more items can be added to I;
end;
```

goto(I, X)

```
begin
let J be the set of items [A -> aX.b, a] such that
[A -> a.Xb, a] is in I
return closure(J);
end;
procedure items(G')
begin
        C := {closure({S' -> .S, $})};
        repeat
                for each set of items I in C and each grammar symbol X such
                that goto(I, X) is not empty and not in C do
                add goto(I, X) to C
        until no more sets of items can be added to C;
end;
```

An example:

Consider the following grammer,

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/d
C->.d,c/d

I4: C->d.,c/d

I5: S->CC.,$

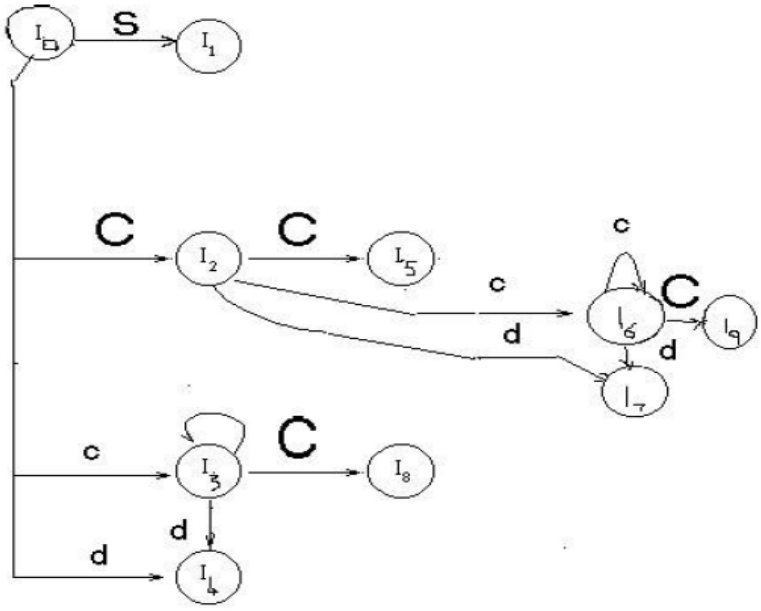
I6: C->c.C,$
C->.cC,$
C->.d,$

I7: C->d.,$

I8: C->cC.,c/d

I9: C->cC.,$

Here is what the corresponding DFA looks like



ALGORITHM FOR CONSTRUCTION OF THE CANONICAL LR PARSING TABLE

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

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 goto(Ii, a) = Ij, then set action[i, a] to "shift j". Here a 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 goto transitions for state i are constructed for all *nonterminal*s *A* using the rule: If goto(Ii, A)= Ij, then goto[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, $].

CLR Parsing Table:

State	ACTION			GOTO	
	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		

LALR PARSER:

We begin with two observations. First, some of the states generated for LR(1) parsing have the same set of core (or first) components and differ only in their second component, the look ahead symbol. Our intuition is that we should be able to merge these states and reduce the number of states we have, getting close to the number of states that would be generated for LR(0) parsing. This observation suggests a hybrid approach: We can construct the canonical LR(1) sets of items and then look for sets of items having the same core. We merge these sets with common cores into one set of items. The merging of states with common cores can never produce a shift/reduce conflict that was not present in one of the original states because shift actions depend only on the core, not the look ahead. But it is possible for the merger to produce a reduce/reduce conflict.
Our second observation is that we are really only interested in the look ahead symbol in places where there is a problem. So our next thought is to take the LR(0) set of items and add look ahead's only where they are needed. This leads to a more efficient, but much more complicated method.

ALGORITHM FOR EASY CONSTRUCTION OF AN LALR TABLE
Input: G'
Output: LALR parsing table functions with action and goto for G'.
Method:
1. Construct C = {I0, I1 , ..., In} the collection of sets of LR(1) items for G'.

2. For each core present among the set of LR(1) items, find all sets having that core and replace these sets by the union.

3. Let C' = {J0, J1 , ..., Jm} be the resulting sets of LR(1) items. The parsing actions for state i are constructed from Ji in the same manner as in the construction of the canonical LR parsing table.

4. If there is a conflict, the grammar is not LALR(1) and the algorithm fails.

5. The goto table is constructed as follows: If J is the union of one or more sets of LR(1) items, that is, J = I0U I1 U ... U Ik, then the cores of goto(I0, X), goto(I1, X), ..., goto(Ik, X) are the same, since I0, I1 , ..., Ik all have the same core. Let K be the union of all sets of items having the same core asgoto(I1, X).

6. Then goto(J, X) = K.

Consider the above example:

I3 & I6 can be replaced by their union

I36: C->c.C,c/d/$
C->.Cc,C/D/$
C->.d,c/d/$

I47: C->d.,c/d/$

I89: C->Cc.,c/d/$

Parsing Table:

State	ACTION			GOTO	
	c	**d**	**$**	**S**	**C**
0	S36	S47		1	2
1			acc		
2	S36	S47			5
36	S36	S47			89
47	r3	r3	r3		
5			r1		
89	r2	r2	r2		