UNIT - II

Syntax Analysis: Introduction, Context-Free Grammars, Writing a Grammar, Top-Down Parsing, Bottom-Up Parsing, Introduction to LR Parsing: Simple LR, More Powerful LR Parsers, Using Ambiguous Grammars and Parser Generators.

2.1 INTRODUCTION

Syntax Analysis is a second phase of the compiler design process in which the given input string is checked for the confirmation of rules and structure of the formal grammar. It analyses the syntactical structure and checks if the given input is in the correct syntax of the programming language or not.

Syntax Analysis in Compiler Design process comes after the Lexical analysis phase. It is also known as the Parse Tree or Syntax Tree. The Parse Tree is developed with the help of pre-defined grammar of the language. The syntax analyser also checks whether a given program fulfills the rules implied by a context-free grammar. If it satisfies, the parser then creates the parse tree of that source program. Otherwise, it will display error messages.

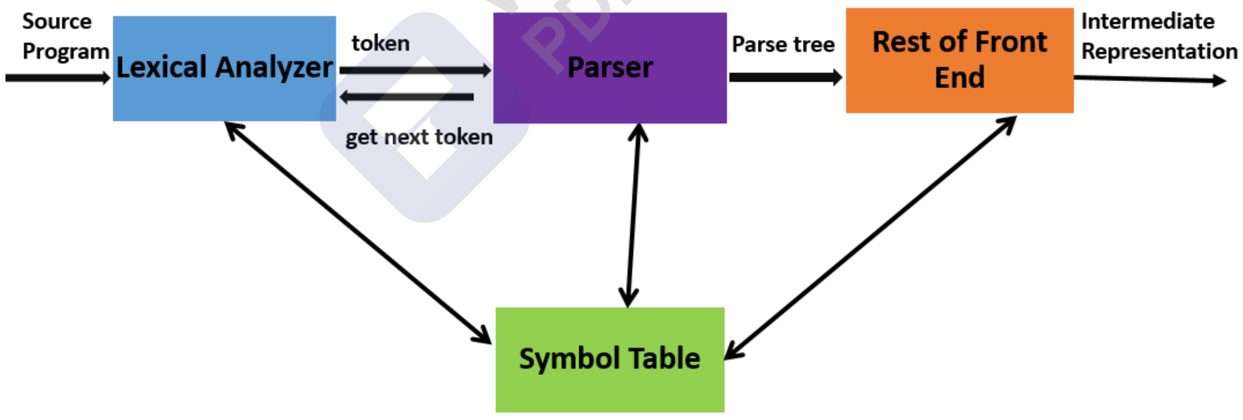


Figure: Syntax Analyser Process

2.2 CONTEXT-FREE GRAMMARS

Context-free grammars can generate Context-Free-Langauges. They do this by taking a set of variables which are defined recursively, in terms of one another, by a set of production rules. Context-free grammars are named as such because any of the production rules in the grammar can be applied regardless of context—it does not depend on any other symbols that may or may not be around a given symbol that is having a rule applied to it.

Context-free grammars have the following components:

- A set of **terminal symbols** which are the characters that appear in the language/strings generated by the grammar. Terminal symbols never appear on the left-hand side of the production rule and are always on the right-hand side.
- A set of **nonterminal symbols** (or **variables**) which are placeholders for patterns of terminal symbols that can be generated by the nonterminal symbols. These are the symbols that will always appear on the left-hand side of the production rules, though they can be included on the right-hand side. The strings that a CFG produces will contain only symbols from the set of nonterminal symbols.
- A set of **production rules** which are the rules for replacing nonterminal symbols. Production rules have the following form: variable \rightarrow→ string of variables and terminals.
- A **start symbol** which is a special nonterminal symbol that appears in the initial string generated by the grammar.

A context-free grammar can be described by a four-element Tuples Σ = (V, T, P, S) Where,

- V is a finite set of variables (which are non-terminal)
- T is a finite set of Terminal
- P is a set of production rules where each production rule maps a variable to a strings
- S is a start symbol.

Example:

L= {wcw^R | w € (a, b)*}

Production rules:

1. S → aSa
2. S → bSb
3. S → c

Now check that abbcbba string can be derived from the given CFG.

S ⇒ aSa

S ⇒ abSba

S ⇒ abbSbba

S ⇒ abbcbba

By applying the production S → aSa, S → bSb recursively and finally applying the production

S → c, we get the string abbcbba.

Derivation

A derivation is basically a sequence of production rules, in order to get the input string. During parsing, we take two decisions for some sentential form of input:

- Deciding the non-terminal which is to be replaced.
- Deciding the production rule, by which, the non-terminal will be replaced.

To decide which non-terminal to be replaced with production rule, we can have two options.

Left-most Derivation

If the sentential form of an input is scanned and replaced from left to right, it is called left-most derivation. The sentential form derived by the left-most derivation is called the left-sentential form.

Right-most Derivation

If we scan and replace the input with production rules, from right to left, it is known as right-most derivation. The sentential form derived from the right-most derivation is called the right-sentential form.

Example

Production rules:

E → E + E

E → E - E

E → id

Input string: id + id * id

The left-most derivation is:

E → E - E

E → E + E - E

E → id + E - E

E → id + id - E

E → id + id - id

Notice that the left-most side non-terminal is always processed first.

The right-most derivation is:

E → E + E

E → E + E - E

E → E + E - id

E → E + id - id

E → id + id - id

Parse Tree

A parse tree is a graphical depiction of a derivation. It is convenient to see how strings are derived from the start symbol. The start symbol of the derivation becomes the root of the parse tree. For the string **id + id – id**, to construct two parse trees: **Left-Most -Derivation** and **Right- Most -Derivation**

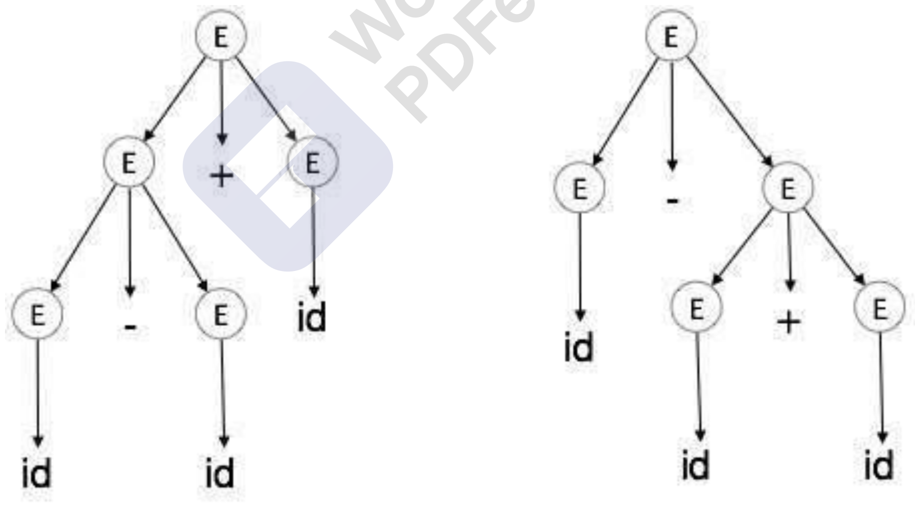


Figure: Left-Most -Derivation and Right- Most -Derivation

2.3 TOP-DOWN PARSING TECHNIQUIES

There are two types of parsing by which the top-down parsing can be performed.

1. Backtracking
2. Predictive parsing

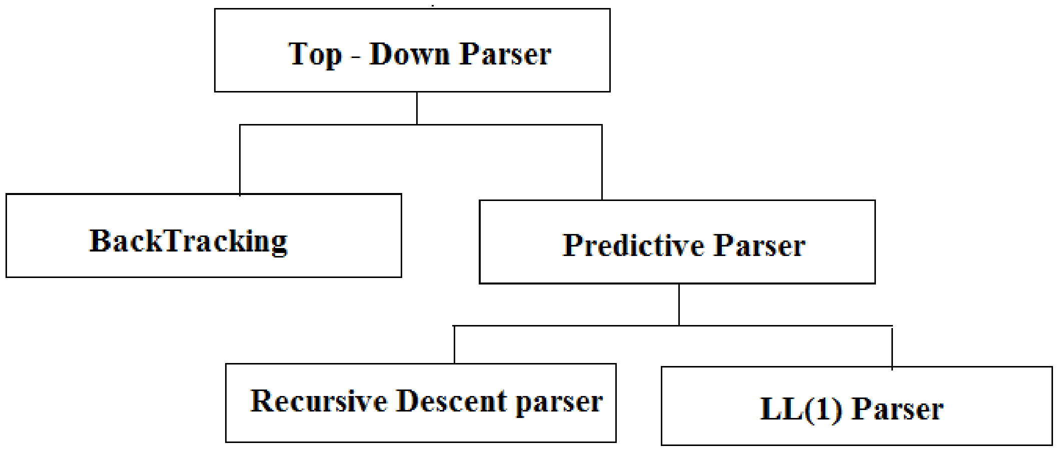


Fig: Types of Top-Down Parsing Techniques

Backtracking

Backtracking is a technique in which for expansion for non-terminal symbol we choose one alternative and if some mismatch occurs then we try another alternative if any.

Example

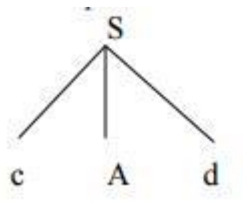
Consider the grammar G : S → cAd

A→ab|a and the input string w=cad.

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

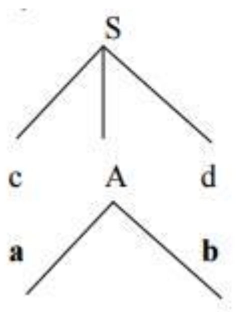
Step1:

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:

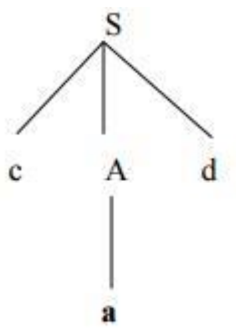
The leftmost leaf ‘c’ matches the first symbol of w, so advance the input pointer to the second symbol of w ‘a’ and consider the next leaf ‘A’. Expand A using the first alternative.



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.

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

Predictive parser

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

1. Recursive Descent parser
2. LL(1) parser

1.4 PROBLEMS IN TOP-DOWN PARSING

There are certain problems in top-down parsing. In order to implement the parsing we need to eliminate these problems. The problems are:

1. Ambiguity
2. Back Tracking
3. Left Recursion
4. Left Factoring

Ambiguity

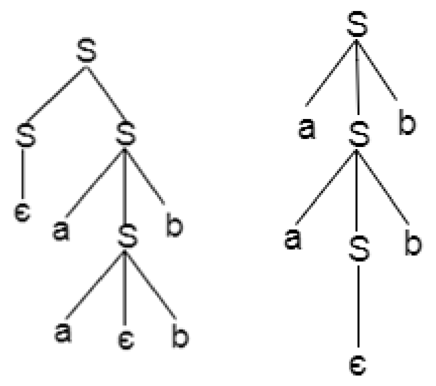
A grammar is said to be ambiguous if there exists more than one leftmost derivation or more than one rightmost derivative or more than one parse tree for the given input string. If the grammar is not ambiguous then it is called unambiguous.

Example:

S = aSb | SS

S = ∈

For the string aabb, the above grammar generates two parse trees:



If the grammar has ambiguity then it is not good for a compiler construction. No method can automatically detect and remove the ambiguity but you can remove ambiguity by re-writing the whole grammar without ambiguity.

Backtracking

Backtracking is a technique in which for expansion for non-terminal symbol we choose one alternative and if some mismatch occurs then we try another alternative if any.

Example

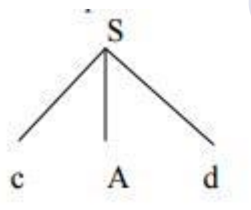
Consider the grammar G : S → cAd

A→ab|a and the input string w=cad.

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

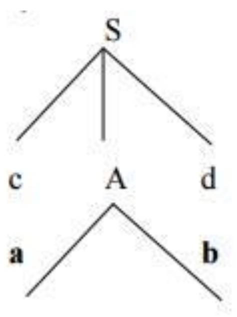
Step1:

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:

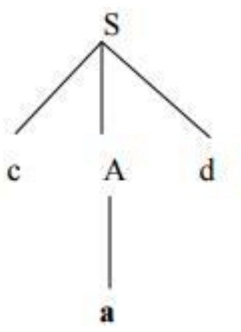
The leftmost leaf 'c' matches the first symbol of w, so advance the input pointer to the second symbol of w 'a' and consider the next leaf 'A'. Expand A using the first alternative.



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.

Left Recursion

There is a formal technique for eliminating left-recursion from productions

Step One: Direct-Recursion

For each rule which contains a left-recursive option, A --> A α | β

introduce a new nonterminal A' and rewrite the rule as

A --> β A'

A' --> ε | α A'

Thus the production:

E --> E + T | T

is left-recursive with "E" playing the role of "A","+ T" playing the role of α, and "T" playing the role of β A'. Introducing the new nonterminal E', the production can be replaced by:

E --> T E'

E' --> ε | + T E'

Of course, there may be more than one left-recursive part on the right-hand side. The general rule is to replace:

A --> A α_1 | α_2 | ...α_n | β_1 | β_2 | ... | β_m

by

A --> β_1A' | β_2A'| ...| β_mA'

A' --> ε | α_1A' | α_2A' | ...| α_nA'

Note that this may change the "structure". For the expression above, the original grammar is left-associative, while the non-left recursive one is now right-associative.

Step Two: Indirect-Recursion

Step one describes a rule to eliminate direct left recursion from a production. To eliminate left-recursion from an entire grammar may be more difficult because of indirect left-recursion. For example,

A --> B x y | x

B --> C D

C --> A | c

D --> d

is indirectly recursive because

A ==> B x y ==> C D x y ==> A D x y.

That is, A ==> ... ==> A ω where ω is D x y.

To eliminates left-recursion entirely. It contains a "call" to a procedure which eliminates direct left-recursion (as described in step one).

Left Factoring

Left Factoring is a grammar transformation technique. It consists in "factoring out" prefixes which are common to two or more productions.

Left factoring is removing the common left factor that appears in two productions of the same non-terminal. It is done to avoid back-tracing by the parser. Suppose the parser has a look-ahead ,consider this example-

A -> qB | qC

where A,B,C are non-terminals and q is a sentence. In this case, the parser will be confused as to which of the two productions to choose and it might have to back-trace. After left factoring, the grammar is converted to-

A -> qD

D -> B | C

In this case, a parser with a look-ahead will always choose the right production.

Left recursion is a case when the left-most non-terminal in a production of a non-terminal is the non-terminal itself(direct left recursion) or through some other non-terminal definitions, rewrites to the non-terminal again(indirect left recursion). Consider these examples -

(1) A -> Aq (direct)

(2) A -> Bq B -> Ar (indirect)

Left recursion has to be removed if the parser performs top-down parsing

2.5 RECURSIVE DESCENT PARSER

A recursive-descent parsing program consists of a set of recursive procedures, one for each non terminal. Each procedure is responsible for parsing the constructs defined by its non terminal, Execution begins with the procedure for the start symbol, which halts and announces success if its procedure body scans the entire input string.

If the given grammar is

E -> TE′

E′-> +TE′ | €

T -> FT′

T′-> *FT′ | €

F -> (E) | id

Reccursive procedures for the recursive descent parser for the given grammar are given below

```
procedure E( )
{
        if(Lookahead==' $ ')
        declare as Success ;
        else error;
        T( );
        E′( );
}
procedure T ( )
{
        F( );
        T′( );
}
Procedure E′( )
{
                if (Lookahead= = '+')
                {
                        match('+');
                         T ( );
                        E′( );


                }
        else if (Lookahead==NULL)
        {
                match('NULL') ;
        }
        else error;
}

procedure T′( )
{
        if (Lookahead == '*')
        {
                match('+');
                F ( );
                T′( );
        }
        else if(Lookahead == 'NULL')
        {
                match(NULL);
        }
        else error;
}

procedure F( )
{
        if (Lookahead== '(' )
        {
                match( '(' );
                E( );
        }
        if else(Lookahead==' )' )
        {
                match(' )' );
        }
        if else(Lookahead==' id ')
        {
                match('id');
        }
        else error;
}

Procedure match( token t)
{
        token t = next token;
        t= token;
}

Procedure Error ( )
{
        printf(" Error!");
}

Procedure NULL( )
{
        Printf(" Empty!");
}
```

2.6 PREDICTIVE LL(1) PARSER

It is possible to build a non- recursive 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.

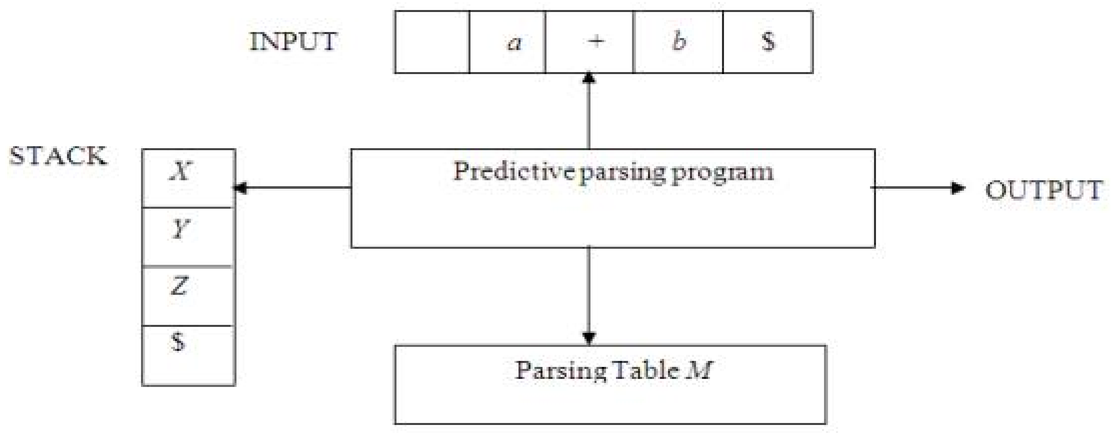


Fig. Predictive LL(1) parser

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 end marker 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.

Algorithm for predictive LL(1) 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 */
```

Predictive LL(1) 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 LL(1) 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.

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

Example:

Consider the following grammar:

E→E+T|T

T→T*F|F

F→(E)|id

Solution

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) = { $,) }

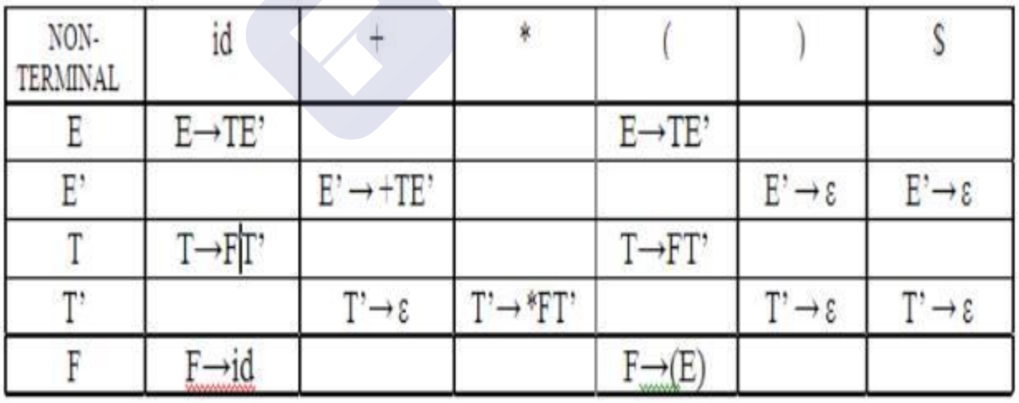
FOLLOW(E') = { $,) }

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

1.4 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.

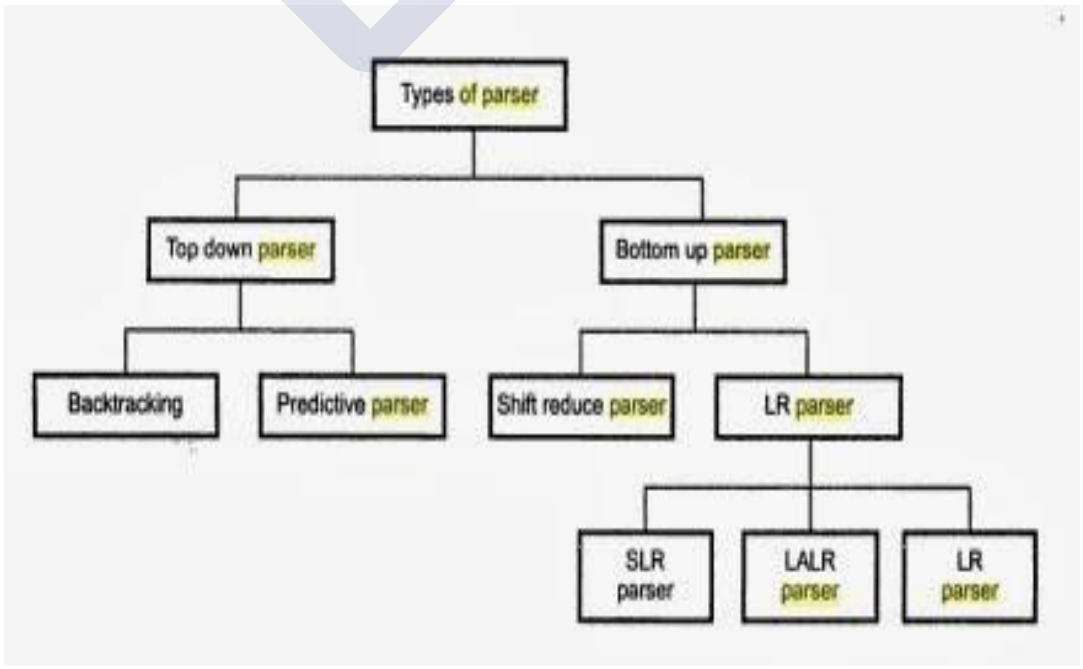


Fig: Types of Bottom up Parser

2.7 SHIFT REDUCING PARSER

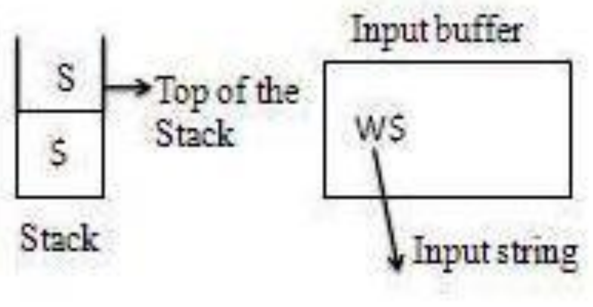
Shift-reduce parsing uses two unique steps for bottom-up parsing. These steps are known as shift-step and reduce-step.

- **Shift step**: The shift step refers to the advancement of the input pointer to the next input symbol, which is called the shifted symbol. This symbol is pushed onto the stack. The shifted symbol is treated as a single node of the parse tree.
- **Reduce step** : When the parser finds a complete grammar rule (RHS) and replaces it to (LHS), it is known as reduce-step. This occurs when the top of the stack contains a handle. To reduce, a POP function is performed on the stack which pops off the handle and replaces it with LHS non-terminal symbol.

Shift Reducing Parser to construct parse free from leaves to root. It works on the same principle of bottom up parser. A shift reduce parser requires following steps:

1. Input buffer storing the input-string
2. A Stack for storing and accessing the LHS and RHS of rules.

Initial configuration of Shift Reduce Parser



The parser performs following basic operations

Shift: Moving of Symbols from input buffer out to the stack, this action is called shift.

Reduce: If the handle appears on the top of the stack there reduction of if by appropriate rule is done that means RHS of rule is popped of and LHS is pushed in.

Accept: If the stack contain start symbol only and input buffer is empty at the same time then that action is called accept.

Example: Consider the grammar

$E \rightarrow E-E$

$E \rightarrow E*E$

$E \rightarrow id$

Perform shift-reduce parsing of the input string id-id * id

Stack	**Input buffer**	**Action**
$	id-id * id	Shift
$ id	-id * id	Reduced by E → id
$E	id * id	Shift
$E –	id * id	Shift
$E – id	* id	Reduced by E → id
$E – E	* id $	Shift
$E – E*id	id$	Shift
$E – E*id	$	Reduced by E → id
$E – E*E	$	Reduced by E → E * E
$E – E	$	Reduced by E → E – E
$E	$	Accept

2.8 OPERATOR PRECEDENCE PARSING

Operator precedence grammar is kinds of shift reduce parsing method. It is applied to a small class of operator grammars.

A grammar is said to be operator precedence grammar if it has two properties:

- No R.H.S. of any production has a∈.

- No two non-terminals are adjacent.

Operator precedence can only established between the terminals of the grammar. It ignores the non-terminal.

There are the three operator precedence relations:

a ⋗ b means that terminal "a" has the higher precedence than terminal "b".

a ⋖ b means that terminal "a" has the lower precedence than terminal "b".

a ≐ b means that the terminal "a" and "b" both have same precedence.

Precedence table:

	+	*	(	)	id	$
+	⋗	⋖	⋖	⋗	⋖	⋗
*	⋗	⋗	⋖	⋗	⋖	⋗
(	⋖	⋖	⋖	≐	⋖	X
)	⋗	⋗	X	⋗	X	⋗
id	⋗	⋗	X	⋗	X	⋗
$	⋖	⋖	⋖	X	⋖	X

2.9 LR PARSERS

LR parsing is one type of bottom up parsing. It is used to parse the large class of grammars.

In the LR parsing, "L" stands for left-to-right scanning of the input.
"R" stands for constructing a right most derivation in reverse.
"K" is the number of input symbols of the look ahead used to make number of parsing decision.

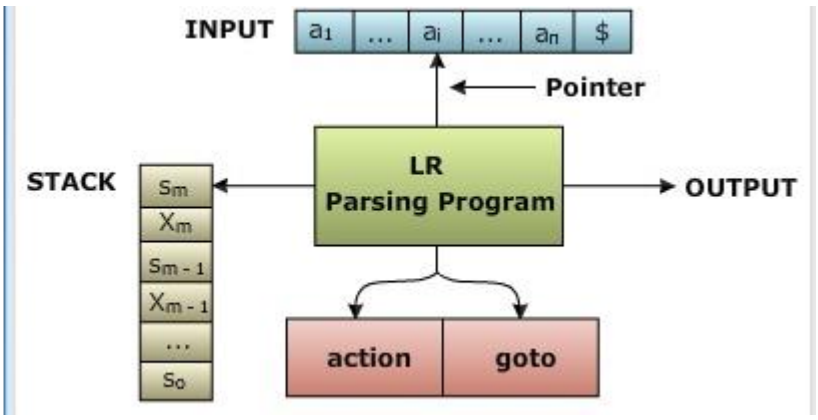


Figure: Structure of LR Parser

LR parsing is divided into four parts:

Input

Contains the string to be parsed and pointer

Parsing Table

Contains two parts ACTION and GOTO which is used by the parser program

1. ACTION Part

The ACTION part of the table is a two dimensional array indexed by state and the input symbol, i.e. **ACTION**[state][input], An action table entry can have one of following four kinds of values in it. They are:

1. Shift X, where X is a State number.
2. Reduce X, where X is a Production number.
3. Accept, signifying the completion of a successful parse.

2. GO TO Part

The GO TO part of the table is a two dimensional array indexed by state and a Non terminal, i.e. GOTO[state][NonTerminal]. A GO TO entry has a state number in the table.

Augment Grammar

The Augment Grammar G`, is G with a new starting symbol S` an additional production S` ->S. this helps the parser to identify when to stop the parsing and announce the acceptance of the input. The input string is accepted if and only if the parser is about to reduce by S`-> S. **NOTE:** Augment Grammar is simply adding one extra production by preserving the actual meaning of the given Grammar G.

Stack

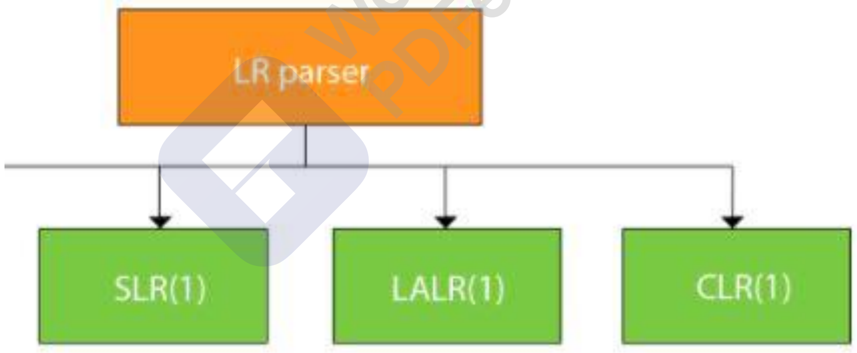
Contains string of the form $s_0X1s_1X2\ldots..Xms_m$ where s_m is on the top of the stack. Each Xi denotes a grammar symbol and s_i a state. Each state symbol summarizes the information contained in the stack below it

Parser Program

This determines the state on the s_m on the top of the stack and the current input symbol a to determine the next step

Types LR Parsers

SLR parsing, CLR parsing and LALR parsing.



2.9.1 SLR PARSING

SLR (1) refers to simple LR Parsing. It is same as LR(0) parsing. The only difference is in the parsing table. To construct SLR (1) parsing table, we use canonical collection of LR (0) item.

In the SLR (1) parsing, we place the reduce move only in the follow of left hand side.

Various steps involved in the SLR (1) Parsing:

- For the given input string write a context free grammar
- Check the ambiguity of the grammar

- Add Augment production in the given grammar
- Create Canonical collection of LR (0) items
- Draw a data flow diagram (DFA)
- Construct a SLR (1) parsing table

Example : To construct SLR (1) parser of the given grammar

E → E + T

E→T

T → T * F

T→ F

F →(E)

F → id

Solution:

The canonical collection of SLR(1) items are

Closure

I0:

E' → .E

E → .E + T

E → .T

T → .T * F

T → .F

F → .(E)

F → .id

I1: GOTO(I0,E)

E' → E.

E → E.+ T

I2: GOTO(I0,T)

E → T.

T → T .* F

I3: GOTO(I0,F)

T → F.

I4: GOTO(I0,( )

F → (.E)

E → . E + T

E → .T

T → .T * F

T → .F

F → .(E)

F → .id

I5: GOTO(I0,id)

F → id.

I6: GOTO(I1,E)

E → E + .T

T → .T * F

T → .F

F → .(E)

F → .id

I7: GOTO(I2,*)

T → T * .F

F → .(E)

F → .id

I8: GOTO(I4,E)

F → (E .)

E → E. + T

I9: GOTO(I6,T)

E → E + T.

T → T. * F

I10: GOTO(I7,F)

T → T * F.

I11: GOTO(I8,))

F → (E).

SLR Parsing Table

	ACTION						GOTO		
STATES	id	+	*	(	)	$	E	T	F
0	s5						1	2	3
1		s6		s4		ACC			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			S4			8		
5		r6	r6		r6	r6			
6	s5			s4				9	
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

2.9.2 CLR PARSING

CLR refers to canonical lookahead. 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 lookahead symbols. Various steps involved in the CLR (1) Parsing:

- For the given input string write a context free grammar
- Check the ambiguity of the grammar
- Add Augment production in the given grammar

- Create Canonical collection of LR (0) items
- Draw a data flow diagram (DFA)
- Construct a CLR (1) parsing table

LR (1) item

LR (1) item is a collection of LR (0) items and a look ahead symbol.

LR (1) item = LR (0) item + look ahead

The look ahead is used to determine that where we place the final item.

The look ahead always add $ symbol for the argument production.

Example: To construct CLR Parser of the given grammar

S → AA
A → aA
A → b

Add Augment Production, insert '•' symbol at the first position for every production in G and also add the lookahead.

S` → •S, $
S → •AA, $
A → •aA, a/b
A → •b, a/b

I0 State:

Add Augment production to the I0 State and Compute the Closure

I0 = Closure (S` → •S)

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

I0 = S` → •S, $

S → •AA, $

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

I0= S` → •S, $

S → •AA, $

A → •aA, a/b

A → •b, a/b

I1= Go to (I0, S) = closure (S` → S•, $) = S` → S•, $

I2= Go to (I0, A) = closure (S → A•A, $)

Add all productions starting with A in I2 State because "." is followed by the non-terminal. So, the I2 State becomes

I2= S → A•A, $

A → •aA, $

A → •b, $

I3= Go to (I0, a) = Closure (A → a•A, a/b)

Add all productions starting with A in I3 State because "." is followed by the non-terminal. So, the I3 State becomes

I3= A → a•A, a/b

A → •aA, a/b

A → •b, a/b

Go to (I3, a) = Closure (A → a•A, a/b) = (same as I3)

Go to (I3, b) = Closure (A → b•, a/b) = (same as I4)

I4= Go to (I0, b) = closure (A → b•, a/b) = A → b•, a/b

I5= Go to (I2, A) = Closure (S → AA•, $) =S → AA•, $

I6= Go to (I2, a) = Closure (A → a•A, $)

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

I6 = A → a•A, $
A → •aA, $
A → •b, $

Go to (I6, a) = Closure (A → a•A, $) = (same as I6)

Go to (I6, b) = Closure (A → b•, $) = (same as I7)

I7= Go to (I2, b) = Closure (A → b•, $) = A → b•, $

I8= Go to (I3, A) = Closure (A → aA•, a/b) = A → aA•, a/b

I9= Go to (I6, A) = Closure (A → aA•, $) = A → aA•, $

Drawing DFA:

CLR Parsing table:

States	a	b	$	S	A
I_0	S_3	S_4			2
I_1			Accept		
I_2	S_6	S_7			5
I_3	S_3	S_4			8
I_4	R_3	R_3			
I_5			R_1		
I_6	S_6	S_7			9
I_7			R_3		
I_8	R_2	R_2			
I_9			R_2		

2.9.3 LALR PARSING

LALR refers to the lookahead LR. To construct the LALR (1) parsing table, we use the canonical collection of LR (1) items.

In the LALR (1) parsing, the LR (1) items which have same productions but different look ahead are combined to form a single set of items

LALR (1) parsing is same as the CLR (1) parsing, only difference in the parsing table.

Example : To construct LALR Parser of the given Grammar

$S \to AA$
$A \to aA$
$A \to b$

Add Augment Production, insert '•' symbol at the first position for every production in G and also add the look ahead.

$S` \to •S, \$$
$S \to •AA, \$$
$A \to •aA, a/b$
$A \to •b, a/b$

I0 State:

Add Augment production to the I0 State and Compute the ClosureL

I0 = Closure ($S` \to •S$)

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

I0 = $S` \to •S, \$$
$S \to •AA, \$$

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

I0= $S` \to •S, \$$
$S \to •AA, \$$
$A \to •aA, a/b$
$A \to •b, a/b$

I1= Go to (I0, S) = closure ($S` \to S•, \$$) = $S` \to S•, \$$

I2= Go to (I0, A) = closure ($S \to A•A, \$$)

Add all productions starting with A in I2 State because "•" is followed by the non-terminal.

So, the I2 State becomes

I2= S → A•A, $

A → •aA, $

A → •b, $

I3= Go to (I0, a) = Closure (A → a•A, a/b)

Add all productions starting with A in I3 State because "•" is followed by the non-terminal.

So, the I3 State becomes

I3= A → a•A, a/b

A → •aA, a/b

A → •b, a/b

Go to (I3, a) = Closure (A → a•A, a/b) = (same as I3)

Go to (I3, b) = Closure (A → b•, a/b) = (same as I4)

I4= Go to (I0, b) = closure (A → b•, a/b) = A → b•, a/b

I5= Go to (I2, A) = Closure (S → AA•, $) =S → AA•, $

I6= Go to (I2, a) = Closure (A → a•A, $)

Add all productions starting with A in I6 State because "•" is followed by the non-terminal.

So, the I6 State becomes

I6 = A → a•A, $

A → •aA, $

A → •b, $

Go to (I6, a) = Closure (A → a•A, $) = (same as I6)

Go to (I6, b) = Closure (A → b•, $) = (same as I7)

I7= Go to (I2, b) = Closure (A → b•, \$) = A → b•, \$

I8= Go to (I3, A) = Closure (A → aA•, a/b) = A → aA•, a/b

I9= Go to (I6, A) = Closure (A → aA•, \$) A → aA•, \$

If we analyze then LR (0) items of I3 and I6 are same but they differ only in their lookahead.

I3 = { A → a•A, a/b
A → •aA, a/b
A → •b, a/b
}

I6= { A → a•A, \$
A → •aA, \$
A → •b, \$
}

Clearly I3 and I6 are same in their LR (0) items but differ in their lookahead, so we can combine them and called as I36.

I36 = { A → a•A, a/b/\$
A → •aA, a/b/\$
A → •b, a/b/\$
}

The I4 and I7 are same but they differ only in their look ahead, so we can combine them and called as I47.

I47 = {A → b•, a/b/\$}

The I8 and I9 are same but they differ only in their look ahead, so we can combine them and called as I89.

I89 = {A → aA•, a/b/\$}

Drawing DFA:

LALR Parsing table:

States	a	b	$	S	A
I_0	S_{36}	S_{47}		12	
I_1		accept			
I_2	S_{36}	S_{47}			5
I_{36}	$S_{36}S_{47}$				89
I_{47}	R_3R_3	R_3			
I_5			R_1		
I_{89}	R_2	R_2	R_2		

2.10 AUTOMATIC PARSER GENERATOR-YACC

YACC stands for Yet Another Compiler Compiler. This program is available in UNIX OS . The construction of LR parser requires lot of work for parsing the input string. Hence, the process must involve automation to achieve efficiency in parsing an input . Basically YACC is a LALR parser generator that reports conflicts or uncertainties (if at all present) in the form of error messages. The typical YACC translator can be represented as shown in the image

YACC Specification

The YACC specification file consists of three parts.

Declaration section: In this section, ordinary C declarations are inserted and grammar tokens are declared. The tokens should be declared between %{ and %}

Translation rule section

It includes the production rules of context free grammar with corresponding actions

Rule-1 action-1
Rule-2 action-2
:
:
Rule n action n

If there is more than one alternative to a single rule then those alternatives are separated by '|' (pipe) character. The actions are typical C statements. If CFG is

```
LHS:    alternative 1 | alternative 2 | …… alternative n
Then
LHS:    alternative 1      {action 1}
        | alternative 2   {action 1}
        :
        :
        alternative n      {action n}
```

C functions Section: this consists of one main function in which the routine yyparse() is called. And it also contains required C functions

The specification file comprising these sections can be written as:

```
%{
 / * declaration section */
%}
/* Translation rule section */
%%
/* Required C functions */
```

Example:

YACC Specification of a simple desk calculator:

```
%{
#include <ctype.h>
%}
%token DIGIT
%%
line: expr '\n'     { printf("%d\n", $1); }
            ;
expr : expr '+' term        { $$ = $1 + $3; }
            | term
            ;
term : term '*' factor     { $$ = $1 * $3; }
            | factor
            ;
factor : '(' expr ')'           { $$ = $2; }
            | DIGIT
            ;
%%
yylex()  {
                        int c;
                        c = getchar();
                        if(isdigit(c)
                        {
                                    yylval = c-'0';
                                    return DIGIT;
                        }
return c;
}
```

2.11 AMBIGUOUS GRAMMARS

YACC declarations resolve shift/reduce and reduce/reduce conflicts using operator precedence and operator associativity information

YACC has default methods for resolving conflicts. However, it is better to find out what conflicts arose and how they were resolved using the '-v' option

The declarations provide a way to override YACCs defaults

Productions have the precedence of their rightmost terminal, unless otherwise specified by "%prec" element

The declaration keywords %left, %right and %nonassoc inform YACC that the following tokens are to be treated as left-associative (as binary + - * & / commonly are), right-associative (as exp often is), or non-associative (as binay < & > often are)

The order of declarations informs YACC that the tokens should be accorded increasing precedence

%left '+' '-'	Effect is that * has higher precedence
%left '*' '|'	Than +, so x+y*z is grouped like x+(y*z)

YACC Specification for a more advanced desk calculator:

```
%{
#include <ctype.h>
#include <stdiio.h>
#define YYSTYPE double              /* double type for YACC stack */
%}
%token NUMBER
%left '+' '-'
%left '*' '/'
%right UMINUS
%%
lines  :  lines expr '\n' { printf("%g\n", $2); }
            | lines '\n'
            | /* empty */
            ;
expr : expr '+' expr { $$ = $1 + $3; }
            | expr '-' expr { $$ = $1 - $3; }
            | expr '*' expr {$$ = $1 * $3; }
            | expr '/' expr {$$ = $1 / $3; }
            | '(' expr ')' {$$ = $2;}
            | '-' expr %prec UMINUS { $$ = -$2; }
            | NUMBER
            ;
```

```
%
yylex() {
            int c;
            while (( c = getchar() ) == ‘ ‘ );
            if ((c == ‘.’) || (isdigit(c) ) {
                        unget(c, stdin);
                        scanf(“%1f”, &uulval);
                        return Number;
            }
            return c;
```