RGPVNOTES.IN

Program : **B.Tech**

Subject Name: **Compiler Design**

Subject Code: **CS-603**

Semester: **6th**

Department of Computer Science and Engineering
Subject Notes
CS 603(C) Compiler Design

UNIT- II:
Syntax analysis: CFGs, Top down parsing, Brute force approach, recursive descent parsing, transformation on the grammars, predictive parsing, bottom up parsing, operator precedence parsing, LR parsers (SLR, LALR, LR), Parser generation. Syntax directed definitions: Construction of Syntax trees, Bottom up evaluation of S-attributed definition, L-attribute definition, Top down translation, Bottom Up evaluation of inherited attributes Recursive Evaluation, Analysis of Syntax directed definition.

---

1 Introduction of Syntax Analysis

The second stage of translation is called Syntax analysis or parsing. In this phase expressions, statements, declarations etc… are identified by using the results of lexical analysis. Syntax analysis is aided by using techniques based on formal grammar of the programming language.
Where lexical analysis splits the input into tokens, the purpose of syntax analysis (also known as parsing) is to recombine these tokens. Not back into a list of characters, but into something that reflects the structure of the text. This "something" is typically a data structure called the syntax tree of the text. As the name indicates, this is a tree structure. The leaves of this tree are the tokens found by the lexical analysis, and if the leaves are read from left to right, the sequence is the same as in the input text. Hence, what is important in the syntax tree is how these leaves are combined to form the structure of the tree and how the interior nodes of the tree are labeled. In addition to finding the structure of the input text, the syntax analysis must also reject invalid texts by reporting syntax errors.
As syntax analysis is less local in nature than lexical analysis, more advanced methods are required. We, however, use the same basic strategy: A notation suitable for human understanding is transformed into a machine-like low-level notation suitable for efficient execution. This process is called parser generation.
A syntax analyzer or parser takes the input from a lexical analyzer in the form of token streams. The parser analyzes the source code (token stream) against the production rules to detect any errors in the code. The output of this phase is a **parse tree**.
This way, the parser accomplishes two tasks, i.e., parsing the code, looking for errors and generating a parse tree as the output of the phase.

1.1 Context Free Grammar

Definition − A context-free grammar (CFG) consisting of a finite set of grammar rules is a quadruple **(N, T, P,S),** where
N is a set of non-terminal symbols.
T is a set of terminals where **N ∩ T = NULL.**
P is a set of rules, **P: N → (N ∪ T)***, i.e., the left-hand side of the production rule **P** does have any right context or left context.
S is the start symbol.
Example
The grammar ({A}, {a, b, c}, P, A), P : A → aA, A → abc.
The grammar ({S, a, b}, {a, b}, P, S), P: S → aSa, S → bSb, S → ε

Generation of Derivation Tree
A derivation tree or parse tree is an ordered rooted tree that graphically represents the semantic information a string derived from a context-free grammar.
Representation Technique
Root vertex − Must be labeled by the start symbol.
Vertex − Labeled by a non-terminal symbol.

Leaves − Labeled by a terminal symbol or ε.
If $S \rightarrow x_1x_2 \ldots\ldots x_n$ is a production rule in a CFG, then the parse tree / derivation tree will be as follows –

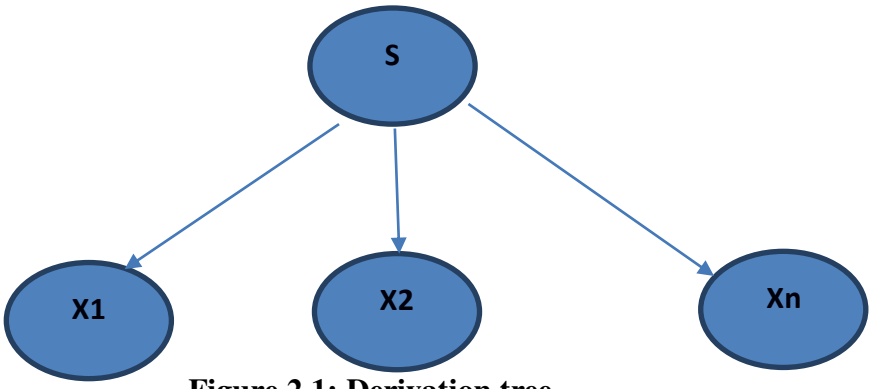

Figure 2.1: Derivation tree

There are two different approaches to draw a derivation tree −
Top-down Approach −
Starts with the starting symbol **S**
Goes down to tree leaves using productions

Bottom-up Approach −
Starts from tree leaves
Proceeds upward to the root which is the starting symbol **S**
Leftmost and Rightmost Derivation of a String

Leftmost derivation − A leftmost derivation is obtained by applying production to the leftmost variable in each step.
Rightmost derivation − A rightmost derivation is obtained by applying production to the rightmost variable in each step.
Example
Let any set of production rules in a CFG be
X → X+X | X*X |X| a
over an alphabet {a}.
The leftmost derivation for the string **"a+a*a"** may be −
X → X+X → a+X → a + X*X → a+a*X → a+a*a
The stepwise derivation of the above string is shown as below −
The rightmost derivation for the above string **"a+a*a"** may be −
X → X*X → X*a → X+X*a → X+a*a → a+a*a
The stepwise derivation of the above string is shown as below −

2. Parsing

Syntax analyzers follow production rules defined by means of context-free grammar. The way the production rules are implemented (derivation) divides parsing into two types: top-down parsing and bottom-up parsing.

2.1 Top-Down Parsing

A program that performs syntax analysis is called a parser. A syntax analyzer takes tokens as input and output error message if the program syntax is wrong. The parser uses symbol-look-ahead and an approach called top-down parsing without backtracking. Top-down parsers check to see if a string can be generated by a grammar by creating a parse tree starting from the initial symbol and working down. Bottom-up parsers, however, check to see a string can be generated from a grammar by creating a parse tree from the

leaves, and working up. Early parser generators such as YACC creates bottom-up parsers whereas many of Java parser
generators such as JavaCC create top-down parsers.

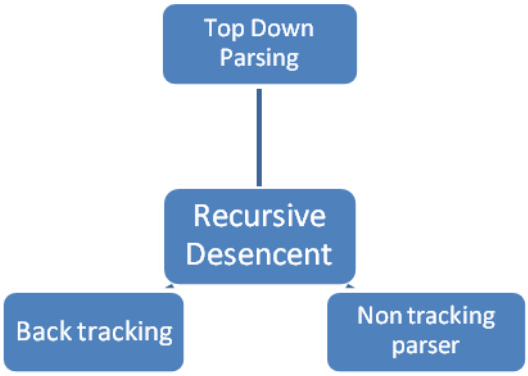

Figure 2.2: Top down parsing

2.2 Brute-Force Approach

A top-down parse moves from the goal symbol to a string of terminal symbols. In the terminology of trees, this is moving from the root of the tree to a set of the leaves in the syntax tree for a program. In using full backup we are willing to attempt to create a syntax tree by following branches until the correct set of terminals is reached. In the worst possible case, that of trying to parse a string which is not in the language, all possible combinations are attempted before the failure to parse is recognized.

Top-down parsing with full backup is a " brute-force" method of parsing. In general terms, this method operates as follows:

1. Given a particular non-terminal that is to be expanded, the first production for this non-terminal is applied.

2. Then, within this newly expanded string, the next (leftmost) non-terminal is selected for expansion and its first production is applied.

3. This process (step 2) of applying productions is repeated for all subsequent non-terminals that are selected until such time as the process cannot or should not be continued. This termination (if it ever occurs) may be due to two causes. First, no more non-terminals may be present, in which case the string has been successfully parsed. Second, it may result from an incorrect expansion which would be indicated by the production of a substring of terminals which does not match the appropriate segment of the source string. In the case of such an incorrect expansion, the process is "backed up" by undoing the most recently applied production. Instead of using the particular expansion that caused the error, the next production of this non-terminal is used as the next expansion, and then the process of production application continues as before.

If, on the other hand, no further productions are available to replace the production that caused the error, this error-causing expansion is replaced by the non-terminal itself, and the process is backed up again to undo the next most recently applied production. This backing up continues either until we are able to resume normal application of productions to selected non-terminals or until we have backed up to the goal symbol and there are no further productions to be tried. In the latter case, the given string must be unappeasable because it is not part of the language determined by this particular grammar.

As an example of this brute-force parsing technique, let us consider the simple grammar

S->aAd/aB	**A->b/c**	**B->ccd/ddc**

Where S is the goal or start symbol. Figure 6-1 illustrates the working of this brute-force parsing technique by showing the sequence of syntax trees generated during the parse of the string 'accd'.

2.3 Recursive Decent Parsing

Typically, top-down parsers are implemented as a set of recursive functions that descent through a parse tree for a string. This approach is known as recursive descent parsing, also known as LL(k) parsing where the first L stands for left-to-right, the second L stands for leftmost-derivation, and k indicates k-symbol look ahead. Therefore, a parser using the single symbol look-ahead method and top-down parsing without backtracking is called LL(1) parser. In the following sections, we will also use an extended BNF notation in which some regulation expression operators are to be incorporated.

A syntax expression defines sentences of the form,or. A syntax of the form defines sentences that consist of a sentence of the form followed by a sentence of the form followed by a sentence of the form. A syntax of the form defines zero or one occurrence of the form. A syntax of the form defines zero or more occurrences of the form.

A usual implementation of an LL(1) parser is.

- initialize its data structures,
- get the lookahead token by calling scanner routines, and
- call the routine that implements the start symbol.

Here is an example.

```
proc syntax Analysis()
begin
initialize(); // initialize global data and structures
nextToken(); // get the lookahead token
program(); // parser routine that implements the start symbol
end;
```


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.

2.4 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

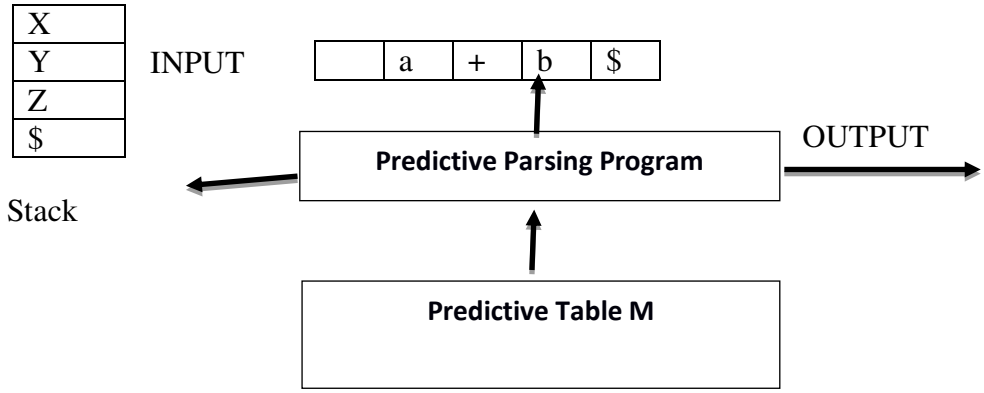

Figure 2.3: Predictive parsing

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

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, thecurrent input symbol. These two symbols determine the parser action. There are three possibilities:

- If X = a = $, the parser halts and announces successful completion of parsing.
- If X = a ≠ $, the parser pops X off the stack and advances the input pointer to the next input symbol.
- 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.

Predictive parsing table construction:

The construction of a predictive parser is aided by two functions associated with a grammarG :
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 forsome 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 infollow (B).
3. If there is a production A → αB, or a production A → αBβ where FIRST (β) contains ε, theneverything in FOLLOW (A) is in FOLLOW (B).

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) = {+, * , $,) }

Non terminal	Id	+	*	(	)	$
E	E → TE'			E → TE'		
E'		E' → +TE'			E' → ε	E' → ε
T	T → FT'			T → FT'		
T'		T→ ε			T→ ε	T→ ε
F	F → id			F → (E)		

Table 2.1: Predictive parsing table

3. Bottom-Up Parsing

Constructing a parse tree for an input string beginning at the leaves and going towards theroot is called bottom-up parsing.

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

3.1 Shift-Reduce Parsing

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.

Example:
Consider the grammar:
S → aABe
A → Abc | b
B → d
The sentence to be recognized is abbcde

REDUCTION (LEFT MOST)	RIGHTMOST DERIVATION
abcde (A->b)	S->aABe
aAcde (A->Avc)	->aAde
aAde(B->d)	->aAbcde
aABe(S->aABe)	->abbcde

Table 2.2: Shift Reduce Parser

3.2 Operator Precedence Parsing

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. These parser rely 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.
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

	Id	**+**	*****	**$**
Id		.>	.>	.>
+	<.	.>	<.	.>
*****	<.	.>	.>	.>
$	<.	<.	<.	.>

Table 2.3: Operator precedence parsing

3.3 LR Parsing Introduction

The "L" is for left-to-right scanning of the input and the "R" is for constructing a rightmost derivation in reverse.

Why LR Parsing:

LR parsers can be constructed to recognize virtually all programming-language constructs for which context-free grammars can be written.

The LR parsing method is the most general non-backtracking shift-reduce parsing method known, yet it can be implemented as efficiently as other shift-reduce methods.

The class of grammars that can be parsed using LR methods is a proper subset of the class of grammars that can be parsed with predictive parsers.

An LR parser can detect a syntactic error as soon as it is possible to do so on a left-to-right scan of the input.

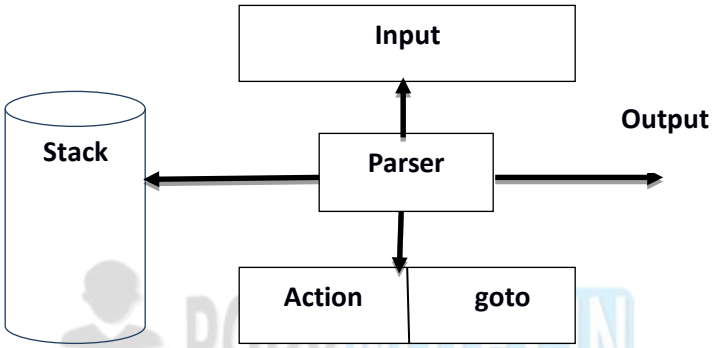

Figure 2.4: LR Parser

The disadvantage is that it takes too much work to construct an LR parser by hand for a typical programming-language grammar. But there are lots of LR parser generators available to make this task easy.

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)

3.4 SLR (1) – Simple LR Parser:

Shift-reduce parsing attempts to construct a parse tree for an input string beginning at the leaves and working up towards the root. In other words, it is a process of “reducing” (opposite of deriving a symbol using a production rule) a string w to the start symbol of a grammar. At every (reduction) step, a particular substring matching the RHS of a production rule is replaced by the symbol on the LHS of the production.

A general form of shift-reduce parsing is LR (scanning from Left to right and using Right-most derivation in reverse) parsing, which is used in a number of automatic parser generators like Yacc, Bison, etc.

A convenient way to implement a shift-reduce parser is to use a stack to hold grammar symbols and an input buffer to hold the string w to be parsed. The symbol $ is used to mark the bottom of the stack and also the right-end of the input.

Notation ally, the top of the stack is identified through a separator symbol |, and the input string to be parsed appears on the right side of |. The stack content appears on the left of |.

For example, an intermediate stage of parsing can be shown as follows:

$id1 | + id2 * id3$ …. (1)

Here "$id1" is in the stack, while the input yet to be seen is "+ id2 * id3$*

In shift-reduce parser, there are two fundamental operations: shift and reduce.

Shift operation: The next input symbol is shifted onto the top of the stack.

After shifting + into the stack, the above state captured in (1) would change into:

$id1 + | id2 * id3$

Reduce operation: Replaces a set of grammar symbols on the top of the stack with the LHS of a production rule.

After reducing id1 using E → id, the state (1) would change into:

$E | + id2 * id3$

In every example, we introduce a new start symbol (S'), and define a new production from this new start symbol to the original start symbol of the grammar.

Consider the following grammar (putting an explicit end-marker $ at the end of the first production:

(1) S' → S$

(2) S → Sa

(3) S → b

For this example, the NFA for the stack can be shown as follows:

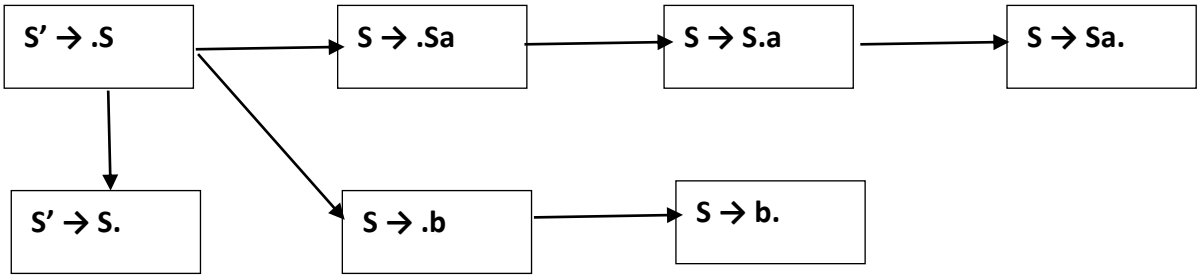

Figure 2.5: Shift Operation

After doing ε-closure, the resulting DFA is as follows:

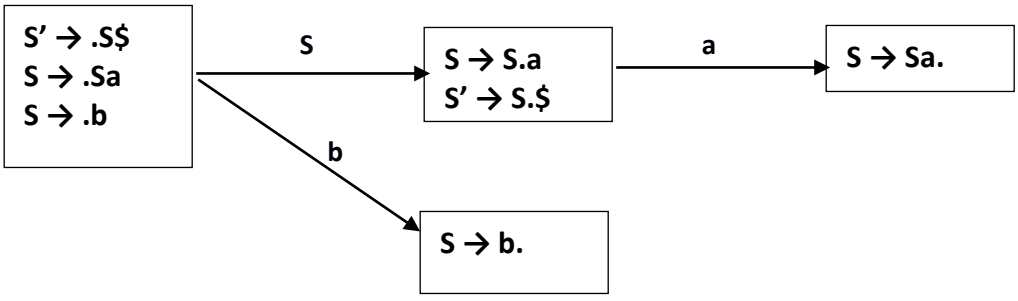

Figure 2.6: Reduce Operation

The states of DFA are also called "Canonical Collection of Items". Using the above notation, the ACTION-GOTO table can be shown as follows:

State	A	B	$	s
1		S3		
2	S4,r1	R1	R1	
3	R3	R3	R3	
4	R2	R2	R2	

Table 2.4: Simple LR Parser

3.5. Canonical LR Parsing

- Carry extra information in the state so that wrong reductions by A ⟶α will be ruled out.
- Redefine LR items to include a terminal symbol as a second component (look ahead symbol).
- The general form of the item becomes [A ⟶α . □, a] which is called LR(1) item.
- Item [A ⟶α ., a] calls for reduction only if next input is a. The set of symbols

Canonical LR parsers solve this problem by storing extra information in the state itself. The problem we have with SLR parsers is because it does reduction even for those symbols of follow (A) for which it is invalid. So LR items are redefined to store 1 terminal (look ahead symbol) along with state and thus, the items now are LR(1) items.

An LR(1) item has the form : [A⟶ a . □, a] and reduction is done using this rule only if input is 'a'. Clearly the symbols a's form a subset of follow (A).

Closure (I)

repeat
for each item [A ⟶α .B □, a] in I
for each production B ⟶γ in G'
and for each terminal b in First(□a)
add item [B ⟶. γ , b] to I
until no more additions to I

To find closure for Canonical LR parsers:

Repeat
for each item [A ⟶α .B □, a] in I
for each production B ⟶γ in G'
and for each terminal b in First(□a)
add item [B ⟶. γ , b] to I
until no more items can be added to I

Example

Consider the following grammar
S' ⟶S
S ⟶CC
C ⟶cC | d
Compute closure (I) where I={[S' ⟶.S, $]}

S' ⟶.S, $
S ⟶.CC, $
C ⟶.cC, c
C ⟶.cC, d
C ⟶.d, c
C ⟶.d, d

For the given grammar:
S ' → S
S → CC
C → cC | d
I : closure([S ' → S, $])

S ' → .S	$	as first(e $) = {$}
S → .CC	$	as first(C$) = first(C) = {c, d}
C → .cC	c	as first(Cc) = first(C) = {c, d}
C → .cC	d	as first(Cd) = first(C) = {c, d}
C → .d	c	as first(e c) = {c}
C → .d	d	as first(e d) = {d}

Table 2.5: Canonical LR Parsing

Example
Construct sets of LR(1) items for the grammar on previous slide

I_0 :	S ' → .S,	$
	S → .CC,	$
	C → .cC,	c /d
	C → .d,	c /d
I_1 :	goto(I_0 ,S)	
	S ' → S.,	$
I_2 :	goto(I 0 ,C)	
	S → C.C,	$
	C → .cC,	$
	C → .d,	$
I_3 :	goto(I 0 ,c)	
	C → c.C,	c/d
	C → .cC,	c/d
	C → .d,	c/d
I_4 :	goto(I 0 ,d)	
	C → d.,	c/d
I_5 :	goto(I 2 ,C)	
	S → CC.,	$
I_6 :	goto(I 2 ,c)	
	C → c.C,	$
	C → .cC,	$
	C → .d,	$
I_7 :	goto(I 2 ,d)	
	C → d.,	$
I_8 :	goto(I 3 ,C)	

	C ⟶ cC.,	c/d
I_9 :	goto(I 6 ,C)	
	C ⟶ cC.,	$

To construct sets of LR (1) items for the grammar given in previous slide we will begin by computing closure of {[S ´ ⟶ .S, $]}.
To compute closure we use the function given previously.
In this case α = ε , B = S, ß =ε and a=$. So add item [S ⟶ .CC, $].
Now first(C$) contains c and d so we add following items
We have A=S, α = ε , B = C, ß=C and a=$
Now first(C$) = first(C) contains c and d
So we add the items [C ⟶ .cC, c], [C ⟶ .cC, d], [C ⟶ .dC, c], [C ⟶ .dC, d].
Similarly we use this function and construct all sets of LR (1) items.

Construction of Canonical LR parse table
Construct C={I_0 , . , I_n } the sets of LR(1) items.
If [A ⟶ α .a □, b] is in I_i and goto (I_i , a)=I_j then action[i,a]=shift j
If [A ⟶ α ., a] is in I_i then action[i,a] reduce A ⟶ α
If [S ' ⟶ S., $] is in I_i then action [i,$] = accept
If goto (I i , A) = I_j then goto[i,A] = j for all non
We are representing shift j as sj and reduction by rule number j as rj. Note that entries corresponding to [state, terminal] are related to action table and [state, non-terminal] related to goto table. We have [1,$] as accept because [S ´ ⟶ S., $] ε I_1 .

Parse table

state	id	+	*	(	)	$		E	T	F
0	s5			s4				1	2	3
1		s6				acc				
2		r2	s7		r2	r2				
3		r4	r4		r4	r4				
4	s5			s4				8	2	3
5		r6	r6		r6	r6				
6	s5			s4					9	3
7	s5			s4						10
8		s6			s11					
9		r1	s7		r1	r1				

10		**r3**	**r3**		**r3**	**r3**				
11		**r5**	**r5**		**r5**	**r5**				

Table 2.6: Parser Table

We are representing shift j as sj and reduction by rule number j as rj. Note that entries corresponding to [state, terminal] are related to action table and [state, non-terminal] related to goto table. We have [1,$] as accept because [S ´ ⟶S., $] ε I_1

LALR Parse table

Look Ahead LR parsers

Consider a pair of similar looking states (same kernel and different lookaheads) in the set of LR(1) items

I_4 : C ⟶d. , c/d I_7 : C ⟶d., $

Replace I_4 and I_7 by a new state I_{47} consisting of (C ⟶d., c/d/$)

Similarly I_3& I_6 and I_8& I_9 form pairs

Merge LR(1) items having the same core

We will combine Ii and Ij to construct new Iij if Ii and Ij have the same core and the difference is only in look ahead symbols. After merging the sets of LR(1) items for previous example will be as follows:

I_0 : S' ⟶S $

S ⟶.CC $

C ⟶.cC c/d

C ⟶.d c/d

I_1 : goto(I 0 ,S)

S' ⟶S. $

I_2 : goto(I 0 ,C)

S ⟶C.C $

C⟶ .cC $

C ⟶.d $

I_{36} : goto(I 2 ,c)

C ⟶c.C c/d/$

C ⟶.cC c/d/$

C ⟶.d c/d/$

I_4 : goto(I 0 ,d)

C ⟶d. c/d

I_5 : goto(I 2 ,C)

S ⟶CC. $

I_7 : goto(I 2 ,d)

C ⟶d. $

I_{89} : goto(I 36 ,C)

C ⟶cC. c/d/$

Construct LALR parse table

Construct C={I_0 , .. ,I_n } set of LR(1) items

For each core present in LR(1) items find all sets having the same core and replace these sets by their union

Let C' = {J_0 , .. .,J_m } be the resulting set of items

Construct action table as was done earlier

Let J = I_1 U I_2 .. .U I_k

since I_1 , I_2 .. ., I_k have same core, goto(J,X) will have he same core

Let K=goto(I_1 ,X) U goto(I_2 ,X) .. goto(I_k ,X) the goto(J,X)=K

The construction rules for LALR parse table are similar to construction of LR(1) parse table.

LALR parse table

state	id	+	*	(	)	$		E	T	F
0	s5			s4				1	2	3
1		s6				acc				
2		r2	s7		r2	r2				
3,6		r4	r4		r4	r4			9	3
4,7	s5,7			s4,7				8	2	3,10
5,8	s5	r6	r6	s4,8	r6	r6				
9,10		r1	s7		r1	r1				
10		r3	r3		r3	r3				
11		r5	r5		r5	r5				

Table 2.7: LALR parse table.

The construction rules for LALR parse table are similar to construction of LR(1) parse table.

Notes on LALR parse table

Modified parser behaves as original except that it will reduce C —→d on inputs like ccd. The error will eventually be caught before any more symbols are shifted.

In general core is a set of LR(0) items and LR(1) grammar may produce more than one set of items with the same core.

4 Parser Generators

Some common parser generators

YACC: **Y**et **A**nother **C**ompiler **C**ompiler

Bison: GNU Software

ANTLR: **AN** other **T**ool for **L**anguage **R**ecognition

Yacc/Bison source program specification (accept LALR grammars)

declaration
%%
translation rules

%%
supporting C routines

Yacc and Lex schema

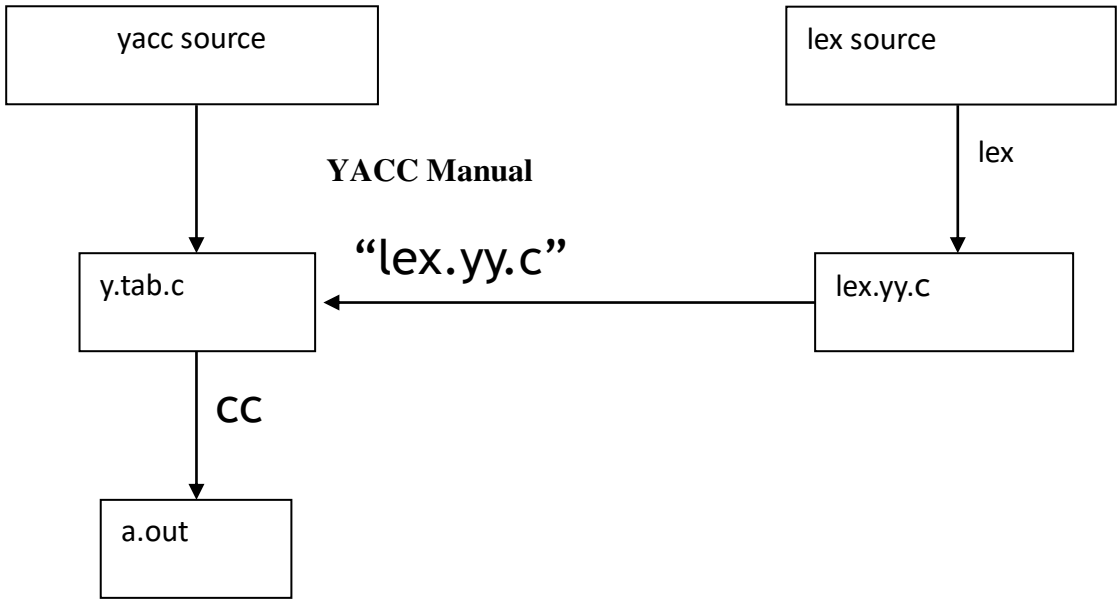

Figure 2.7: YACC

5. Syntax Directed Definition

Specifies the values of attributes by associating semantic rules with the grammar productions.

It is a context free grammar with attributes and rules together which are associated with grammar symbols and productions respectively.

The process of syntax directed translation is two-fold:

• Construction of syntax tree and

• Computing values of attributes at each node by visiting the nodes of syntax tree.

Semantic actions

Semantic actions are fragments of code which are embedded within production bodies by syntax directed translation.

They are usually enclosed within curly braces ({ }).

It can occur anywhere in a production but usually at the end of production.

E---> E_1 + *T {print* '+'}

Types of translation

• L-attributed translation

It performs translation during parsing itself.

No need of explicit tree construction.

L represents 'left to right'.

• S-attributed translation

It is performed in connection with bottom up parsing.

'S' represents synthesized.

Types of attributes

• Inherited attributes

It is defined by the semantic rule associated with the production at the parent of node.

Attributes values are confined to the parent of node, its siblings and by itself.

The non-terminal concerned must be in the body of the production.

• Synthesized attributes

It is defined by the semantic rule associated with the production at the node.

Attributes values are confined to the children of node and by itself.
The non-terminal concerned must be in the head of production.
Terminals have synthesized attributes which are the lexical values (denoted by lex val) generated by the lexical analyzer.

Syntax directed definition of simple desk calculator

Production	Semantic rules
L ---> E_n	*L.val = E.val*
E ---> E_1+ T	$E.val = E_1.val + T.val$
E ---> T	*E.val = T.val*
T---> T_1*F	$T.val = T_i.val \times F.val$
T ---> F	*T.val = F.val*
F ---> (E)	*F.val = E.val*
F ---> digit	*F.val = digit.lexval*

Table 2.8: Syntax directed

Syntax-directed definition-inherited attributes

Production	Semantic Rules
D --->TL	*L.inh = T.type*
T ---> *int*	*T.type* =integer
T ---> *float*	*T.type = float*
L ---> L_1, **id**	L_1.inh = *L.inh* *addType (id.entry, Linh)*
L ---> **id**	*addType (id.entry, L.inh)*

Table 2.9: Syntax directed

• Symbol T is associated with a synthesized attribute *type.*
• Symbol L is associated with an inherited attribute *inh,*

Types of Syntax Directed Definitions

5.1 S-Attributed Definitions

Syntax directed definition that involves only synthesized attributes is called S-attributed. Attribute values for the non-terminal at the head is computed from the attribute values of the symbols at the body of the production.

The attributes of a S-attributed SDD can be evaluated in bottom up order of nodes of the parse tree. i.e., by performing post order traversal of the parse tree and evaluating the attributes at a node when the traversal leaves that node for the last time.

Production	Semantic rules
L ---> E_n	*L.val = E.val*
E ---> E_1+ T	$E.val = E_1.val + T.val$
E ---> T	*E.val = T.val*
T---> T_1*F	$T.val = T_i.val \times F.val$
T ---> F	*T.val = F.val*
F ---> (E)	*F.val = E.val*
F ---> digit	*F.val = digit.lexval*

Table 2.10: S-attributed Definitions

L-Attributed Definitions

The syntax directed definition in which the edges of dependency graph for the attributes in production body, can go from left to right and not from right to left is called L-attributed definitions. Attributes of L-attributed definitions may either be synthesized or inherited.
If the attributes are inherited, it must be computed from:
• Inherited attribute associated with the production head.
• Either by inherited or synthesized attribute associated with the production located to the left of the attribute which is being computed.
• Either by inherited or synthesized attribute associated with the attribute under consideration in such a way that no cycles can be formed by it in dependency graph.

Production	Semantic Rules
T ---> FT'	*T '.inh = F.val*
T ' ---> *FT$_1$'	T'$_1$.*inh =T'.inh x F.val*

Table 2.11: L-attributed Definitions

In production 1**,** the inherited attribute T' is computed from the value of F which is to its left. In production 2, the inherited attributed Tl' is computed from T'. *inh* associated with its head and the value of F which appears to its left in the production. i.e., for computing inherited attribute it must either use *from the above* or *from the left* information of SDD

Construction of Syntax Trees
SDDs are useful for is construction of syntax trees. A syntax tree is a condensed form of parse tree.

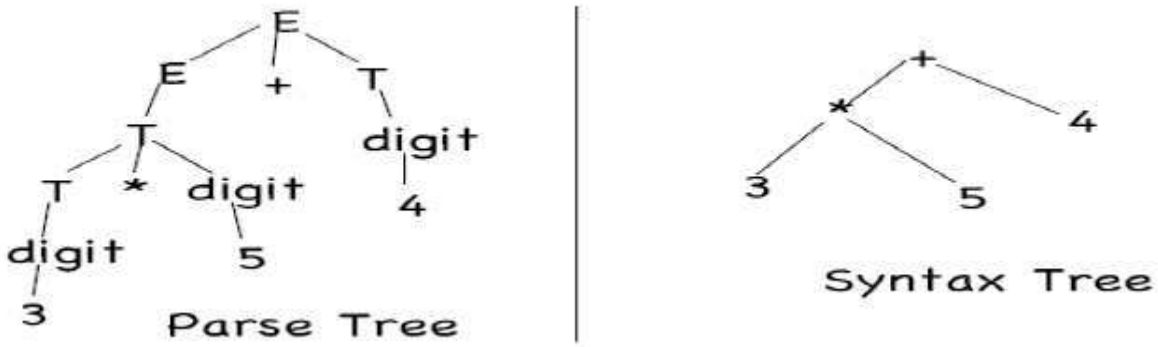

Figure 2.8 : Syntax Trees

• Syntax trees are useful for representing programming language constructs like expressions and statements.
•They help compiler design by decoupling parsing from translation.
• Each node of a syntax tree represents a construct; the children of the node represent the meaningfulcomponents of the construct.
• e.g. a syntax-tree node representing an expression E1 + E2 has label + and two children representing the sub expressions E1 and E2
• Each node is implemented by objects with suitable number of fields; each object will have an op field that is the label of the node with additional fields as follows:
——————————— If the node is a leaf, an additional field holds the lexical value for theleaf. This is created by function Leaf (op, val)
——————————— If the node is an interior node, there are as many fields as the node haschildren in the syntax tree. This is created by function Node (op, c1, c2,...,ck) .
Example: The S-attributed definition in figure below constructs syntax trees for a simple expression grammar involving only the binary operators + and -. As usual, these operators are at the same precedence level and are jointly left associative. All non-terminals have one synthesized attribute node, which

represents a node of the syntax tree.

S.no	Production	Semantic Rules
1	E->E_1+E	E.node =new Node('+',E1.node, T.node)
2	E->E_1-T	E.node =new Node('-',E1.node, T.node)
3	E->T	E.node=T.node
4	E->(E)	E.node=T.node
5	T->id	T.node=new leaf(id,id.entry)
6	T->num	T.node=new Leaf(num, num.val)

Syntax tree for a-4+c using the above SDD is shown below.

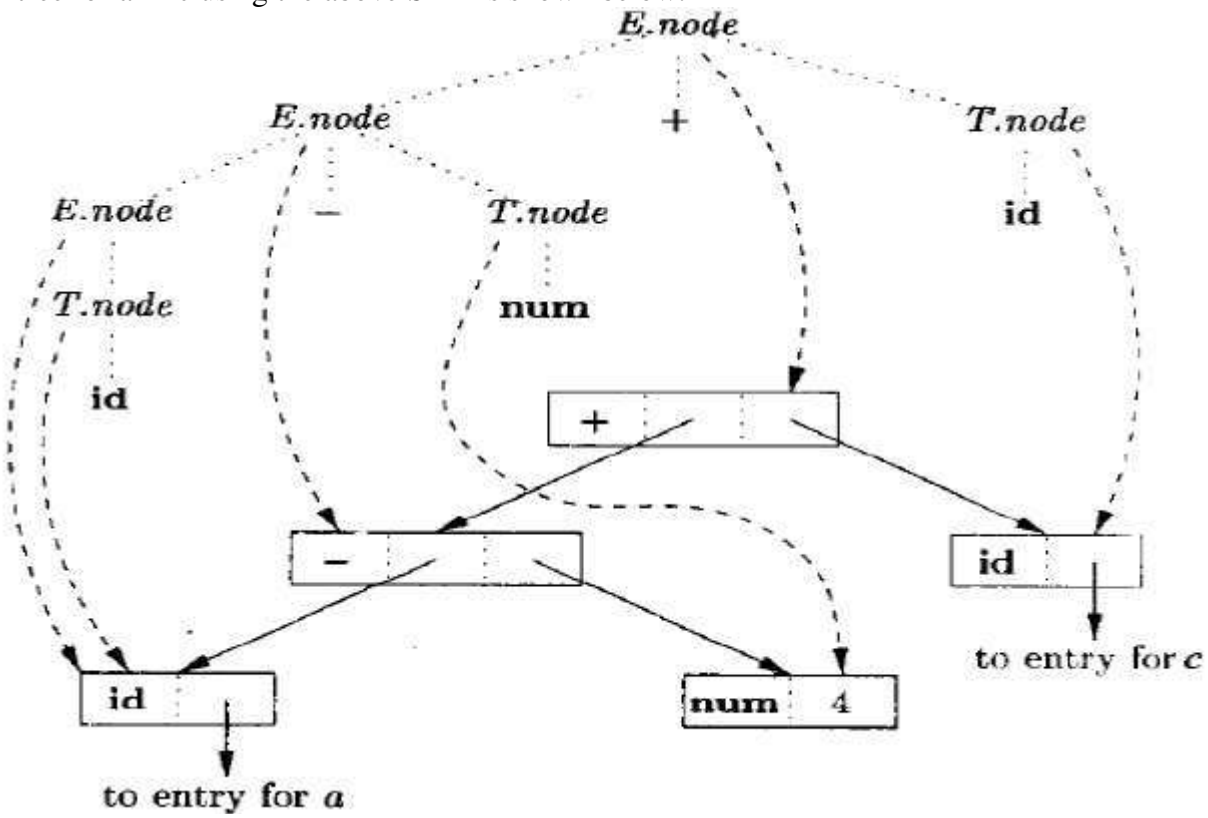

Figure 2.9:L-Attribute

5.2 Bottom-Up Evaluation Of SDT

Given an SDT, we can evaluate attributes even during bottom-up parsing. To carry out the semantic actions, parser stack is extended with semantic stack. The set of actions performed on semantic stack are mirror reflections of parser stack. Maintaining semantic stack is very easy.

During shift action, the parser pushes grammar symbols on the parser stack, whereas attributes are pushed on to semantic stack.

During reduce action, parser reduces handle, whereas in semantic stack, attributes are evaluated by the corresponding semantic action and are replaced by the result.

For example, consider the SDT

A → X Y Z	{A · a := f(X · x, Y · y, Z · z);}

Strictly speaking, attributes are evaluated as follows

A → X Y Z	{val[ntop] := f(val[top - 2], val[top - 1], val[top]);}

Evaluation of Synthesized Attributes

- Whenever a token is shifted onto the stack, then it is shifted along with its attribute value placed in val[top].
- Just before a reduction takes place the semantic rules are executed.
- If there is a synthesized attribute with the left-hand side non-terminal, then carrying out semantic rules will place the value of the synthesized attribute in val[ntop].

Let us understand this with an example:

	$E \rightarrow E_1$ "+" T	{val[ntop] := val[top-2] + val[top];}
	E → T	{val[top] := val[top];}
	$T \rightarrow T_1$ "*" F	{val[ntop] := val[top-2] * val[top];}
	T → F	{val[top] := val[top];}
	F → "(" E ")"	{val[ntop] := val[top-1];}
	F → **num**	{val[top] := num.lvalue;}

Table 2.12:

Figure shows the result of shift action. Now after performing reduce action by E → E * T resulting stack is as shown in figure.

Along with bottom-up parsing, this is how attributes can be evaluated using shift action/reduce action.

5.3 L-Attributed Definition

It allows both types, that is, synthesized as well as inherited. But if at all an inherited attribute is present, there is a restriction. The restriction is that each inherited attribute is restricted to inherit either from parent or from left sibling only.

For example, consider the rule

A → XYZPQ assume that there is an inherited attribute, "i" is present with each non-terminal. Then,

Z•I = f(A•i|X•i|Y•i) but Z•i = f(P•i|Q•i) is wrong as they are right siblings.

Semantic actions can be placed anywhere on the right hand side.

Attributes are generally evaluated by traversing the parse tree depth first and left to right. It is possible to rewrite any L-attributes to S-attributed definition.

L-attributed definition for converting infix to post fix form.

E → TE"
E" → +T #1 E" | ε
T → F T"
T" → * F #2 T" | ε
F → id #3

Where #1 corresponds to printing "+" operator, #2 corresponds to printing "*," and # 3 corresponds to printing id.val.

Look at the above SDT; there are no attributes, it is L-attributed definition as the semantic actions are in between grammar symbols. This is a simple example of L-attributed definition. Let us analyze this L-attributed definition and understand how to evaluate attributes with depth first left to right traversal. Take

the parse tree for the input string "a + b*c" and perform *Depth first left to right traversal, i.e. at each node traverse the left sub tree depth wise completely then right sub tree completely.*
Follow the traversal in. During the traversal whenever any dummy non-terminal is seen, carry out the translation.

Converting L-Attributed to S-Attributed Definition

Now that we understand that S-attributed is simple compared to L-attributed definition, let us see how to convert an L-attributed to an equivalent S-attributed.
Consider an L-attributed with semantic actions in between the grammar symbols. Suppose we have an L-attributed as follows:

$S \rightarrow A \{\} B$

How to convert it to an equivalent S-attributed definition? It is very simple!!
Replace actions by non-terminal as follows:

$S \rightarrow A\ M\ B$
$M \rightarrow \varepsilon$ {}

Convert the following L-attributed definition to equivalent S-attributed definition.

	E → TE"	
	E" → +T	#1 E" | ε
	T → F T"	
	T" → *F	#2 T" | ε
	F → id	#3

Table 2.13: Solution

Solution:
Replace dummy non-terminals that is, actions by non-terminals.

	E → TE"	
	E" → +T A E" | ε	
	A →	{ print("+");}
	T → F T"	
	T" → *F **B** T" |ε	
	B →	{ print("*");}
	F → id	{ print("id");}

Table 2.14: Solution

6. YACC

YACC—Yet Another Compiler Compiler—is a tool for construction of automatic LALR parser generator.

Using Yacc

Yacc specifications are prepared in a file with extension ".y" For example, "test.y." Then run this file with the Yacc command as "$yacc test.y." This translates yacc specifications into C-specifications under the default file mane "y.tab.c," where all the translations are under a function name called yyparse(); Now compile "y.tab.c" with C-compiler and test the program. The steps to be performed are given below:

Commands to execute

$yacc test.y

This gives an output "y.tab.c," which is a parser in c under a function name yyparse().

With –v option ($yacc -v test.y), produces file y.output, which gives complete information about the LALR parser like DFA states, conflicts, number of terminals used, etc.

```
$cc y.tab.c
$./a.out
```

Preparing the Yacc specification file

Every yacc specification file consists of three sections: the *declarations, grammar rules*, and *supporting* subroutines. The sections are separated by double percent "%%" marks.

```
  declarations
% %
  Translation rules
% %
  supporting subroutines
```

The declaration section is optional. In case if there are no supporting subroutines, then the second %% can also be skipped; thus, the smallest legal Yacc specification is

```
% %
  Translation rules
```

Declarations section

Declaration part contains two types of declarations—Yacc declarations or C-declarations. To distinguish between the two, C-declarations are enclosed within %{ and %}. Here we can have C-declarations like global variable declarations (int x=l;), header files (#include....), and macro definitions(#define...). This may be used for defining subroutines in the last section or action part in grammar rules.

Yacc declarations are nothing but tokens or terminals. We can define tokens by %token in the declaration part. For example, "num" is a terminal in grammar, then we define

% token num in the declaration part. In grammar rules, symbols within single quotes are also taken as terminals.

We can define the precedence and associativity of the tokens in the declarations section. This is done using %left, %right, followed by a list of tokens. The tokens defined on the same line will have the same precedence and associativity; the lines are listed in the order of increasing precedence. Thus,

```
%left '+' '-'
%left '*' '/'
```

are used to define the associativity and precedence of the four basic arithmetic operators '+','-','/','*'. Operators '*' and '/' have higher precedence than '+' and both are left associative. The keyword %left is used to define left associativity and %right is used to define right associativity.

Translation rules section

This section is the heart of the yacc specification file. Here we write grammar. With each grammar rule, the user may associate actions to be performed each time the rule is recognized in the input process. These actions may return values, and may obtain the values returned by previous actions. Moreover, the lexical analyzer can return values when a token is matched. An action is defined with a set of C statements. Action can do input and output, call subprograms, and alter external vectors and variables. The action normally sets the pseudo variable "$$" to some value to return a value. For example, an action that does nothing but return the value 1 is { $$ = 1;}

To obtain the values returned by previous actions, we use the pseudo-variables $1, $2, . . ., which refer to the values returned by the grammar symbol on the right side of a rule, reading from left to right.

7. Analysis Syntax Directed Definition

Are a generalizations of context-free grammars in which:

1. Grammar symbols have an associated set of Attributes;
2. Productions are associated with Semantic Rules for computing the values of attributes.

• Such formalism generates Annotated Parse-Trees where each node of the tree is a record with a field for each attribute (e.g., X.a indicates the attribute a of the grammar symbol X).

• The value of an attribute of a grammar symbol at a given parse-tree node is defined by a semantic rule associated with the production used at that node.

• We distinguish between two kinds of attributes:

1. **Synthesized Attributes:** They are computed from the values of the attributes of the children nodes.

2. **Inherited Attributes:** They are computed from the values of the attributes of both the siblings and the parent nodes.

Form of Syntax Directed Definitions

Each production, A ! α, is associated with a set of semantic rules: b := f(c1, c2, . . . , ck), where f is a function and either.

b is a synthesized attribute of A, and c1, c2, . . . , ck are attributes of the grammar symbols of the production, or

b is an inherited attribute of a grammar symbol in α, and c1, c2, . . . , ck are attributes of grammar symbols in α or attributes of A.

RGPVNOTES.IN

We hope you find these notes useful.

You can get previous year question papers at https://qp.rgpvnotes.in .

If you have any queries or you want to submit your study notes please write us at rgpvnotes.in@gmail.com

LIKE & FOLLOW US ON FACEBOOK
facebook.com/rgpvnotes.in