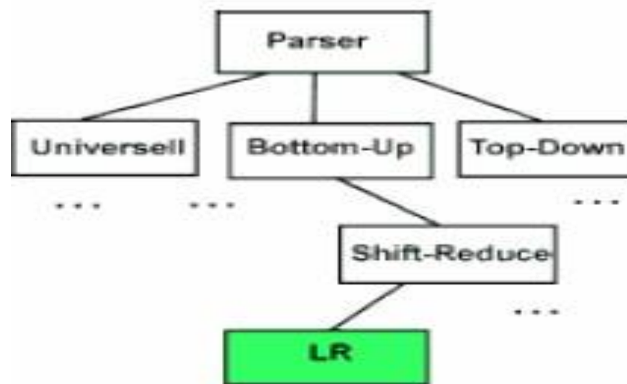


WHY LR-PARSING:

1.LRparsers can be constructed to recognize virtually all programming-language constructs for which context-free grammarscan be written.

2.TheLRparsing method is the most general non-backtracking shift- reduce parsing method known, yetitcanbeimplementedas efficiently as other shift-reducemethods.

3.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 predictiveparsers.

4,AnLR parser can detect a syntactic error as soon as it is possible to do so on a left-to-right scan of theinput.
The disadvantage is that it takes too much work to constuct an LR parser by hand for a typical programming-language grammar. But there are lots of LR parser generators available to make this taskeasy.

LR-PARSERS:

LR(k) parsers are most general non-backtracking shift-reduce parsers. Two cases of interest are k=0 and k=1. LR(1) is of practical relevance.

„L"stands for "Left-to-right" scan of input.
„R"" stands for "Rightmost derivation (in reverse)".

K"standsfornumber ofinput symbolsoflook-a-head thatareusedin makingparsingdecisions.When (K) is omitted, „K"is assumed to be 1.
LR(1) parsers are table-driven, shift-reduce parsers that use a limited right context (1 token) for handle recognition.

LR(1) parsers recognize languages that have an LR(1) grammar.

A grammar is LR(1) if, given a right-most derivation

$S \Rightarrow r_0 \Rightarrow r_1 \Rightarrow r_2$- - - $r_{n-1} \Rightarrow r_n \Rightarrow$sentence.

We can isolate the handle of each right-sentential form r_i and determine the production by which to reduce, by scanning r_i from left-to-right, going atmost 1 symbol beyond the right end of the handle of r_i.

Parser accepts input when stack contains only the start symbol and no remaining input symbol areleft.

LR(0)item: (no lookahead)

Grammar rule combined with a dot that indicates a position in its RHS.
Ex– 1: $S^I \rightarrow$ .S$
S→.
x S→.(L)

Ex-2: A→XYZ generates 4LR(0) items

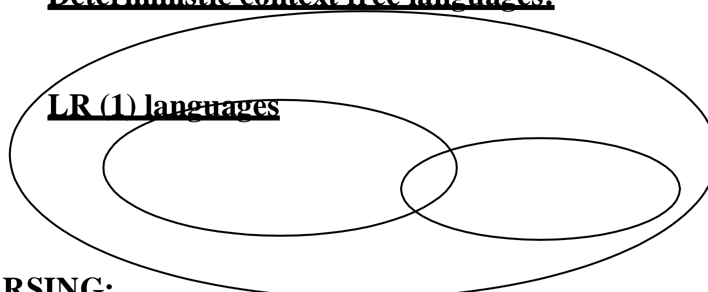
A→.XYZ
A→X.
YZ A→XY.
Z A→XYZ.

A→XY.Z indicates that the parser has seen a string derived from XY and is looking for one derivable from Z.

→ **LR(0) items play a key role in the SLR(1) table constructionalgorithm.**

→ **LR(1) items play a key role in the LR(1) and LALR(1) table constructionalgorithms. LR parsers have more information available than LL parsers when choosing aproduction:**

- LR knowseverything derived fromRHS plus„K"lookaheadsymbols.
- **LL just knows„K"lookaheadsymbols into what"sderived fromRHS.**
- **Deterministic context free languages:**
- **LR (1) languages**



LALR PARSING:

Example:

Construct C={I0,I1,… .... ,In} The collection of sets of LR(1)items

For each core present among the set of LR (1) items, find all sets having that core, and replace there sets by their Union# (clus them into a singleterm)

I_0 →same asprevious
I_1 → “
I_2 → “
I_{36} – Clubbing item I3 and I6 into one I36 item.
C →cC,c/d/$
C→cC,c/d/$
C→d,c/d/$
I_5→some as previous
I_{47}→C→d,c/d/$
I_{89}→C→cC, c/d/$

LALR Parsing table construction:

State	Action			Goto	
	c	d	$	S	C
I_o	S_{36}	S_{47}		1	2
1			Accept		
2	S_{36}	S_{47}			5
36	S_{36}	S_{47}			89
47	r_3	r_3			
5			r_1		
89	r_2	r_2	r_2		

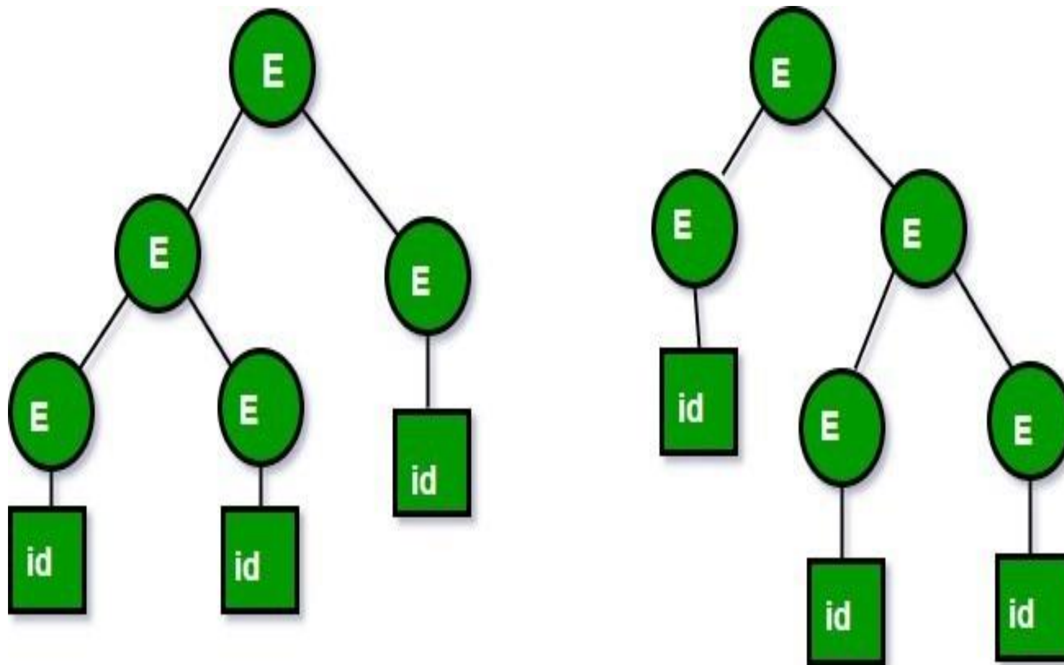
Ambiguous grammar:

A CFG is said to ambiguous if there exists more than one derivation tree for the given input string i.e., more than one **L**eft**M**ost **D**erivation **T**ree (LMDT) or **R**ight**M**ost **D**erivation **T**ree (RMDT).

Definition: G = (V,T,P,S) is a CFG is said to be ambiguous if and only if there exist a string in T* that has more than on parse tree.
where V is a finite set of variables.
T is a finite set of terminals.
P is a finite set of productions of the form, A -> α, where A is a variable and α ∈ (V ∪ T)* S is a designated variable called the start symbol.

For Example:

1. Let us consider this grammar : **E ->E+E|id**
We can create 2 parse tree from this grammar to obtain a string **id**+**id**+**id** :
The following are the 2 parse trees generated by left most derivation:



Both the above parse trees are derived from same grammar rules but both parse trees are different. Hence the grammar is ambiguous.

YACC PROGRAMMING

A parser generator is a program that takes as input a specification of a syntax, and produces as output a procedure for recognizing that language. Historically, they are also called compiler-compilers.

YACC (yet another compiler-compiler) is an LALR(1) (LookAhead, Left-to-right, Rightmost derivation producer with 1 lookahead token) parser generator. YACC was originally designed for being complemented by Lex.

Input File:
YACC input file is divided in three parts.

```
/* definitions */

 ....


%%

/* rules */

....

%%


/* auxiliary routines */

....
```

Input File: Definition Part:

- The definition part includes information about the tokens used in the syntax definition:
- %token NUMBER

```
%token ID
```

- Yacc automatically assigns numbers for tokens, but it can be overridden by

```
%token NUMBER 621
```

- Yacc also recognizes single characters as tokens. Therefore, assigned token numbers should no overlap ASCII codes.
- The definition part can include C code external to the definition of the parser and variable declarations, within **%{**and **%}** in the first column.
- It can also include the specification of the starting symbol in the grammar:

```
%start nonterminal
```

- The rules part contains grammar definition in a modified BNF form.
- Actions is C code in { } and can be embedded inside (Translation schemes).

Input File: Auxiliary Routines Part:

- The auxiliary routines part is only C code.
- It includes function definitions for every function needed in rules part.
- It can also contain the main() function definition if the parser is going to be run as a program.
- The main() function must call the function yyparse().

Input File:

- If yylex() is not defined in the auxiliary routines sections, then it should be included:

```
#include "lex.yy.c"
```

- YACC input file generally finishes with:

```
.y
```

Output Files:

- The output of YACC is a file named **y.tab.c**
- If it contains the **main()** definition, it must be compiled to be executable.
- Otherwise, the code can be an external function definition for the function **int yyparse()**
- If called with the **–d** option in the command line, Yacc produces as output a header file **y.tab.h** with all its specific definition (particularly important are token definitions to be included, for example, in a Lex input file).
- If called with the **–v** option, Yacc produces as output a file **y.output** containing a textual description of the LALR(1) parsing table used by the parser. This is useful for tracking down how the parser solves conflicts.

Semantics

Syntax Directed Translation:

- A formalist called as syntax directed definition is used fort specifying translations for programming languageconstructs.
- A syntax directed definition is a generalization of a context free grammar in which each grammar symbol has associated set of attributes and each and each productions is associated with a set of semanticrules

Definition of (syntax Directed definition) SDD :

- SDD is a generalization of CFG in which each grammar productions X->α is associated with it a set of semantic rules of the form

a: = f(b1,b2…..bk)

Where a is an attributes obtained from the function f.

A syntax-directed definition is a generalization of a context-free grammar inwhich:

- Each grammar symbol is associated with a set ofattributes.

- Thissetofattributesforagrammarsymbolispartitionedintotwosubsetscalledsynthesized and inherited attributes of that grammar symbol.

- Each production rule is associated with a set of semanticrules.
- Semantic rules set up dependencies between attributes which can be represented by a dependencygraph.
- This dependency graph determines the evaluation order of these semanticrules.
- Evaluation of a semantic rule defines the value of an attribute. But a semantic rule may also have some side effects such as printing avalue.

The two attributes for non terminalare :

The two attributes for non terminalare :
Synthesized attribute (S-attribute) :(↑)
An attribute is said to be synthesized attribute if its value at a parse tree node is determined from attribute values at the children of the node

Inherited attribute:(↑,→)

An inherited attribute is one whose value at parse tree node is determined in terms of attributes at the parent and | or siblings of thatnode.

- The attribute can be string, a number, a type, a, memory location or anythingelse.
- The parse tree showing the value of attributes at each node is called an annotated parse tree.

The process of computing the attribute values at the node is called annotating or decorating the parse tree.Terminals can have synthesized attributes, but not inherited attributes.

Annotated Parse Tree

• A parse tree showing the values of attributes at each node is called an Annotated parsetree.

• The process of computing the attributes values at the nodes is called annotating(or decorating) of the parse tree.
• Of course, the order of these computations depends on the dependency graph induced by the semanticrules.

Ex1:1) Synthesized Attributes : Ex: Consider the CFG :
S→ EN
E→E+T
E→E-T

E→ T
T→ T*F
T→T/F
T→F
F→(E)
F→digit N→;

Solution: The syntax directed definition can be written for the above grammar by using semantic actions for each production

Productionrule	**Semanticactions**
S→EN	S.val=E.val
E→E1+T	E.val =E1.val +T.val
E→E1-T	E.val = E1.val –T.val
E→T	E.val=T.val
T→T*F	T.val = T.val *F.val
T→T|F	T.val =T.val | F.val
F→ (E)	F.val=E.val
T→F	T.val=F.val
F→digit	F.val =digit.lexval
N→;	can be ignored by lexical Analyzer as;I is terminating symbol

For the Non-terminals E,T and F the values can be obtained using the attribute "Val".

The taken digit has synthesized attribute "lexval".

In S→EN, symbol S is the start symbol. This rule is to print the final answer of expressed.

Following steps are followed to Compute S attributed definition

Write the SDD using the appropriate semantic actions for corresponding production rule of the givenGrammar.

The annotated parse tree is generated and attribute values are computed. The Computation is done in bottom upmanner.

The value obtained at the node is supposed to be final output.

L-attributed SDT

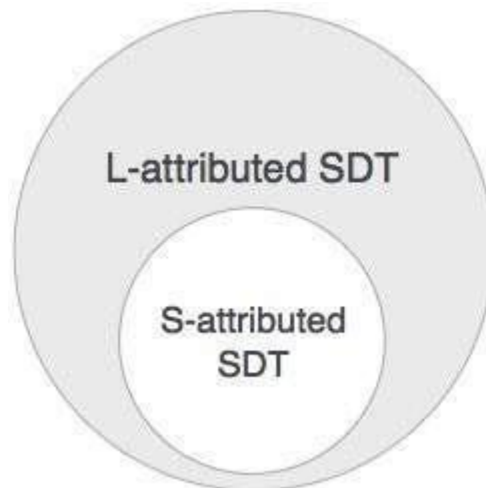
This form of SDT uses both synthesized and inherited attributes with restriction of not taking values from right siblings.

In L-attributed SDTs, a non-terminal can get values from its parent, child, and sibling nodes. As in the following production

```
S → ABC
```

S can take values from A, B, and C (synthesized). A can take values from S only. B can take values from S and A. C can get values from S, A, and B. No non-terminal can get values from the sibling to its right.

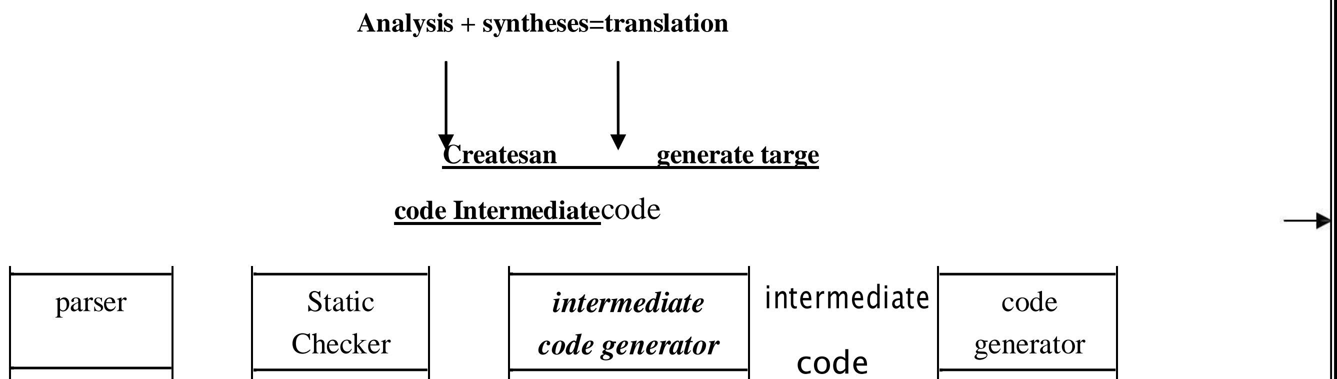
Attributes in L-attributed SDTs are evaluated by depth-first and left-to-right parsing manner.



We may conclude that if a definition is S-attributed, then it is also L-attributed as L-attributed definition encloses S-attributed definitions

Intermediate Code

An intermediate code form of source program is an internal form of a program created by the compiler while translating the program created by the compiler while translating the program from a high –level language to assembly code(or)object code(machine code).an intermediate source form represents a more attractive form of target code than does assembly. An optimizing Compiler performs optimizations on the intermediate source form and produces an objectmodule.



In the analysis –synthesis model of a compiler, the front-end translates a source program into an intermediate representation from which the back-end generates target code, in many compilers the source code is translated into a language which is intermediate in complexity between a HLL and machine code .the usual intermediate code introduces symbols to stand for various temporary quantities.

We assume that the source program has already been parsed and statically checked..the various intermediate code forms are:

a) Polishnotation
b) Abstract syntax trees(or)syntaxtrees
c) Quadruples
d) Triples three address code
e) Indirecttriples
f) Abstract machinecode(or)pseudocopde

postfix notation:

The ordinary (infix) way of writing the sum of a and b is with the operator in the middle: a+b. the postfix (or postfix polish)notation for the same expression places the operator at the right end, asab+.

In general, if e1 and e2 are any postfix expressions, and Ø to the values denoted by e1 and e2 is indicated in postfix notation nby e1e2Ø.no parentheses are needed in postfix notation because the position and priority (number of arguments) of the operators permits only one way to decode a postfixexpression.

Syntax Directed Translation:

- A formalist called as syntax directed definition is used fort specifying translations for programming languageconstructs.
- A syntax directed definition is a generalization of a context free grammar in which each grammar symbol has associated set of attributes and each and each productions is associated with a set of semanticrules

Definition of (syntax Directed definition) SDD :

SDD is a generalization of CFG in which each grammar productions X->α is associated with it a set of semantic rules of the form

a: = f(b1,b2…..bk)
Where a is an attributes obtained from the function f.

- A syntax-directed definition is a generalization of a context-free grammar inwhich:
- Each grammar symbol is associated with a set ofattributes.

Thissetofattributesforagrammarsymbolispartitionedintotwosubsetscalledsynthesized and inherited attributes of that grammar symbol.

- Each production rule is associated with a set of semanticrules.

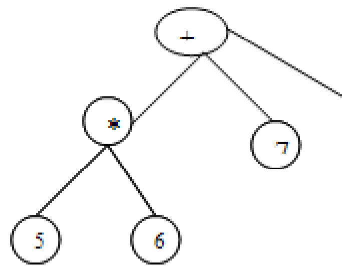
- Semantic rules set up dependencies between attributes which can be represented by a dependencygraph.

Annotated Parse Tree

- A parse tree showing the values of attributes at each node is called an Annotated parsetree.

- The process of computing the attributes values at the nodes is called annotating(or decorating) of the parse tree.Of course, the order of these computations depends on the dependency graph induced by the

Syntax tree:



Annotated parse tree :

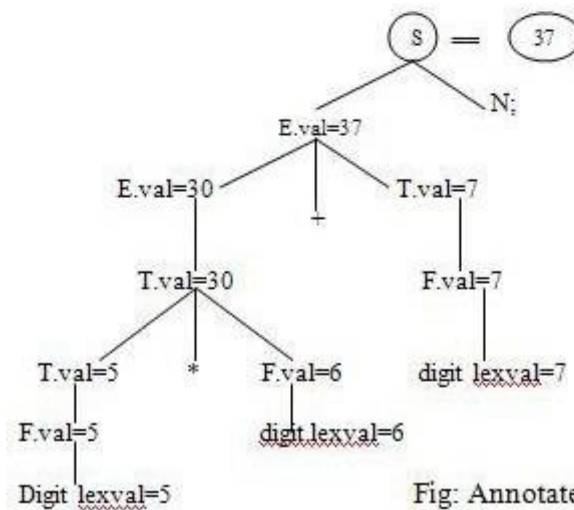


Fig: Annotated parse tree

ASSIGNMENT STATEMENTS

Suppose that the context in which an assignment appears is given by the following grammar. P □M D

M □ ε
D □D ; D | **id** : T | **proc id** ; N D ; S
N □ ε

Nonterminal P becomes the new start symbol when these productions are added to those in the translation scheme shown below.

Translation scheme to produce three-address code for assignments

S →id := E { p : = lookup (**id**.name);
ifp≠nil **then**
emit(p ' : =' E.place)
elseerror }

E → E_1 + E_2 { E.place : = newtemp;

emit(E.place ': =' E_1.place ' + ' E_2.place) }

E → E_1 * E_2 { E.place : = newtemp;

emit(E.place ': =' E_1.place ' * ' E_2.place) }

E →- E_1 { E.place : = newtemp;

emit (E.place ': =' 'uminus' E_1.place) }

E →(E_1) { E.place : = E_1.place }

E → id { p : = lookup (**id**.name);

ifp≠nil **then**

E.place : = p

elseerror }

Flow-of-Control Statements

We now consider the translation of boolean expressions into three-address code in the context of if-then, if-then-else, and while-do statements such as those generated by the following grammar:

S → **if** E **then** S1

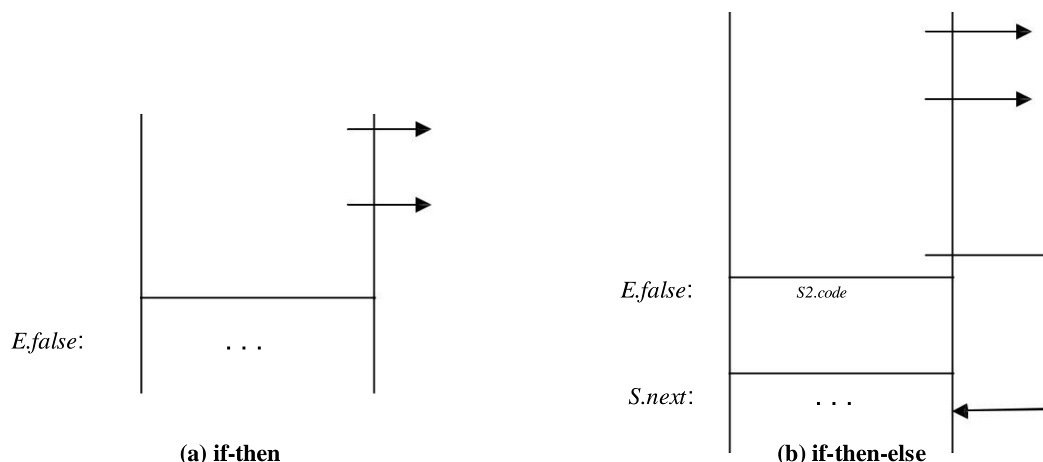
if E **then** S1 **else**

| S2

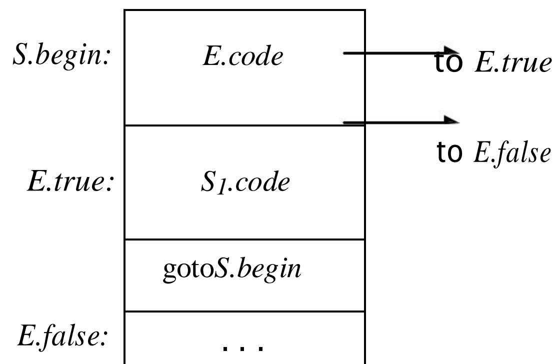
| **while** E **do** S1

In each of these productions, *E* is the Boolean expression to be translated. In the translation, we assume that a three-address statement can be symbolically labeled, and that the function *newlabel*returns a new symbolic label each time it is called.

- E.true is the label to which control flows if E is true, and E.false is the label to which control flows if E is false.
- The semantic rules for translating a flow-of-control statement S allow control to flow from the translation S.code to the three-address instruction immediately following S.code.
- S.nextis a label that is attached to the first three-address instruction to be executed after the code for **Code for if-then , if-then-else, and while-do statements**



(a) if-then **(b) if-then-else**



(c) while-do

PRODUCTION	SEMANTIC RULES
$S \rightarrow$ **if** E **then** S_1	$E.true := newlabel;$ $E.false := S.next;$ $S_1.next := S.next;$ *S.code : = E.code || gen(E.true „:") || S₁.code*
$S \rightarrow$ **if** E **then** S_1 **else** S_2	$E.true := newlabel;$ $E.false := newlabel;$ $S_1.next := S.next;$ $S_2.next := S.next;$ *S.code : = E.code || gen(E.true „:") || S₁.code ||* *gen(„**goto**" S.next) ||* *gen(E.false „:") || S₂.code*
$S \rightarrow$ **while** E **do** S_1	$S.begin := newlabel;$ $E.true := newlabel;$ $E.false := S.next;$ $S_1.next := S.begin;$ *S.code : = gen(S.begin „:")|| E.code ||* *gen(E.true „:") || S₁.code ||* *gen(„**goto**" S.begin)*