Semantic Analysis

Need for Semantic analysis

Semantic analysis add semantic information to the parse tree & performs certain checks based on the information

Following things are done in Semantic analysis.

1. Disambiguate overloaded operators: If an operator is overloaded, one could like to specify the meaning of that operator.

2. Type checking: It is the process of verifying that each operation executed in the program respects the type system of the language i.e it checks that all operands in an expression are of appropriate types and number.

3. Uniqueness checking: whether a variable name is unique or not.

4. Name checks: Checks whether any variable has a name which is not allowed.

There are 2 notations for associating semantic rules with productions

1. Syntax directed definitions
2. Translation schemes.

With both SDD & translation schemes, we parse the input token stream, build the parse tree & then traverse the tree as needed to evaluate the semantic rules at the parse tree nodes.

* Syntax Directed Definition (see text - 280)

is a generalization of context free grammar in which each grammar production $X \to \alpha$ is associated with it a set of semantic rules of the form $a = f(b_1, b_2 \ldots b_k)$ where a is an attribute obtained from the function f. This attribute can be a string, a number, a type, a memory location or anything else.

There are two types of attributes

Synthesized attributes

Consider $X \to \alpha$ be a CFG and $a = f(b_1, b_2 \ldots b_k)$ where a is the attribute of X and $b_1, b_2 \ldots b_k$ are attributes belonging to production symbols. The value of attribute a is called synthesized attribute. Synthesized attributes at the children of that node in the parse tree.

Production rule	Semantic actions
$S \to E\ n$ (n→newline)	Print (E.val)
$E \to E_1 + T$	$E.val = E_1.val + T.val$
$E \to E_1 - T$	$E.val = E_1.val - T.val$
$E \to T$	$E.val = T.val$
$T \to T_1 * F$	$T.val = T_1.val * F.val$
$T \to T_1 / F$	$T.val = T_1.val / F.val$
$T \to F$	$T.val = F.val$
$F \to (E)$	$F.val = E.val$
$F \to digit$	$F.val = digit.lexval$

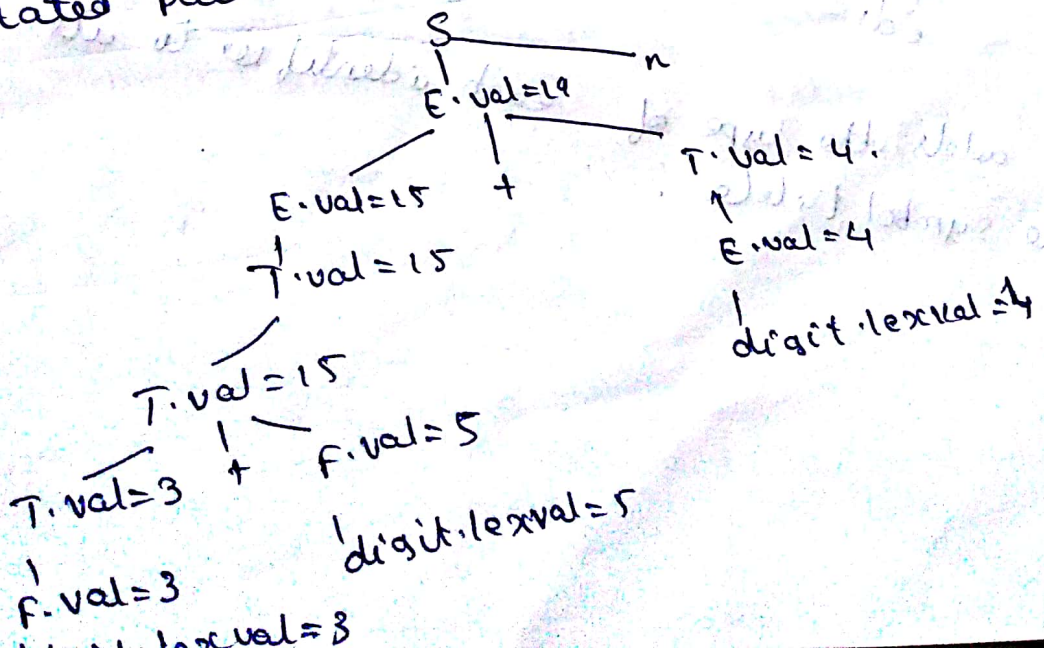
Here val is a synthesized attribute. The token digit has synthesized lexical value which can be obtained from lexical analyser.

In SDD terminals have synthesized attributes only

In a parse tree at each node the semantic rule is evaluated for computing S attributed definition.

Parse tree containing the values of attributes at each node is called annotated parse tree.

Annotated parse tree for input string 3*5+4n.



When we apply the semantic rule at this node, ... left ch...

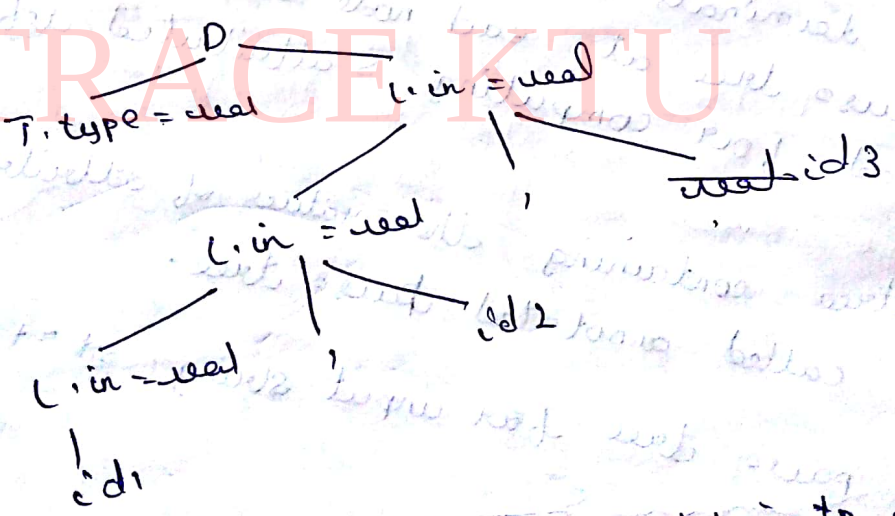
for the computation of attribute start from leftmost bottommost node. n-new line character.

Inherited attribute

An inherited attribute is one whose values at a node in a parse tree is defined in terms of attributes at the parent or siblings of that node.

Production	Semantic rule
$D \rightarrow TL$	$L.in = T.type$
$T \rightarrow int$	$T.type = integer$
$T \rightarrow real$	$T.type = real$
$L \rightarrow L_1, id$	$L_1.in = L.in$, addtype (id.entry, L.in)
$L \rightarrow id$	

Annotated parse tree for i/p real id1, id2, id3 $, $



addtype () adds the type of each identifier to its entry in the symbol table.

Dependency graph.

are useful tool tool for determining an evaluation order for the attribute instances in a given parse tree. It depicts the flow of information among attribute instances in a particular parse tree. The graph ~~tree~~ has a node for each attribute and an edge from one attribute instance to another means that value of 1st is needed to compute 2nd.

The dependency graph for a pause tree is

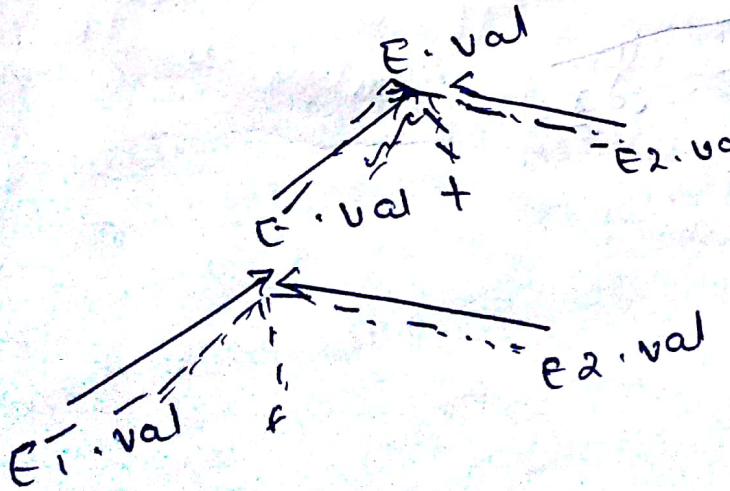
for each node 'n' in the pause tree do for each attribute 'a' of the grammar. Symbol at 'n' do construct a node in the dependency graph for 'a' for each semantic rule $b = f(c_1, c_2, \ldots c_k)$ associated with the production used at 'n' do

for $i = 1$ to k do

construct an edge from node for c_i to node for b.

eg: $E \to E_1 + E_2$ $\quad E.val = E_1.val + E_2.val$

$E \to E_1 * E_2$ $\quad E.val = E_1.val * E_2.val$.



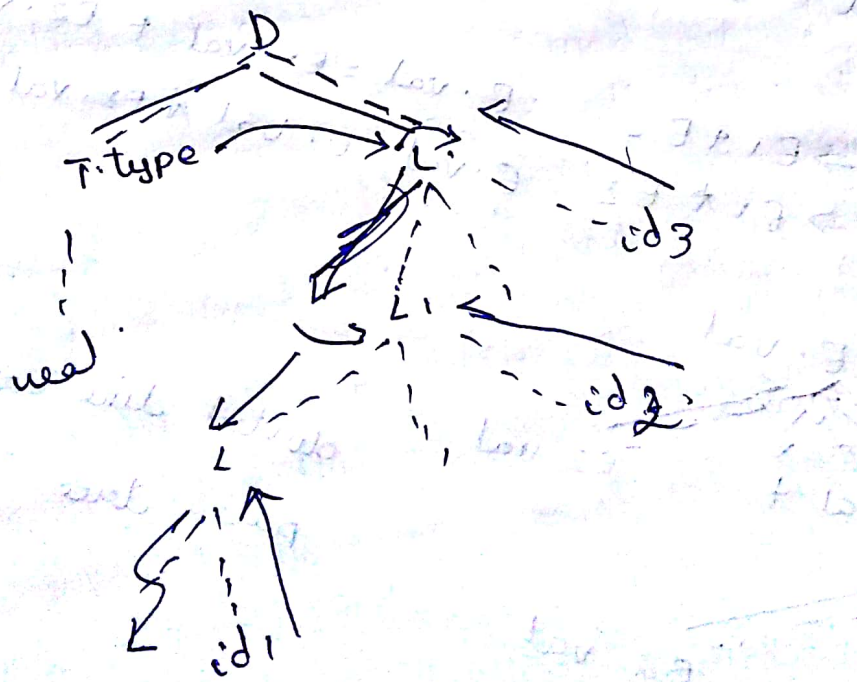
dotted line represent the pause tree.

Evaluation order:

The topological sort of the dependency graph decides the evaluation order in a parse tree. In deciding evaluation order ~~in a parse tree~~ the semantic rule in the syntax directed definitions are used.

Translation using SDD consist of:

1. Grammar is used to construct parse tree from the input.
2. Dependency graph is constructed
3. Perform topological sort of dependency graph to obtain evaluation order from semantic rules.
4. Evaluate semantic rules based on their order to obtain translation of input string.



Methods for evaluating semantic rule.

1. Parse tree method: At compile time, these methods obtain an evaluation order from a topological sort of the dependency graph constructed from the parse tree for each input. This method will fail to find an evaluation order only if the dependency graph contain cycle.

2. Rule based method: At compiler construction time, the semantic rules associated with the production are analyzed either by hand or by a specialised tool. For each production the order of attributes evaluation is ~~predefined~~ predetermined at compiler construction time.

3. Oblivious method: evaluation order is choosen without considering semantic rules. for eg: if translation takesplace during parsing then the order of evaluation is forced by the parsing method, independent of the semantic rules.

S-attributed definition

SDD with only synthesized attributes is called S- attributed definition. Attributes can be evaluated using bottom up parser. The parser can keep the values of the synthesized attributes associated with the grammar symbols on the stack.

Now consider the node for the production ... attribute *T.val* at this node is defined by ... SEMANTIC RULE

When ... left ... val ... T ... L ...

Bottom UP evaluation

A translator for S attributed definition is implemented using LR parser generator. A bottom up method is used to parse the i/p string. A parser stack is used to hold the values of synthesized attribute. The stack is implemented as a pair of state and value. Each state entry is the pointer to the LR(1) parsing table. Parser stack is denoted as stack[i] & is a combination of state[i] and value[i].

eg: $X \rightarrow ABC$

State	value
A	A.a
B	B.b
C	C.c

→ top

Before ABC is reduced to X, the value of the attribute C.c is in val[top], that of B.b in val[top-1] and that of A.a in val[top-2].

After reduction the left hand side symbol of the production is X will be placed in the stack along with the value of X.x at the top. After reduction top is decremented by 2 the state covering X is placed at the top of state[top] and the value of synthesized attribute X.x is put in value[top]. If the symbol has no attribute then the corresponding entry in the value arrays will be kept condebined.

eg :

Production	Code fragment
$S \to En$	print (val[top])
$E \to E_1 + T$	val[ntop] = val[top-2] + val[top]
$E \to T$	
$T \to T_1 * F$	val[ntop] = val[top-2] * val[top]
$T \to F$	
$F \to (E)$	val[ntop] = value[top-1]
$F \to digit$	

ntop - new top of the stack.

Input	State	Value	Production used
3*5+4n	-	3	F → digit
*5+4n	3	3	T → F
*5+4n	F	3	
*5+4n	T	3	
5+4n	T*	3 -	
+4n	T*5	3-5	F → digit
+4n	T*F	3-5	T → T*F
+4n	T	15	E → T
+4n	E	15	
4n	E+	15-	F → digit
n	E+4	15-4	T
n	E+F	15-4	T → F
n	E+T	15-4	E → E+T
	E	19	
	En	19-	S → En.
	S	19	

L-attributed definition contain both synthesized and inherited attributes and they do not need a dependency graph to evaluate them. The attributes are evaluated in depth first order. L is for left because attribute information appears to flow from left to right. A SDD is L attributed if each inherited attribute of X_j, $1 \le j \le n$ on right side of $A \to X_1 X_2 \ldots X_n$ depends only on

1) the attributes of symbols $X_1, X_2, \ldots X_{j-1}$ to left of X_j in the production and
2) inherited attributes of A.

Inherited attributes in L-attributed definitions can be computed by preorder traversal of the parse tree.

Depth 1st evaluation order for L attributed definition

```
procedure dvisit (n: node)
begin
for each child m of n from left to right
begin
evaluate inherited attributes of m
dfvisit(m)
end
evaluate synthesized attributes of n
end.
```

Translation scheme -

is a CFG in which attributes are associated with the grammar symbols and semantic actions enclosed b/w braces {} are inserted within the right side of production.

eg: $E \rightarrow E_1 + T \{ E.val = E_1.val + T.val \}$

yacc uses translation schemes. Translation schemes deal with both synthesized and inherited attributes. execution of syntax directed definition can be done bs syntax directed translation scheme.

Rules four implementing S-attributed translation scheme

The action is put at the end of the production

eg: production action

$T = T_1 * F$ $\quad T.val = T_1.val * F.val$.

translation scheme

$T \rightarrow T_1 * F \{ T.val = T_1.val * F.val \}$.

A synthesized attribute for the non terminal on the left can only be computed after all attributes it references have been computed. The action computing such attributes can usually be placed at the end of the right side of the production.

of this node is 3. Similarly ... $T \to T * F$ ...
ha...

Rules for implementing L-attributed definitions

An inherited attribute for a symbol on the right side of the production must be computed in an action before the symbol.

$D \to T \{ L.in = T.type \} L$

Bottom up evaluation of inherited attributes

Bottom up parser reduces the right side of the production $A \to ABC$ by removing C, B & A from parser stack.

Parser stack is implemented as is implemented as combination of state & value.

The state [i] is for grammar symbol A & value[i] holds the synthesized attributes A.a

eg: $S \to T\ List$ { List.in = T.type }

$T \to int$ { T.type = int }

$T \to float$ { T.type = float }

$List \to List_1, id$ { $List_1$.in = List.in } { enter_type (id.entry, List.in) }

$List \to id$ { enter_type (id.entry, List.in) }

For input int a, b, c bottom up evaluation is

i/p string	state	
int a, b, c		
a, b, c	int	
, b, c	T	T → int
, b, c	T a	
b, c	T list	
, c	T list ,	
, c	T list , b	
c	T list	list → list , id
	T list ,	
	T list , c	list → list , id
	T list	
	S	S → T list

when T → int

state	value
T	int
T list , b	

when, list → list , b

translation scheme

production rule code

S → T list
T → int
T → bloat
list → list , id
list → id

value[top] = int
value[top] = bloat
enter_type (value[top], value[top-3])
enter_type (value[top], value[top-1]).

Top down translation

Implementation of attributed definition using top down translation scheme

Production	action
$E \rightarrow E + T$	{ E.val = E1.val + T.val }
$E \rightarrow E - T$	{ E.val = E1.val - T.val }
$E \rightarrow T$	{ E.val = T.val }
$T \rightarrow (E)$	{ T.val = E.val }
$T \rightarrow digit$	{ T.val = digit.lexval }

1st remove left recursion -

$E \rightarrow T\ P$ { P.in = T.val } { E.val = P.s }

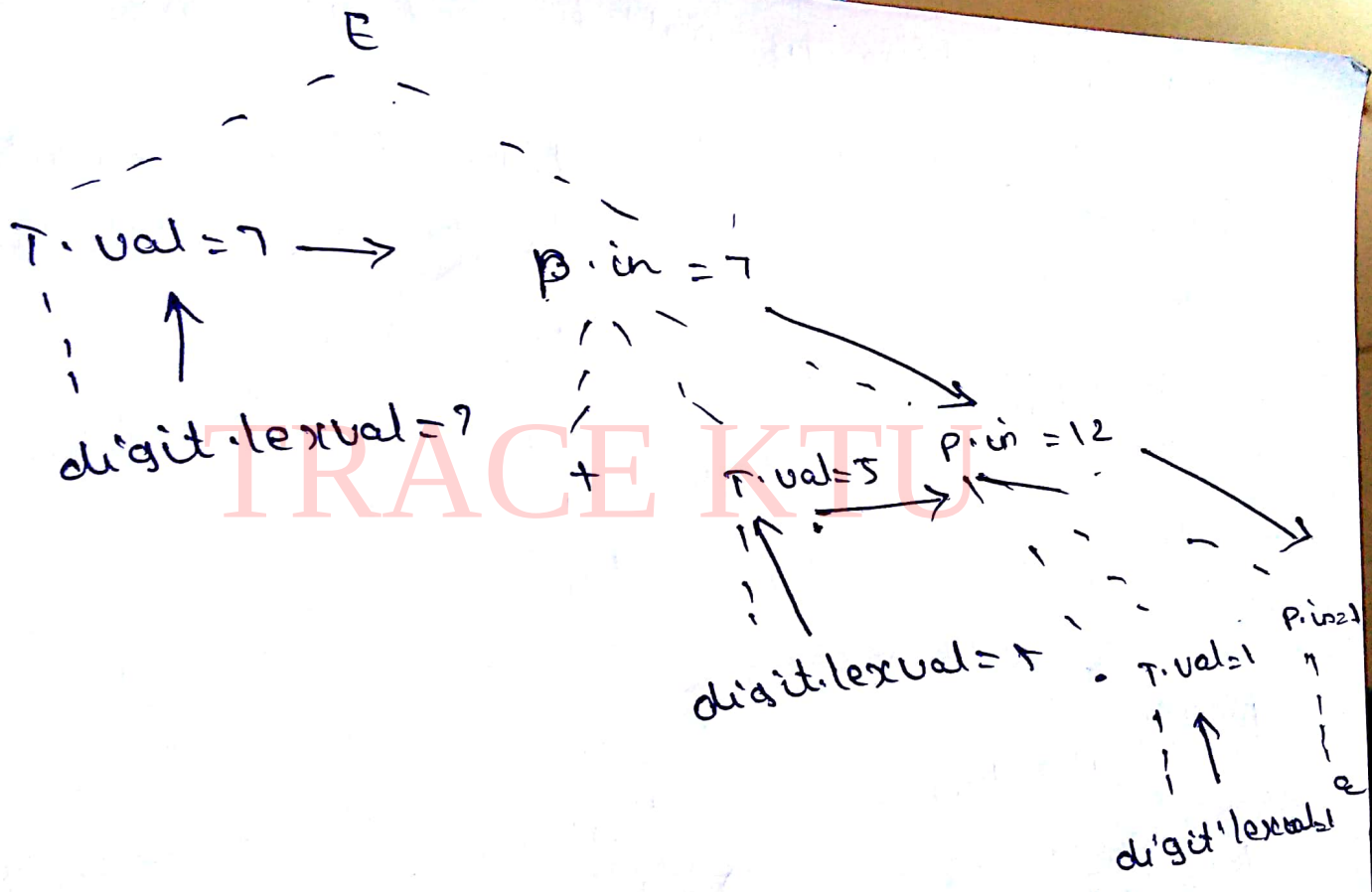
$P \rightarrow + T\ P1$ { P1.in = T.in + T.val } { P.s = P1.s }

$P \rightarrow - T\ P1$ { P1.in = P.in - T.val } { P.s = P1.s }

$P \rightarrow \epsilon$ { P.s = P.in }

$T \rightarrow (E)$ { T.val = E.val }

$T \rightarrow digit$ { T.val = digit.lexval }



$E \to E + T \mid T$

$E \to T$

$E \to T E'$