UNIT-IV Semantic Analysis

V. Bhargavi
11501A1281

- Semantic Analysis phase checks the Source program for Semantic errors and gathers type of information for the Subsequent code generations.

- There are two notations associated with Semantic rules with productions
  1. Syntax-directed definitions
  2. translation scheme

- Generally Conceptual view of Syntax directed translation is

Input String → parse tree → dependency graph → evalution order for Semantic rules

Syntax directed definition:-(SDD):-

- Syntax directed definition is a generalization of Context free glammar in which each grammar Production A→α is associated with it a set of semantic rules of the form b:= f(c1, c2, .. ck) where f is a function, b is attributes.

- b is a synthesized attribute of A and c1, c2 .. ck are attributes belonging to the grammar symbols of productions.

- The value of Synthesized attribute at a node is compute from the values of attributes at the children of that node in parse tree [illegible]

- b is an <u>Inherited attribute</u> of one of the grammar Symbol on the right side of production (i.e α), and c1, c2 -- ck are are attributes belonginging to the grammar Symbols of the Production (i.e A(or)α)
- Inherited attributes can be computed from the values of the attribute at the sibilings and parent of that node.

Synthasized attribute

Ex: How to compute Syntherized attributes Grammar for Simple desk calculator

L → En
E → E.+T
E → T
T → T.*F
T → F
F → (E)
F → digit

- Syntax directed defination can be written for above grammar by using Semantic actions for each Production

Production	Semantic rules
L → En	Print (E.val)
E → E+T	E.val := E.val + T.val
E → T	E.val := T.val
T → T*F	T.val := T.val * F.val
T → F	T.val := F.val
F → (E)	F.val := E.val
F → digit	F.val := digit.lexvalue

- For non terminals E, T, F the values can be obtained by using attribute "val".
- The token digit has a Synthesized attribute lexval whose value is assumed to be supplied by the lexical analyzer.

  The above L is a just a procedure that prints as output of value generated by E.

- In SDD terminals are assumed to have Synthesized attributes only so no need to provide Semantic Rules for terminals.

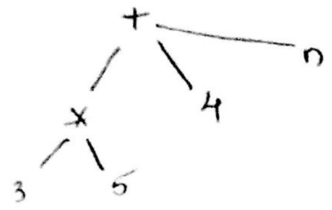
- In SDD that uses Synthesized attributes exclusively is said to be an S-attributed defination

- In a parse tree at each node the Semantic Rule is evaluated for annotated the S-attributed defination process is in bottom-up fashion. i.e from leaves to root

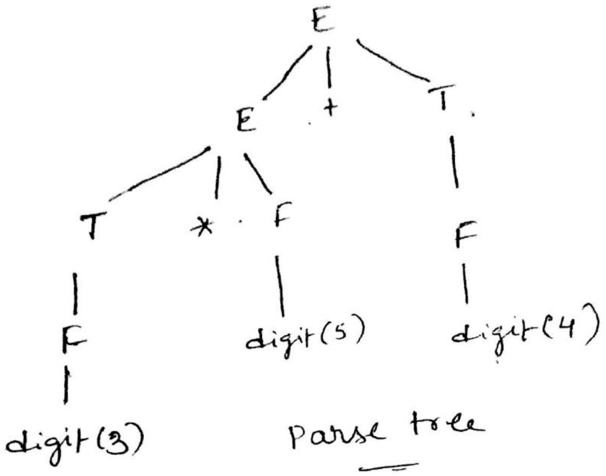
<u>Steps to Compute</u> <u>S-attributed defination</u>

1. Write S.D.D using the appropriate semantic actions for corresponding production rule of given grammar
2. Annotated parse tree is generated and attribute values are computed. computation can be done in bottom-up fashion
3. The value obtained at the root node is supposed to be the final output.

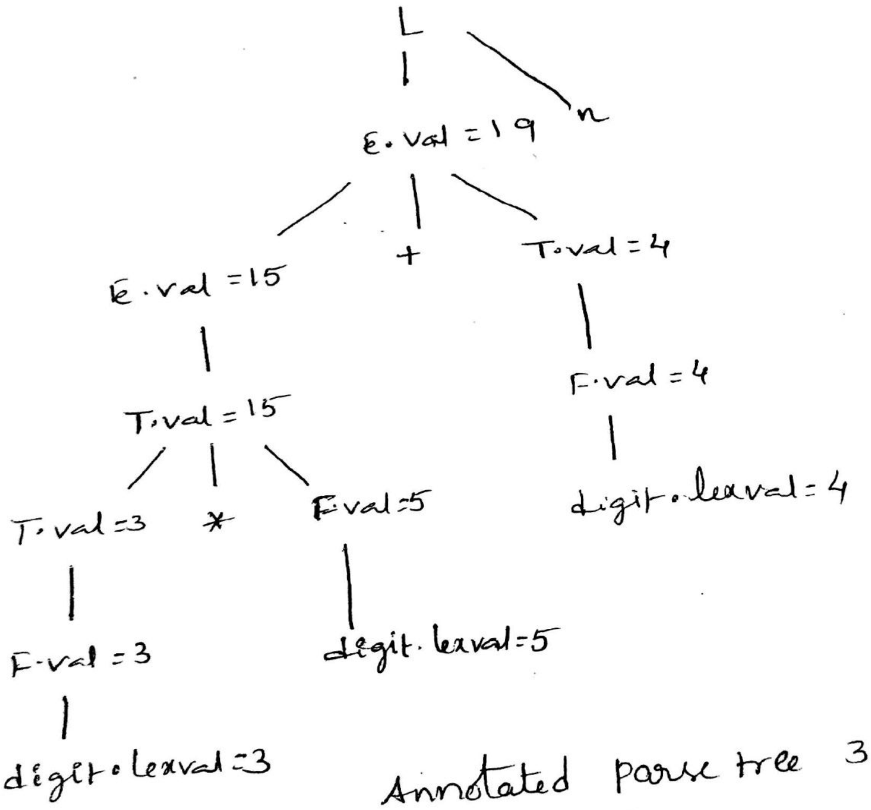
Ex:- Given expression 3*5+4 followed by new line for simple desk calculata



Syntax tree



Parse tree



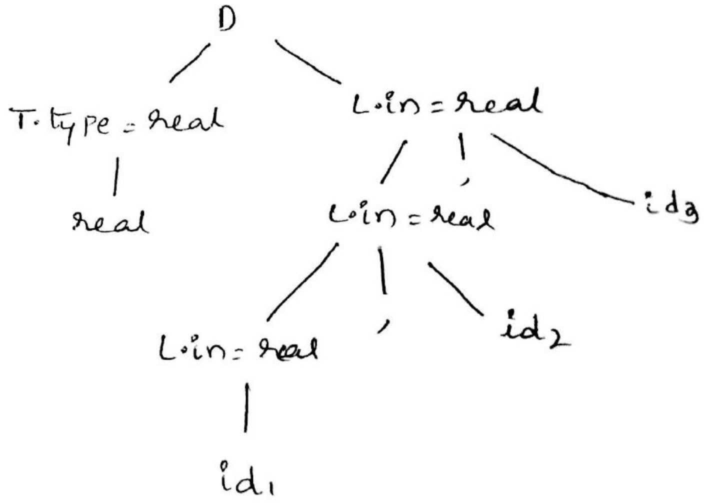
Annotated parse tree 3*5+4n

1. Inherited attribute

- The value of inherited attribute at a node in a parse tree is defined using the attribute value the Parent (or) Sibilings.

Ex:-

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



The above figure for the sentence real id_1, id_2, id_3

The value of L.in at the three L-nodes gives the type of the identifiers id_1, id_2, id_3.

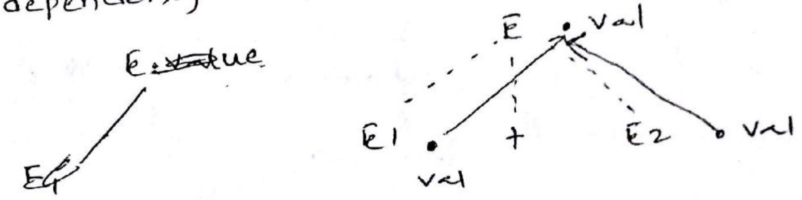
- These values are determined by computing the value of t attribute T.type at the left child of the root and then evaluating L.in topdown at the three L-nodes in the right subtree of the root.
- At each node we call procedure addtype to insert into symbol table the fact that the identifier at the right child of this node has type real.

Dependency Graph: The directed graph that represents the interdependencies between Synthesided and inherited attributes at nodes in the parse tree is called dependency graph. for the rule $X \to YZ$ the semantic action is given by $X.x \to f(Y.y, Z.z)$ then synthesized attribute is $X.x$ and $X.x$ depends upon attributes $Y.y$ and $Z.z$

Ex:-

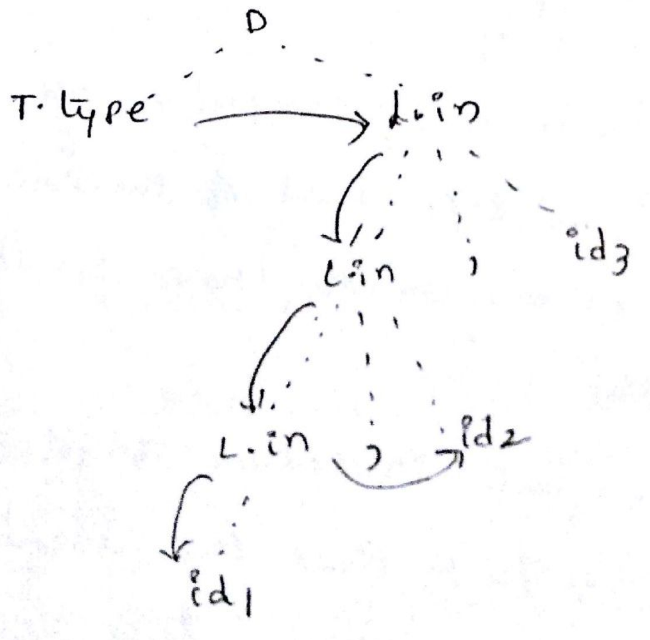
Production rule	Semantic rule
$E \to E_1 + E_2$	$E.val := E_1.val + E_2.val.$

The dependency Graph is shown



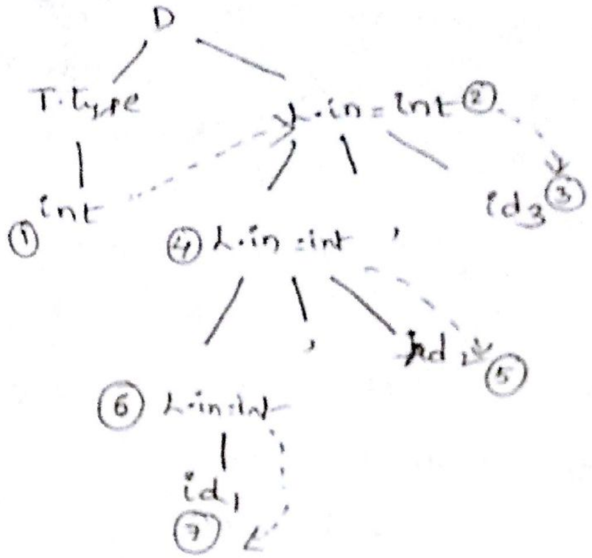
- hear E.val depends upon $E_1.val$ and $E_2.val$
- dotted lines represent the parse tree and are not part of the dependency graph.

dependency graph for inherited attribute.



Evalution order:-

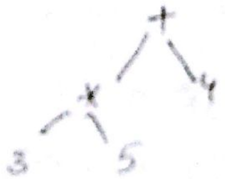
The topological sort of the dependency graph decided the evalution order in a parse tree.



- From the topological sort of the dependency graph we obtain an evalution order for the Semantic rule
- Evalution of the Semantic rules in this order yields the translation of the input string.

Application of SDT:-

Syntax tree: Syntax tree is condensed form of pars tree useful for representating language construct



Syntax directed translation can be based on syntax tree as well as parse tree

Construction of Syntax tree for expression

Construction of Syntax tree for an expression means translation of expression into postfix form.

- The nodes for each operator and operand is created.
- Each node can be represented as a record with multiple fields
- Following are the functions used in Syntax tree for expression

$E \rightarrow E + T$

$E \rightarrow E - T$

$E \rightarrow E * T$

$E \rightarrow T$

$T \rightarrow id$

$T \rightarrow num$

① MKnode (op, left, right) :- This function creates a node with the field operator having operator as label and the two pointer to left and right.

② mkleaf (id, entry) : Creates an identifier node with label id and a pointer to symbol table is given by 'entry'.

③ mkleaf(num, val) :- Creats node for number with label num and val is for value of that number

Ex:- Construction of Syntax tree for expression is

$x * y - 5 + z$

1) Convert expression into Postfix form : $x y * 5 - z +$

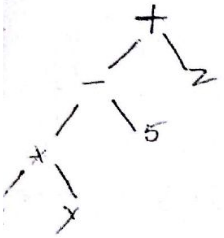
2) Make use of functions mkleaf, mknode.

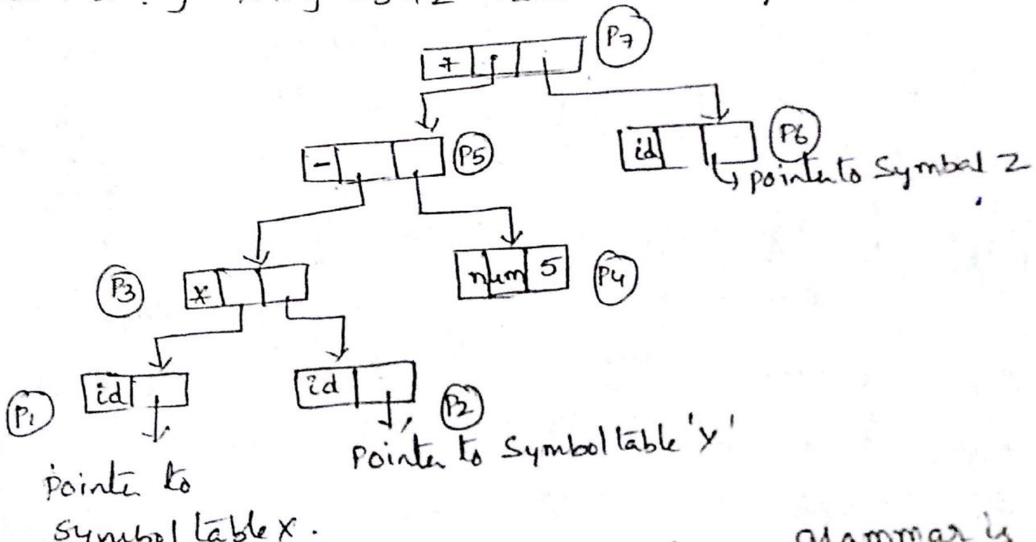
3) Sequence of function calls is given

checking .

Symbol	operation
x	P_1 = mkleaf (id, ptr to entry x)
y	P_2 = mkleaf (id, ptr to entry y)
*	P_3 = mknode (*, P_1, P_2)
5	P_4 = mkleaf (num, 5)
−	P_5 = mknode (−, P_3, P_4)
z	P_6 = mkleaf (id, ptr to entry z)
+	P_7 = mknode (+, P_5, P_6)

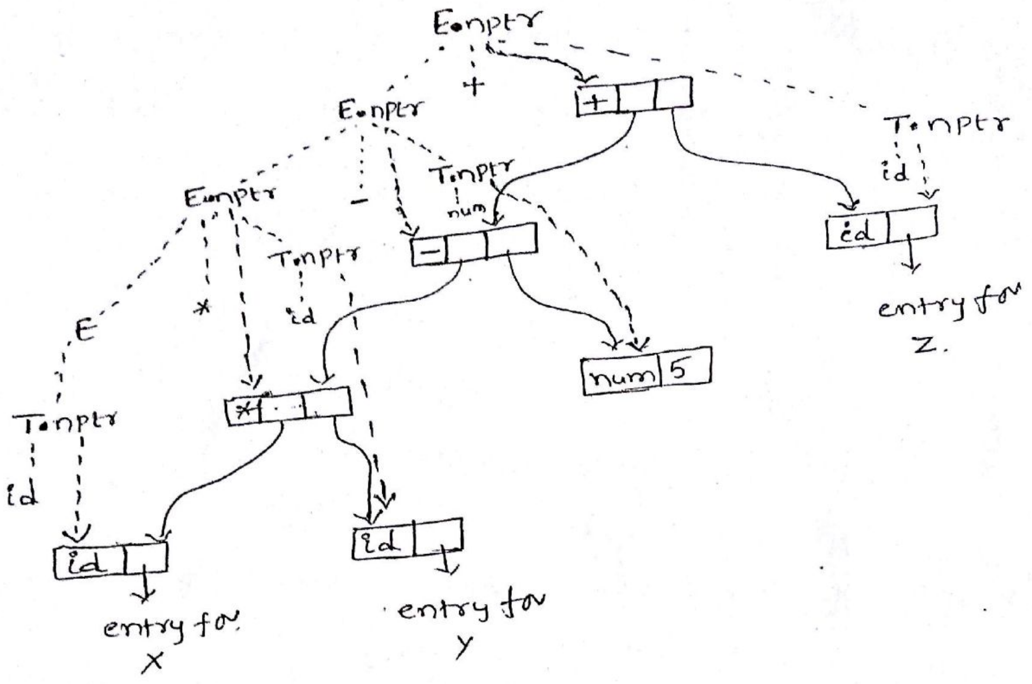
consider string x*y −5 +z now draw syntax tree





- Syntax directed defination for the above grammar is given below

Production rule	Semantic operation
E → E + T	E.nptr := mknode('+', E.nptr, T.nptr)
E → E − T	E.nptr := mknode('−', E.nptr, T.nptr)
E → E * T	E.nptr := mknode('*', E.nptr, T.nptr)
E → T	E.nptr := T.nptr
T → id	E.nptr := mkleaf (id, id.ptr_entry)
T → num	T.nptr := mkleaf (num, num.val)

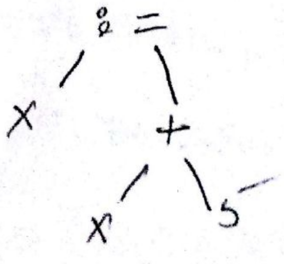


constructed Syntax tree

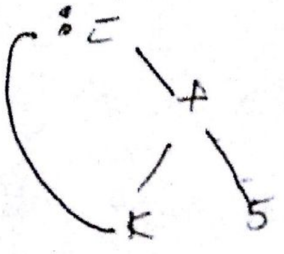
Directed Acyclic graph for expression (DAG)

- DAG is drawn by identifying the common subexpressio
- DAG has nodes representing the subexpression in expression.
- Difference between DAG and syntax tree is that common subexpression has more than one parent and in syntax tree the common sub expression would be represented as duplicated subtree.

Ex: $x := x+5$



syntax tree



Bottom up evalution of S-attributed definitions

- We have seen how to use syntax directed definitions to specify translations, we can begin to study how to implement translators for them.
- Hence a translator is built.
- The task of building translator for any arbitrary syntax directed defination is very dificult
- To overcome this task there are large classes of syntax directed defination for which it is easy to construct translators.
  (1) using S-attributed defenation
  (2) Synthesized attributes can be evaluating using the bottomup parser
  (3) Stack is use to keep track of values of synthesized attributes associated with the glammar symbol on its stack.

Synthesized Attributes on the parser stack

1. A translator for S-attributed definition is implemented using LR-parser Generator.
2. Bottomup method is used to parse the input string
3. Parser stack is used to hold the values of synthesized attribute.

- The stack is implemented as pair of state and value. Each state entry is the pointer to the LR(1) parsing table. There is no need to store glammar symbol implicitl in parser stack at the state entry. for ease of understanding we will refere the state by unique glammar symbol that has been placed in the parser st

. Hence parser stack can be denoted as stack [i]. Stack [i] is combination of State[i] and value[i]. Ex:- Production X→ABC the stack can be as shown in

	State	value
top-2	A	A.a
top-1	B	B.b
top →	C	C.c

Befor reduction

→

State	value	
X	X.x	← top.

After reduction

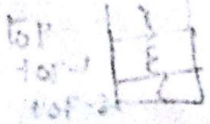
The top symbol on the stack is pointed by pointer top.

Production Rule	Semantic action
X→ABC	X.X = f (A.a, B.b, C.c)

Ex:- construct SDD and generate Code fragment (translator) using S-attributed defination and Parse i/p string 3 × 5 + 4n

Sol:
L → En
E → E+T
E → T
T → T *F
T → F
F → (E)
F → digit

Production	Code fragment (translator)
L→En	print (val [top])
E→E+T	value [top] := val [top-2] + val [to
E→T	
T→T×F	val [top] := val [top-2] × val [to
T→F	
F→(E)	val [top] := val [top-1]
F→digit	



Translation Schemes

- A translation scheme is a context-free grammar in which:
  - attributes are associated with the grammar symbols and
  - semantic actions enclosed between braces {} are inserted within the right sides of productions.
- Ex: $A \to \{ \ldots \}\ X\ \{ \ldots \}\ Y\ \{ \ldots \}$

  Semantic Actions

- When designing a translation scheme, some restrictions should be observed to ensure that an attribute value is available when a semantic action refers to that attribute.
- These restrictions (motivated by L-attributed definitions) ensure that a semantic action does not refer to an attribute that has not yet computed.
- In translation schemes, we use *semantic action* terminology instead of *semantic rule* terminology used in syntax-directed definitions.

The position of the semantic action on the right side indicates when that semantic action will be evaluated.

Translation Schemes for S-attributed Definitions

- If our syntax-directed definition is S-attributed, the construction of the corresponding translation scheme will be simple.
- Each associated semantic rule in a S-attributed syntax-directed definition will be inserted as a semantic action into the end of the right side of the associated production.

Production	Semantic Rule	
$E \to E_1 + T$	$E.val = E_1.val + T.val$	➾ a production of a syntax directed definition

⌄

$E \to E_1 + T\ \{ E.val = E_1.val + T.val \}$ ➾ the production of the corresponding translation scheme

Ex:

- A simple translation scheme that converts infix expressions to the corresponding postfix expressions.

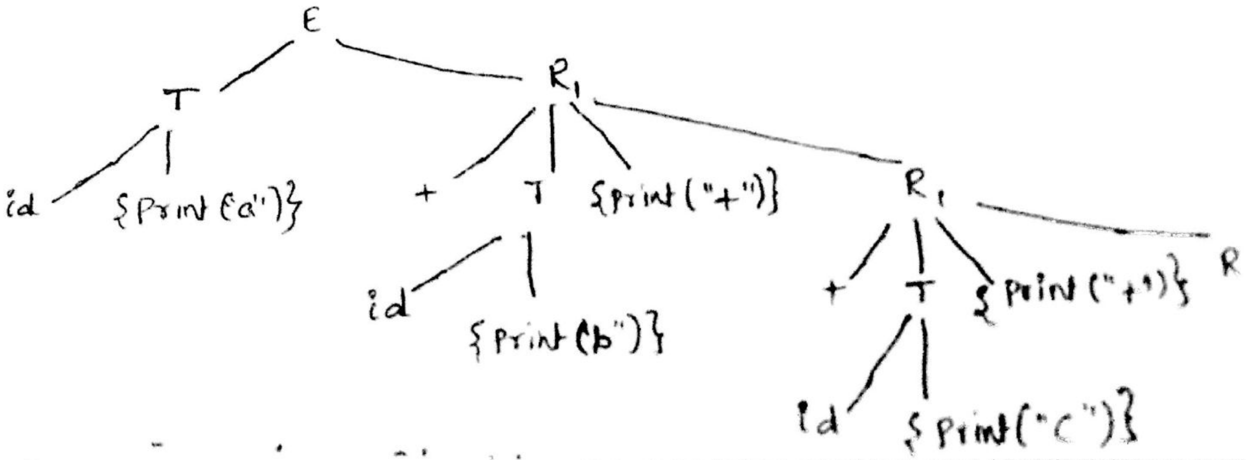
  $E \to T\ R_1$

  $R_1 \to +\ T\ \{ print("+") \}\ R_1$

  $R_1 \to \varepsilon$

  $T \to id\ \{ print(id.name) \}$

  Infix expression is a+b+c ➾ postfix expression is ab+c+



Inherited Attributes in Translation Schemes

- If a translation scheme has to contain both synthesized and inherited attributes, we have to observe the following rules:
  1. An inherited attribute of a symbol on the right side of a production must be computed in a semantic action before that symbol.
  2. A semantic action must not refer to a synthesized attribute of a symbol to the right of that semantic action.
  3. A synthesized attribute for the non-terminal on the left can only be computed after all attributes it references have been computed (we normally put this semantic action at the end of the right side of the production).
  4. With a L-attributed syntax-directed definition, it is always possible to construct a corresponding translation scheme which satisfies these three conditions (This may not be possible for a general syntax-directed translation).

Top-Down Translation

- We will look at the implementation of L-attributed definitions during predictive parsing.
- Instead of the syntax-directed translations, we will work with translation schemes.
- We will see how to evaluate inherited attributes (in L-attributed definitions) during recursive predictive parsing.
- We will also look at what happens to attributes during the left-recursion elimination in the left-recursive grammars.

A Translation Scheme with Inherited Attributes

D → T id { addtype(id.entry,T.type), L.in = T.type } L

T → int { T.type = integer }

T → real { T.type = real }

L → id { addtype(id.entry,L.in), L_1.in = L.in } L_1

L → ε

- This is a translation scheme for an L-attributed definitions.

Eliminating Left Recursion from Translation Scheme

- A translation scheme with a left recursive grammar.

E → E_1 + T { E.val = E_1.val + T.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_1.val * F.val }

T → F { T.val = F.val }

F → (E) { F.val = E.val }

F → digit { F.val = digit.lexval }

T → (E) {T.val = E.val}

T → digit {T.val = digit.lexval}

- When we eliminate the left recursion from the grammar (to get a suitable grammar for the top-down parsing) we also have to change semantic actions.

Intermediate Code : The task of compiler is to convert the Source Program into Machine Program. but it is not always possible to generate such a M/C code directly in one pass. Then compiler generate an easy to represent form of source language which is called intermediate language.

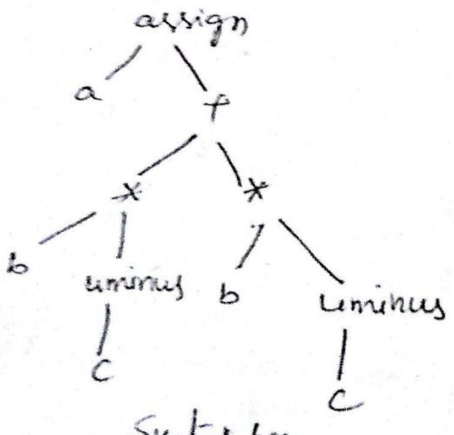
Intermediate forms of Source Program

Mainly three types of Intermediate Code representation

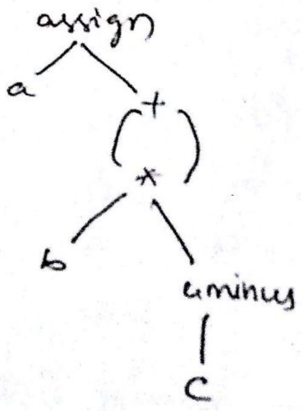
1. Abstract Syntax tree
2. Polish notation.
3. Three address code.

1. **Abstract Syntax tree** :: Hierarichal structure is represented by syntax trees. DAG is very much similar to syntax trees but they are more compact form because Common Subexpression are identified

Ex:- $a := b * -c + b * -c$



Syntax tree



DAG

130350
400665
14/01/5

2. **Polish notation**: In this representation, the operator can be easily associated with corresponding operands.
it also known as prefix notation

$$(a+b)*(c-d) \rightarrow * + ab - cd.$$

3. **Three address code** :- It is used to represent any statement.

General form of three address code representation is

$a := b \text{ op } c$.

where a, b, c are operands that can be names, constants and compiler generated temporary names. where op is operator

Ex: $x + y \times z$

$t_1 := y \times z$

$t_2 := x + t_1$

Implementation of three address code:

- It is an abstract form of intermediate code that can be implemented as a record with the fields for the operator and operands. Three such representations are quadruples triples and indirect triples.

Quadruples: It is a record structure with four fields op which we call op, arg1, arg2 and result.

I/p : $a := b \times -c + b \times -c$

$t_1 := -c$

$t_2 := b \times t_1$

$t_3 := -c$

$t_4 := b \times t_3$

$t_5 := t_2 + t_4$

$a := t_5$

three address code

	op	arg1	arg2	result
(0)	uminus	c		t_1
(1)	*	b	t_1	t_2
(2)	uminus	c		t_3
(3)	*	b	t_3	t_4
(4)	+	t_2	t_4	t_5
(5)	:=	t_5		a

Triples: To avoid entering temporary names into the symbol table we might refer to temporary value by the position of the statement that computes it. It can be represented by record with three fields op, arg1, arg2.

	op	arg1	arg2
0	uminus	c	
1	*	b	0
2	uminus	c	
3	*	b	2
4	+	1	3
5	Assign	a	4

There are pointer. By using pointer we can access directly the symbol table entry

Indirect triples:- Indirect triples representation the listin of triples is beendone and listing pointer are used instead of using statements.

	Statement
0	14
1	15
2	16
3	17
4	18
5	19

	op	arg1	arg2
14	uminus	c	
15	*	b	(14)
16	uminus	c	
17	*	b	(16)
18	+	15	(17)
19	assign	a	18

Types of three address Statements.

The form of three address code is very much similar to assembly language. Here some commonly used three address code for typical language construct

	Language construct	Intermediate Code form	meaninage
1.	Assignment Statement	X:=YOPZ	- Binary operation
2.	Assignment instruction	X:=OPY	- unary operation
3.	Copy statement	X:=Y	- Y assigned to x
4.	unconditional Jump	goto L	- goto label L
5	Conditional Jump	if x Relop Y goto L	- If X Relop y th execute goto L

6) Procedure calls	Parameter x_1 Parameter x_2 ⋮ Param x_n call P, n return y	- Procedure $P(x_1, x_2 \ldots x_n)$. n indicate no of actual parameter in call P, n
7) index Statement (or) Array	x := y[i]	- the value at ith index of array y is assigned to x
"	x[i] := y	- The value of identifier y is assigned at the index i of the array x.
8) Address and pointer assignments	x := &y	- x will be address of y
	x := *y	- y is pointer assigned to x

Conversion of Popular programing language Construct into intermediate Code form:

In this we will learn how to write an intermediate code for various programing constructs such as assignment statement, Boolean expression and so on..

Programing Construct such as

(1) Declarations
(2) Assignment Statements
3. Arrays
4. Boolean expression
5. case statement
6. Procedure calls

(1) **Declarations** :- In the declarative statements the data items along with their datatypes are declared.

S → D	{ offset := 0 }
D → id : T	{ enter_tab (id.name, T.type, offset); offset := offset + T.width
T → integer	{ T.type := integer ; T.width := 4 }
T → real	{ T.type := real; width := 8)
T - array[num] of T_1	{ T.type := array(num.val, T_1.type) T.width := num.val × T_1.width }
T → *T_1	{ T.type := Pointer (T.type) T.width := 4 }

- Initially, the value of offset is set to zero. The computation of offset can be done by using formula

  offset = offset + width.

- Here T.type and T.width are synthasized attributes.
- The rule D→id:T is a declarative statement for id declarati
- The enter-tab is a function used for creating the symbol table entry for identifier along with its type and offset.
- The width of array is obtained by multiplying the width of each element by number of elements in the array.
- The width of pointer type is supposed to be 4.

Assignment Statements :-

- Assignment statements mainly deals with the expressions. The expressions can be of type integer, real, array..
- Here we see how to write the syntax directed translation scheme for generation of three address code for assignme statement containing arithmetic expression.

Ex:-

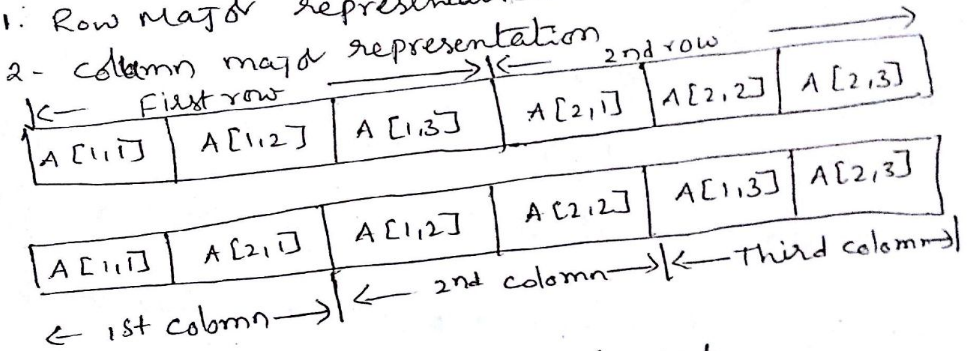
Production	Semantic action
S→id:=E	{ id_entry:= lookup(id.name); if id.entry ≠ nil then emit (id entry':=' E.place) else error; }
E→E₁+E₂	{ E.place := newtemp; emit (E.place':=' E₁.place '+' E₂.place) }
E→E₁*E2	{ E.place := newtemp; emit (E.place ':=' E₁.plac * E₂.place) }
E→−E₁	{ E.place:= newtemp; emit (E.place':=' 'uminus' E₁.place }
E→(E₁)	{ E.place:= E₁.place }
E→id	{ id.entry:= lookup(id.name); if id.entry ≠ nil then emit (id.entry ':=' E.place) else error.

(i) **lookup(id.name)**:- it checks if there is an entry for this occurence of the name in symbol table. If so a pointer to the entry is returned otherwise, lookup returns nil to indicate that no entry was found

(ii) **emit**:- means to emit three address statements to an output file rather than building up code attributes for non terminals.

Arrays:- Array is a collection of contiguous storage of elements. For accessing any element of an array we need its address.

- For statically declared array it is possible to compute the relative address of each element.
- There are two representation of arrays
  1. Row major representation
  2. column major representation



- To compute address of any element

Let base is the address at a[ ] and w is the width of the element then to compute ith address of a[ ]

$\boxed{base + (i - low) \times w}$ low is lower bound on subscript.

Translation scheme for addressing array elements

```
1) S → L := E        { if L.offset = null then
                         emit (L.place ':=' E.place);
                       else
                         emit (L.place '[' L.offset ']' ":=' E.place
                     }
```

We generate normal assignment if L is simple name otherwise indexed assignment into the location denoted by L otherwise.
~~2) E → E1 + E2~~

```
(2) E → E1 + E2      { E.place := newtemp;
                       emit (E.place ':=' E1.place '+' E2.place) }
```

```
3)  E -> (E1)               { E.place := E1.place }

4)  E -> L                  { if L.offset = null then
                                E.place := L.place
                              else begin
                                E.place := newtemp
                                emit (E.place ':=' L.place '[' L.offset ']'
                                end }

5)  L -> Elist ]            { L.place := newtemp;
                              L.offset := newtemp;
                              emit (L.place ':=' c (Elist.array));
                              emit (L.offset ':=' Elist.place '*' width(Elist.arr
                                                                             }

6)  L -> id                 { L.place := id.place;
                              L.offset := NULL
                            }

(7) Elist -> Elist1 , E     { t := newtemp()
                              dim := Elist1.ndim + 1
                              emit (t ':=' t '+' E.place);
                              Elist.array := Elist1.array;
                              Elist.place := t
                              Elist.ndim := m };

(8) Elist -> id [ E         { Elist.array := id.place
                              Elist.place := E.place;
                              Elist.ndim := 1 }
```

Boolean Expressions :-

Two Primary purposes

1. They are used to Compute logical values
2. Conditional Expressions in statement that alt the flow Control (If then, While do)

Methods of translating Boolean expressions

1. Numerical representation
2. Flow of control Statements

Numerical representation.

The translation scheme for Boolean expression having numerical representation is given bellow

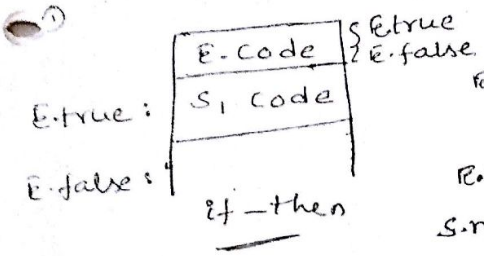
$E \rightarrow E_1$ OR E_2	{ E.place := newtemp(); emit (E.place ':=' E_1.place 'OR' E_2.place)
$E \rightarrow E_1$ and E_2	{ E.place := new temp ; emit (E.place ':=' E_1.place 'and' E_2.place
$E \rightarrow$ not E_1	{ E.place := newtemp; emit (E.place ':=' 'not' E_1.place) }
$E \rightarrow (E_1)$	{ E.place := E_1.place }
$E \rightarrow id_1$ relop id_2	{ E.place := newtemp; emit ('if' id_1.place relop.op id_2.place 'goto' nextstate +3); emit (E.place ':=' '0') emit ('goto' next state +2); emit (E.place ':=' '1')
$E \rightarrow$ true	{ E.place := newtemp emit (E.place ':=' '1') }
$E \rightarrow$ FALSE	{ E.place := newtemp() emit(E.place ':=' '0') }

Flow of Control Statements

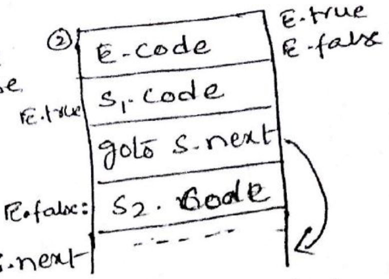
- it can translation of Boolean expression into three address code. The control statements are if-then-else and while-do.

S → if E then S_1
| if E then S_1 else S_2
| while E do S_1

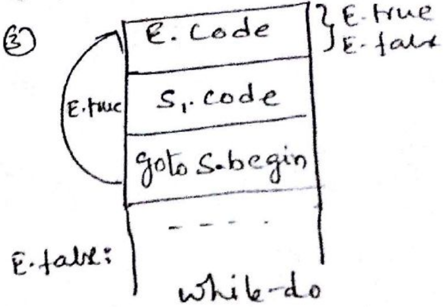
- To generate new symbolic label the function new_label() is used. with the expression E.true and E-false are the labels associated.
- S.Code and E-Code is for generating three address code.

①


if-then

②


if-then-else

③


while-do

S → if E then S_1

E.true := new label ;
E.false := S.next ;
S_1.next := S.next ;
S.code := E.Code || gen_code(E.true':'|| S_1.code

S → If E then S_1 else S_2

E.true := newlabel;
E.false := new label;
S_1.next := S.next;
S_2.next := S.next;
S.Code := E.Code || gen_code(E.true ':') ||
S_1.code || gen_code('goto', S.next) ||
$\overset{code}{gen}$(E.false ':') || S_2.code.

S → ~~If~~ while E do S_1

S.begin := newlabel
E.true := newlabel
E-false := ~~newlabel~~ S.next
S_1.next := S.begin
S.code := gen_code(S.begin ':' || E.code ||
gen_code (E.true ':') || S_1.code || gen_code('goto',

function gen_code is used to evaluate non quoted arguments it and to concatenate complete string

[illegible] three