Unit - 4

Semantic Analysis (C. VENKATAIAH CHOWDARY & 138H1A0516 III/IV CSE)

-) Semantic Analysis is third phase of compiler.

-) Semantic Analysis checks for semantic errors.

-) It identify the operators & operands in expressions & stmnts.

-) It performs type checking.

Intermediate forms of source programs:-

-) Mainly three types of intermediate code representations:

1) Abstract syntax tree.
2) polish notation.
3) 3-address code.

3-address code:-

a = b
a = opb
a = bopc

-) General form of 3-address code is a = bopc

-) where a, b, c are the operands i.e names, constants, compiler generated temporaries.

-) and op represents operator

-) only single operation at right hand side of the expression is allowed at a time.

-) For the expression a = b+c+d the three address code is

$t_1 = b+c$

$t_2 = t_1+d$

$a = t_2$

*) t_1, t_2 are the temporary names generated by the compiler

-) Three address code implementations:-

1) Quadruple
2) Triple
3) Indirect triple

Quadruple:- A quadruple is a record structure with four fields i.e OP, arg1, arg2, result

	OP	arg1	arg2	result
(0)				

-) For the stmnt a = b*-c+b*-c

Three address code is:

$t_1 = -c$

$t_2 = b * t_1$

$t_3 = -c$

$t_4 = b * t_3$

$t_5 = t_2 + t_4$

$a = t_5$

Quadruple.

	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:

=) A triple is a record structure with three fields i.e [op, arg1, arg2]

=) The fields arg1, arg2 are either pointers to the symbol table (or) pointers into the triple structure.

	op	arg1	arg2
(0)			

	op	arg1	arg2
(0)	uminus	c	
(1)	*	b	(0)
(2)	uminus	c	
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	=	a	(4)

(0) => - c
(1) => b*-c
(2) => - c
(3) => b*-c
(4) => b*-c+b*-c
(5) => a = b*-c+b*-c

Indirect-triples:-

=) Listing pointers to triples rather than the listing the triples themselves are called as indirect-triples

	Statement
(0)	(10)
(1)	(11)
(2)	(12)
(3)	(13)
(4)	(14)
(5)	(15)

	op	arg1	arg2
(10)	uminus	c	
(11)	*	b	(10)
(12)	uminus	c	
(13)	*	b	(12)
(14)	+	(11)	(13)
(15)	=	a	(14)

=) For the statement a*-(b+c)

$t_1 = b+c$
$t_2 = -t_1$
$t_3 = a * t_2$

Quadruple:

	(op)	arg1	arg2	result
(0)	+	b	c	t_1
(1)	uminus	t_1		t_2
(2)	*	a	t_2	t_3

Triple:

	(op)	arg1	arg2
(0)	+	b	c
(1)	uminus	(0)	
(2)	*	a	(1)

(0) :) b+c
(1) =) $-t_1$
(2) :) $a * t_2$

Indirect triple:

	Stmnt
(0)	(10)
(1)	(11)
(2)	(12)

	op	arg1	arg2
(10)	+	b	c
(11)	uminus	(10)	
(12)	*	a	(11)

:) Consider the stmt

$-(a*b)+(c+d)-(a+b+c+d)$

$t_1 = a*b$
$t_2 = -t_1$
$t_3 = c+d$
$t_4 = t_2+t_3$
$t_5 = a+b$
$t_6 = t_5+t_3$
$t_7 = t_4-t_6$

Quadruple:

	op	arg1	arg2	result
(0)	*	a	b	t_1
(1)	uminus	t_1		t_2
(2)	+	c	d	t_3
(3)	+	t_2	t_3	t_4
(4)	+	a	b	t_5
(5)	+	t_5	t_3	t_6
(6)	uminus	t_4	t_6	t_7

Triple:

	OP	arg1	arg2
(0)	*	a	b
(1)	uminus	(0)	
(2)	+	c	d
(3)	+	(1)	(2)
(4)	+	a	b
(5)	+	(4)	(2)
(6)	uminus	(3)	(5)

Indirect triple:

	Stmnt
(0)	(10)
(1)	(11)
(2)	(12)
(3)	(13)
(4)	(14)
(5)	(15)
(6)	(16)

	OP	arg1	arg2
(10)	*	a	b
(11)	uminus	(10)	
(12)	+	c	d
(13)	+	(11)	(12)
(14)	+	a	b
(15)	+	(14)	(12)
(16)	uminus	(13)	(15)

Abstract Syntax tree:-

- The natural hierarchical structure of a program is represented by syntax trees.
- DAG (Directed acyclic graph) is very much similar to syntax trees.
- A DAG is a directed graph with no cycles which gives a picture of how the value computed by each stmnt in a basic block.
- A DAG for an expression identifies the common subexpressions in the expression.

Eg: (a+b)*c + (a+b) => the common subexpression is (a+b). this can be easily identify using DAG

- Like a syntax tree, DAG has a node for every subexpression of the given expression. An interior node represents an operator and its children represents operands
- Difference between DAG and Syntax tree is, in DAG the node which is representing a common subexpression has more than one parent, where as in the syntax tree the common subexpression would be represented as a duplicate subtree.
- Construct parse tree and DAG for the expression x = -a*b + -a*b

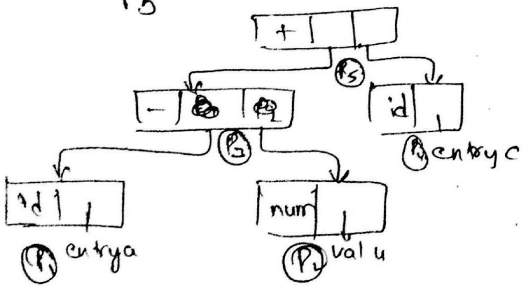
*) mkleaf(num, val):- It creates an number node num and a field containing val, the value of the number.

1) To construct a Syntax tree for expression a-4+c. the sequence of function calls are as shown below

1) P_1, P_2 -- P_5 are the pointers to nodes.

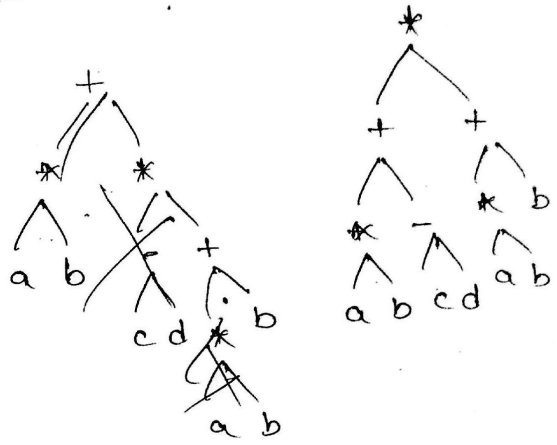
2) Entry a and entry c are pointers to the Symbol table for identifiers a & c respectively

```
P1 = mkleaf(id, a);
P2 = mkleaf(num, 4);
P3 = mknode(-, P1, P2);
P4 = mkleaf(id, c);
P5 = mknode(+, P3, P4);
```

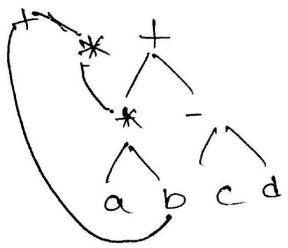


*) Construct syntax tree & DAG for the expression (a*b)+(c-d)*(a*b)+*b

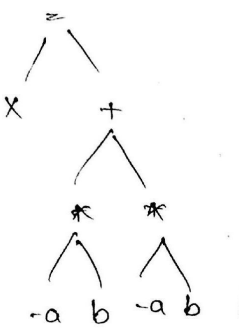
Syntax tree:



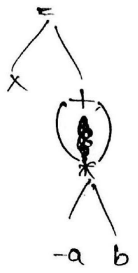
DAG



Syntax tree

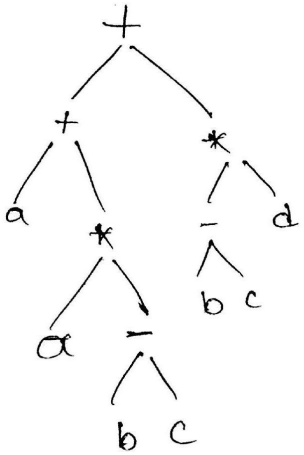


DAG

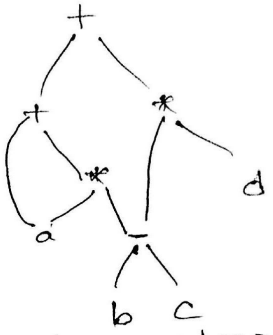


=) Construct Syntax tree & DAG for the expression a+a*(b-c)+(b-c)*d.

Syntax tree.



DAG :-



=) Here the problem subexpressions are a & b-c

=) Here the nodes a has two parents i.e + & * similarly (b-c) node has * as its parents

Syntax trees for expressions:-

=) The following functions are used to create the nodes of syntax trees for given expression.

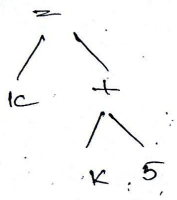
1) mknode (op, left, right): It creates an operator node (op) and two fields containing pointers to left and right operands.

2) mkleaf (id, entry):- It creates an identifier node id and field containing entry, which is a pointer to symbol table entry for the identifier.

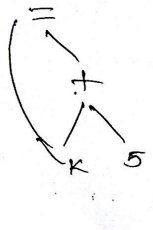
=) Construct DAG & Syntax tree for

K = K + 5

Syntax tree



DAG



Polish notation:-

=) The polish notation is also called as prefix notation in which the operator occurs first and then operands are arranged.

(=) Reverse of polish notation is postfix notation)

=) Eg: Consider the expression

(a+b) * (c-d)

(+ab) * (-cd)

*(+ab-cd

SDT Schemes (Syntax Direct Translation):-

=) Syntax SDT stands for Syntax Direct Translation.

(7)

=) Syntax Direct Definition (SDD):-

=) A syntax direct definition is a generalization of a context free grammar in which each grammar symbol has an associated set of attributes and each production is associated with a set of semantic rules.

=) The attributes are of two types They are:

=) Synthesized attribute

=) Inherited attribute

=) Form of SDD (syntax Directed definition):-

=) In a SDD each grammar production $A \rightarrow \alpha$ has set of semantic rules associated with it, which is of the form $b = f(c_1, c_2, \ldots c_n)$ where f is function and b may be either Synthesized attribute (or) Inherited attribute.

Synthesized attribute:-

=) A Synthesized attribute (A) and $c_1, c_2, \ldots c_n$ are attributes belonging to the grammar symbols of the production

=) The value of Synthesized attribute at a node is computed

from the values of attributes at the children of that node in the parse tree.

Inherited attribute: An inherited attribute of one of the grammar symbols on right hand side of the production and $c_1, c_2 \dots c_n$ are attributes belonging to the grammar symbols of the production

- The value of an inherited attribute is computed from the values of attributes at the siblings and parent of that node.

- The grammar and set of semantic rules constitute the syntax directed definition

Eg: The SDD from a desk calculator program as shown.

Production	Semantic Rule
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 = digital.lexval

- A definition associates an integer valued Synthesized attribute called val with each of the non-terminals (E, T, F)

- The Semantic rule computes the valued attribute val for the non-terminal on left side from the values of val for the non-terminals on right side.

- The token digit has a synthesized attribute lexval whose value is supplied by lexical analyser.

- The first production L → En, prints the value expression generated by E.
  - E is the start symbol of the grammar.
  - The second production returns the value of expression E by performing addition with E.val & T.val
  - Similarly all the remaining productions are assigned with the value of corresponding non-terminal.
  - The last production F → digit. lexval gets the value of the token from the lexical analyser and assigns the value to the nonterminal F

Eg 2) The SDD for infix to postfix translation

Production	Semantic rule.
expr → expr + term	expr.t = expr.t ‖ term.t ‖ '+'
expr → expr - term	expr.t = expr.t ‖ term.t ‖ '-'
expr → term	expr.t = term.t
term → 0	term.t = '0'
term → 1	term.t = '1'
⋮	⋮
term → 9	term.t = '9'

⇒ The postfix notation for an expression E can be defined as

⇒ If E is variable or constant, then the postfix notation for E is E itself.

⇒ If E is an expression of the form E_1 op E_2, where op is an operator then the postfix notation for E is $E_1 E_2$ op

⇒ If E is an expression of the form (E_1), then the postfix notation for E_1 is also E itself. (OR) E.val.

⇒ In the above SDD each non-terminal is associated with an attribute called 't'

⇒ The operator '‖' in Semantic rules represents 'concatination'

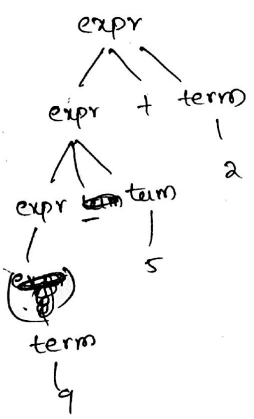
⇒ Construct the parse tree and annotated Parse tree for the Expression 9-5+2 (9)

Annotated Parse tree:-

⇒ The parse tree showing the values of attributes at each node is called an annotated Parsetree.

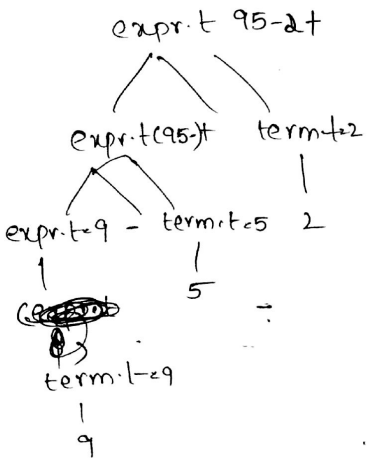
Parse tree

9-5+2



Annotated parse tree

9-5+2



Synthesized attribute:

⇒ An attribute is said to be synthesized if its value at a node in parse tree is determined from attribute values of the children at that node.

⇒ An SDD that uses synthesized attributes is said to be an S-attribute

Explanation :- Following steps are to be followed to compute S-attribute definition.

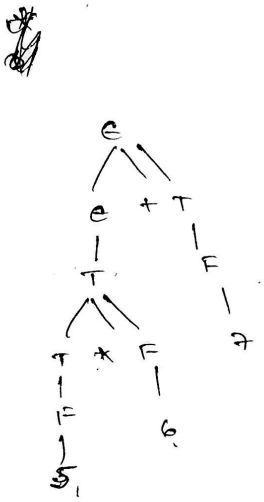
1) write the SDD using the appropriate semantic actions for corresponding productions in the grammar
2) Annotated parse tree is ~~computed~~ generated and attribute values are computed
3) The computation is done in "bottom up manner".
4) the value obtained at the root node is supposed to be the final o/p.

Eg: Construct parse tree and Annoated Parse tree for the i/p string 5*6+7; using S-attribute

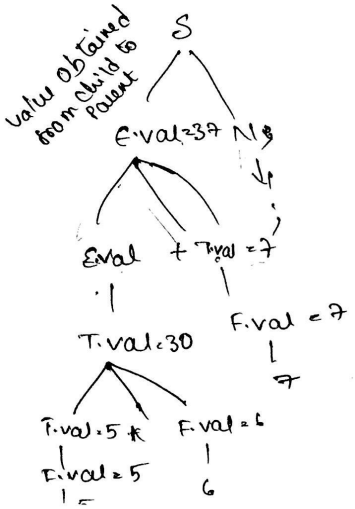
For

Production	Semantic action
S→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→digit	F.val → digit.lexval
N→;	can be ignored by lexical analyser as ';' is terminating symbol

Parse tree :-



Annoated parse tree :-



Inherited attributes:-

=) The value of inherited attribute at a node in a parse tree defined using the attribute values at the "parent" or "Siblings"

-) consider an example and let us compute the inherited attributes

-) construct the annotated parse tree for the string int a, b, c for the computation of inherited attribute

=) For the string int a, b, c we have to distribute the datatype int to all the identifiers a, b and c such that a becomes integer, b becomes integers and c becomes integer

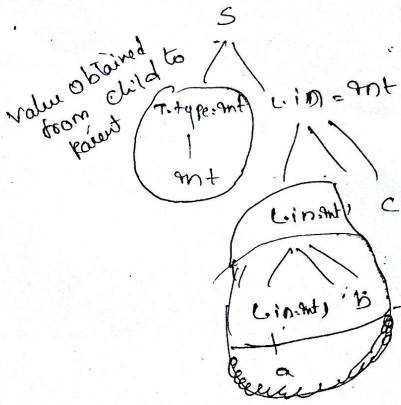
=) Following steps are to be followed:

1) construct the SDD using semantic actions

=) Annotate the parse tree with inherited attributes by processing in top down manner

Production	Semantic Action
S → TL	L.in = T.type
T → int	T.type = int
L → L, id	L.in = L.in, addtype (id.entry, L.in)
L → id	L.in = addtype (id.entry, L.in)

Annotated Parse tree :-



Dependancie graph:

=) The directed graph that represents the interdependencies between synthesized and inherited attributes at nodes in the parse tree is called dependancie graph.

=) For the rule $x \to yz$ the semantic action is given by $x.a = f(y.y, z.z)$ then the synthesized attribute of $x.a$ and $x.a$ depends on attributes $y.y$ and $z.z$

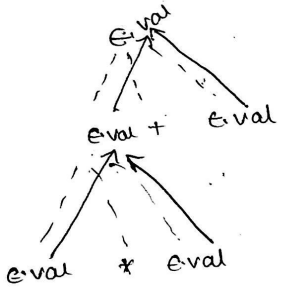
Eg: Design the dependencie graph for the following grammar.

$$E \to E_1 + E_2$$
$$E \to E_1 * E_2$$

Sol The semantic rules for the above grammar is as given below.

Production	Semantic rules
$e \to e_1 + e_2$	$e.val = e_1.val + e_2.val$
$e \to e_1 * e_2$	$e.val = e_1.val * e_2.val$

The dependencie graph is as shown below



1) The synthesized attributes can be represented by 'val'

2) The dependencies among the nodes is given by solid arrows. The arrows from e.val + e.val calculates the value of e.val. Hence the e.val is synthesized attributes

3) we have represented parse tree using dotted lines

4) Design the dependencie graph for the following grammar.

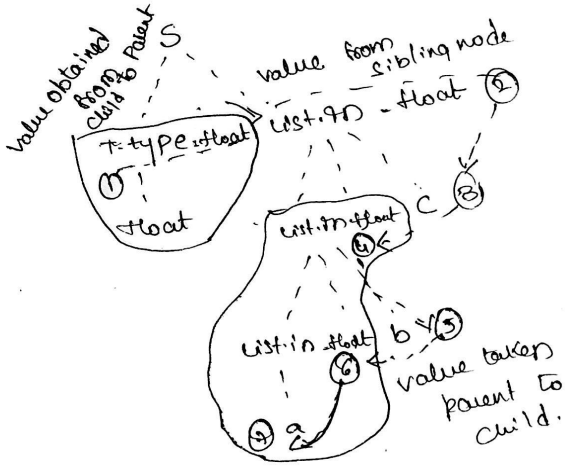
$S \to TL$
$T \to int$
$T \to float$
$L \to L, id$
$L \to id$

1) the dotted lines is for representing parse tree.

2) The semantic rules for the above grammar is as given below

Production	Semantic rule
$S \to TL$	L.in = T.type
$T \to int$	T.type = int
$T \to float$	T.type = float
$L \to L, id$	L.in = L.in, addtype (id.entry, L.in)
$L \to id$	L.in = addtype (id.entry, L.in)

Dependencie graph for the string float a, b, c

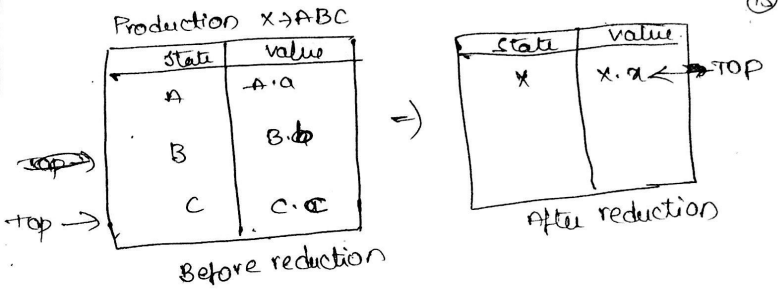


Bottom up evaluation of S-attribute definitions :-

- The SDD that uses ~~syntax~~ Synthesized attributes is said to be an S-attribute definitions.
- Synthesized attributes can be evaluated using the bottom-up parser.
- The purpose of stack is to keep track of values of synthesized attributes associated with grammar symbol on its stack. This stack is commonly known as parser stack.

Synthesized attributes on the parser stack:

- A translator for S-attributed definition is implemented using a LR parser generator.
- A bottom-up method is used to parse the i/p string.
- The parser stack is used to hold the values of synthesized attributes.
- The stack is implemented as a pair of state and value.
- Each state entry is the pointer to LR Parsing table.
- We will refer the state by unique grammar symbol that is being placed in the parser stack. Hence the parser stack can be denoted as stack[i]
- A stack[i] is a combination of state[i] and value[i].
- For example for the production rule $X \rightarrow ABC$ the stack can be shown as ~~shown~~ below.



Parser stack

- The top symbol on the stack is pointed by the pointer TOP.
- The semantic action associated with the production rule $X \rightarrow ABC$ is as follows:

Production	Semantic action
$X \rightarrow ABC$	X.x = f(A.a, B.b, C.c)

- Before reduction the states A, B, C can be inserted in the stack along with the values A.a, B.b, C.c
- The top pointer value[top] will point to the value C.c, lly B.b is in ~~value~~ ~~TOP~~ value[top-1] and A.a is in value[top-2]
- After reduction the left hand side symbol X will be placed in the stack along with the value X.x at the top. Hence after reduction value[top] = X.x
- If a symbol has no attribute then the

corresponding entry in the value array will be kept undefined.

Eg: Construct SDD and generate the code translator using S-attributed definition for the following grammar.

S → EN
E → E+T
E → E-T
E → T
T → T*F
T → T/F
T → F
F → (E)
F → digit
F → ; and also evaluate the i/p string 2*3+4; with parser stack using LR parsing method

Sol: SDD for the given grammar is as follows:

Production rule	Semantic rule
S → EN	Print(E.val)
E → E+T	E.val ≥ E.val + T.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 → 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.lexval
F → ;	can be ignored by lexical analyzer as ; is a terminating symbol

(11)

The LR parser table can be generated to evaluate the attributes, the code translator can be generated by using the parser stack.

Production Rule	Code translator
S → EN	Print(value[top])
E → E+T	value[top] = value[top-2] + value[top]
E → E-T	value[top] = value[top-2] - value[top]
E → T	
T → T*F	value[top] = value[top-2] * value[top]
T → T/F	value[top] = value[top-2] / value[top]
T → F	-
F → (E)	value[top] = value[top-1]
F → digit	-
F → ;	-

The sequence of moves made by the parser for i/p string 2*3+4; are as follows.

~~Evaluate the input string~~

I/p string	state	value	Production Rule used
2*3+4;	-	-	
*3+4;	2	2	
*3+4;	F	2	F→ digit
*3+4;	T	2	T→ F
*3+4;	T*	2-	
+4;	T*3	2-3	
+4;	T*F	2-3	F→digit
+4;	T	6	T→T*F
+4;	E	6	E→T
4;	E+	6-	
;	E+4	6-4	
;	E+4	6-4	
;	E+F	6-4	F→ digit
;	E+T	6-4	T→ F
;	E	10	E→E+T
	E;	10-	
	EN	10	
		10	S→EN

L-Attributed definition:-

1) The SDD can be defined as the L-attributed for the production rule $A \Rightarrow X_1, X_2 \ldots X_n$ where the inherited attribute X_K is such that $1 \le K \le n$. (15)

2) The production $A \Rightarrow X_1, X_2 \ldots X_n$ is such that

i) It depends upon the attributes of the symbol $X_1, X_2 \ldots X_{j-1}$ to the left of X_j.

2) It also depends upon the inherited attribute A

3) Because of these two conditions every S-attributed definition is also L-attributed definition.

4) The class of L-attributes can be evaluated in depth first order

Bottom up evaluation of inherited attributes:

1) The inherited attributes can be handled by L-attributed definitions.

2) The bottom parser reduces the right side of the production $A \Rightarrow ABC$ by removing C, B, A of the parser stack.

3) The parser stack is implemented as a combination of state and value.

4) The state[i] is for the grammar symbol A and value[i] holds the Synthesized attribute A.a

Types and Declarations:-

Type Expressions:- Type expression is the type of a language constructs.

=) A type expression is either a basic type or is formed by applying an operator called a type constructor to other type expressions.

=) Some of the type expressions are given below:

=) A basic type is type expression. Eg: boolean, char, int, real e.t.c. A spl type type_error will signal an error during type checking. A basic type void denotes the absence of a value.

=) A type expression has to be named

=) A type constructor applied to a type expressions is a type expression. Some of the constructors are

=) **Arrays:-** If T is a type expression then array(I,T) is a type expression denoting a type of an array with elements of type T and index set I. I is a range of integers for example the type expression for the array with the type integer and index ranging from 1 to 10 is as shown.

array(1..10, int)

=) **Products:-** If T_1 & T_2 are type expressions then their cartesian product $T_1 \times T_2$ is a type expression.

-) Records :- The record type constructor is applied to a tuple formed by field names and field types. The difference b/w a record and a product is that the fields of record have names. For example consider a program

```
Type row = record;
        address : integer;
        lexeme : arr[1..15] of char;
      end;
Variable : arr[1...50] of row;
```

-) This program declares the type name row representing the type expression record ((address x integer) x (lexeme x array(1..15, char)))

-) The variable table is declared to be an array(50) records of this type.

-) Type Equivalence :-

-) The type equivalence can be of two types

-) Structural equivalence
-) named equivalence.

Structural equivalence :- The structural equivalence of type expression, two type expression for structural equivalence, if and only if they are identical. i.e the two expressions should be of same basic type (or) they should be formed by applying the same constructor

-) An algorithm for testing structural equivalence of type expressions formed by type constructors, arrays, products, pointers & functions is given below

1) Function sequiv(s,t) : boolean;
   begin
2) If s and t are the same base type then
3) return true.
4) else if s = array(s1, s2) and t = array(t1, t2) then
5) return = sequiv(s1, t1) and sequiv(s2, t2)
6) else if s = s1 x s2 and t1 x t2 then
7) return sequiv(s1,t1) and sequiv(s2, t2)
8) else if s = pointer(s1) and t = pointer(t1) then
9) return sequiv(s1, t1)
   else
10) else if s = s1 -> s2 and t = t1 -> t2 then
11) return sequiv(s1,t1) and sequiv(s2 t2)
    else
12) return false
    end.

-) The function sequiv(s,t) checks whether the type expressions S & T are type equivalent

-) First it checks whether S & T are same basic type such as boolean, integer, real -- if they are same then it returns specifying expression S & T are type equivalent otherwise it checks the next the type constructors i.e arrays, products & pointers.

Name equivalence

each type name as a distinct type. So two type expressions are name equivalent if and only if they are identical where as the structural equivalence replaces the names by type expressions they defined.

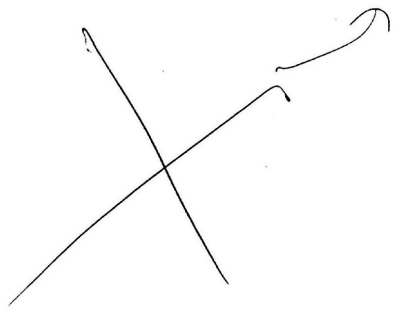
-) Consider a program:

(↑ - pointer symbol)

```
Type link = ↑cell;
Var next : link;
    last : link;
    p    : ↑cell;
    q, r : ↑cell;
```

-) On the above program, the identifier link is declared as the name for the type (Pointer) (↑ cell)

-) For eg if cell is a name of type expression the ~~Pointer(cell)~~ Pointer(cell) is a type expression



Variable	type expression
next	link
last	link
p	Pointer(cell)
q	Pointer(cell)
r	Pointer(cell)

-) when name equivalence is considered the variables next & last have the same type becos they have the same type expression associated with them and also p, q, r have the same type.

-) But p & last are not name equivalent becos they are associated by diff type expressions

Type checking :-

Simple type checker:

A type checker is a translations scheme that synthesizes the type of each expression from the types of its sub-expressions

Consider the grammar

P → D; E

D → D; D | id : T

T → char | integer | array[num] of T | ↑T

E → literal | num | id | E mod E | E[E] | E↑

- It generates programs represented by T, a sequence of declarations represented by D
- The above language consists of two basic types i.e char, integer and third basic type is type_error used to signal errors
- The other type expressions needed are arrays, pointers
- The translation scheme for the above grammar is

$P \rightarrow D;E$	
$D \rightarrow D;D$	
$D \rightarrow id:T$	{addtype (id.entry, T.type)}
$T \rightarrow char$	{T.type := char}
$T \rightarrow integer$	{T.type := integer}
$T \rightarrow array[num]$ of T_1	{T.type := array (1..num.val, T_1.type)}
$T \rightarrow \uparrow T_1$	{T.type := pointer (T_1.type)}

Type checking for expressions:

consider $E \rightarrow literal$	{E.Type = char}
$E \rightarrow num$	{E.Type = integer}

The above Semantic rule specify that the constants represented by the tokens literal and NUM have the type Char & integer respectively

For this stmnt $E \rightarrow id$ {E.type = lookup (id.entry)}

- For this stmnt $E \rightarrow E_1 \bmod E_2$

{E.Type = if E_1.Type = integer and E_2.Type = integer then integer else type_error}

If the expression contains mod operator applied to two sub-expression E_1 & E_2 then E.type) if type integer then E.Type is of type integer otherwise it returns error

Type checking of stmnts:-

- A specific type called as void is assigned for stmnts for which the value does not exists
- If an error is detected it is identified by type_error.
- The below rules checks for the boolean type that is true or false which are called as boolean values.
- If the expression evaluates to anyone of the boolean value then the statement S_1 will be executed otherwise the error will be given.

ii. S → if E then S_1 {S.type = if E.type = boolean then S_1.type else type_error}

iii. S → while E do S_1 {S.type = if E.type = boolean then S_1.type else type_error}

→ In the above rules if any mismatch of type occurs then the type checker produces type_error which reports about error i.e the location of the type of error that has occur.

Type conversions:- Type conversion is the method of converting a variable from its datatype to another datatype depending on the operation and other operands

Eg: consider expression a+b

→ where a is of type real & b is of type integer. Since the two datatypes are different, the compiler has to first convert the datatype of b to real then perform addition operation b/w two operands a & b

→ The type checker is used to insert this conversion operation into the intermediate representation of the source program.

Coercions:-

→ Coercions are implicit type conversions is said to implicit if it is to be done automatically by the compiler.

→ Coercions are limited in many languages it may lead to the loss of information

Eg: → An integer can be converted into real conversion of real to integer the change of accurate value

Type checking rules for coercions from integer to real:-

Production	Semantic rule
E → num	E.type := integer
E → num.num	E.type := real
E → id	E.type := lookup(id.entry)
E → E1 OP E2	E.type := if E1.type = integer and E2.type = integer else if E1.type = integer and E2.type = real then real else if E1.type = real and E2.type = integer then real else if E1.type = real and E2.type = real then real else type_error