Prepared by EBIN P.M (AP, CSE)
IES College of Engineering

MODULE 4

Syntax directed translation:
Syntax directed definitions, Bottom- up evaluation of S-attributed definitions, L- attributed definitions, Top-down translation, Bottom-up evaluation of inherited attributes.
Type Checking:
Type systems, Specification of a simple type checker.

NEED FOR SEMANTIC ANALYSIS

- Semantic analysis is a phase by a compiler that adds semantic information to the parse tree and performs certain checks based on this information.
- It logically follows the parsing phase, in which the parse tree is generated, and logically precedes the code generation phase, in which (intermediate/target) code is generated. (In a compiler implementation, it may be possible to fold different phases into one pass.)
- Typical examples of semantic information that is added and checked is typing information (type checking) and the binding of variables and function names to their definitions (object binding).
- Sometimes also some early code optimization is done in this phase. For this phase the compiler usually maintains *symbol tables* in which it stores what each symbol (variable names, function names, etc.) refers to.

Following things are done in Semantic Analysis:

1. **Disambiguate Overloaded operators** : If an operator is overloaded, one would like to specify the meaning of that particular operator because from one will go into code generation phase next.

2. **Type checking** : The process of verifying and enforcing the constraints of types is called type checking.

- This may occur either at compile-time (a static check) or run-time (` dynamic check).
- Static type checking is a primary task of the semantic analysis carried out by a compiler.
- If type rules are enforced strongly (that is, generally allowing only those automatic type conversions which do not lose information), the process is called strongly typed, if not, weakly typed.

3. **Uniqueness checking** : Whether a variable name is unique or not, in the its scope.

4. **Type coercion** : If some kind of mixing of types is allowed. Done in languages which are not strongly typed. This can be done dynamically as well as statically.

5. **Name Checks** : Check whether any variable has a name which is not allowed. Ex. Name is same as an identifier(Ex. int in java).

- A parser has its own limitations in catching program errors related to semantics, something that is deeper than syntax analysis.
- Typical features of semantic analysis cannot be modeled using context free grammar formalism.
- If one tries to incorporate those features in the definition of a language then that language doesn't remain context free anymore**.**
- These are a couple of examples which tell us that typically what a compiler has to do beyond syntax analysis.
- An identifier **x** can be declared in two separate functions in the program, once of the type int and then of the type char. Hence the same identifier will have to be bound to these two different properties in the two different contexts.

Semantic Errors

We have mentioned some of the semantics errors that the semantic analyzer is expected to recognize:

- Type mismatch
- Undeclared variable
- Reserved identifier misuse.
- Multiple declaration of variable in a scope.
- Accessing an out of scope variable.
- Actual and formal parameter mismatch.

Syntax Directed Translation

The Principle of Syntax Directed Translation states that the meaning of an input sentence is related to its syntactic structure, i.e., to its Parse-Tree. By Syntax Directed Translations we indicate those formalisms for specifying translations for programming language constructs guided by context-free grammars.

– **We associate Attributes to the grammar symbols representing the language constructs.**

– Values for attributes are computed by Semantic Rules associated with grammar productions.

Evaluation of Semantic Rules may:

– Generate Code;

– Insert information into the Symbol Table;

– Perform Semantic Check;

– Issue error messages;

– etc.

- There are two ways to represent the semantic rules associated with grammar symbols.

1. Syntax-Directed Definitions (SDD)
2. Syntax-Directed Translation Schemes (SDT)

Syntax Directed Definitions

Syntax Directed Definitions are a generalization 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.

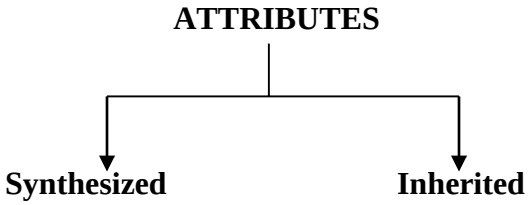
ATTRIBUTE GRAMMAR

- Attributes are properties associated with grammar symbols. Attributes can be numbers, strings, memory locations, data types, etc.
- Attribute grammar is a special form of context-free grammar where some additional information (attributes) are appended to one or more of its non-terminals in order to provide context-sensitive information.
- Attribute grammar is a medium to provide semantics to the context-free grammar and it can help specify the syntax and semantics of a programming language. Attribute grammar (when viewed as a parse-tree) can pass values or information among the nodes of a tree.

Ex: $E \rightarrow E + T$ { E.value = E.value + T.value }

- The right part of the CFG contains the semantic rules that specify how the grammar should be interpreted. Here, the values of non-terminals E and T are added together and the result is copied to the non-terminal E.
- Semantic attributes may be assigned to their values from their domain at the time of parsing and evaluated at the time of assignment or conditions.

- Based on the way the attributes get their values, they can be broadly divided into two categories : synthesized attributes and inherited attributes



1. **Synthesized Attributes:** These are those attributes which get their values from their children nodes i.e. value of synthesized attribute at node is computed from the values of attributes at children nodes in parse tree.

- To illustrate, assume the following production:

EXAMPLE: S -> ABC

S.a= A.a,B.a,C.a

If S is taking values from its child nodes (A, B, C), then it is said to be a synthesized attribute, as the values of ABC are synthesized to S.

Computation of Synthesized Attributes

- Write the SDD using appropriate semantic rules for each production in given grammar.
- The annotated parse tree is generated and attribute values are computed in bottom up manner.
- The value obtained at root node is the final output.

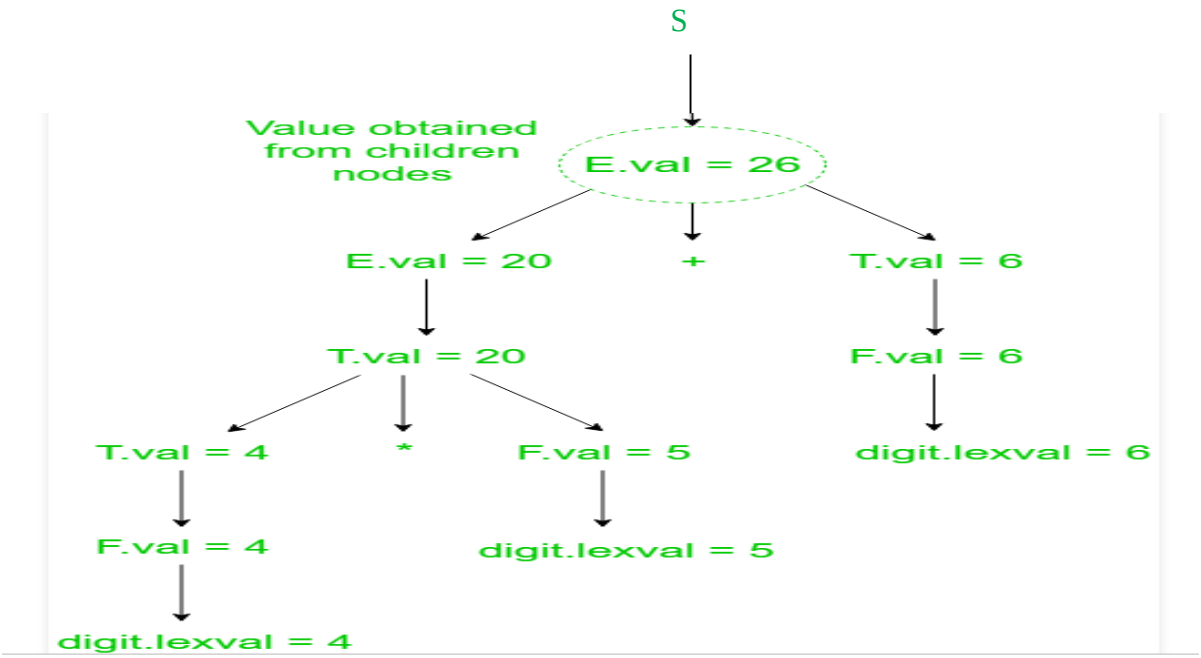
Consider the following grammar:

```
S --> E
E --> E₁ +
T E --> T
T --> T₁ *
F T --> F
F --> digit
```

The SDD for the above grammar can be written as follow

PRODUCTIONS	SEMANTIC RULES
S --> E	Print(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_1.val * F.val
T --> F	T.val = F.val
F --> digit	F.val = digit.lexval

Let us assume an input string **4 * 5 + 6** for computing synthesized attributes. The annotated parse tree for the input string is



- For computation of attributes we start from leftmost bottom node. The rule F –> digit is used to reduce digit to F and the value of digit is obtained from lexical analyzer which becomes value of F i.e. from semantic action F.val = digit.lexval.
- Hence, F.val = 4 and since T is parent node of F so, we get T.val = 4 from semantic action T.val = F.val.
- Then, for T –> T_1 * F production, the corresponding semantic action is T.val = T_1.val * F.val . Hence, T.val = 4 * 5 = 20
- Similarly, combination of E_1.val + T.val becomes E.val i.e. E.val = E_1.val + T.val = 26. Then, the production S –> E is applied to reduce E.val = 26 and semantic action associated with it prints the result E.val . Hence, the output will be 26.

ANNOTATED PARSE TREE

- The parse tree containing the values of attributes at each node for given input string is called annotated or decorated parse tree.

2. Inherited Attributes: These are the attributes which inherit their values from their parent or sibling nodes. i.e. value of inherited attributes are computed by value of parent or sibling nodes.

EXAMPLE:

```
A --> BCD   { C.in = A.in, C.type = B.type }
```

B can get values from A, C and D. C can take values from A, B, and D. Likewise, D can take values from A, B, and C.

Computation of Inherited Attributes

- Construct the SDD using semantic actions.
- The annotated parse tree is generated and attribute values are computed in top down manner.

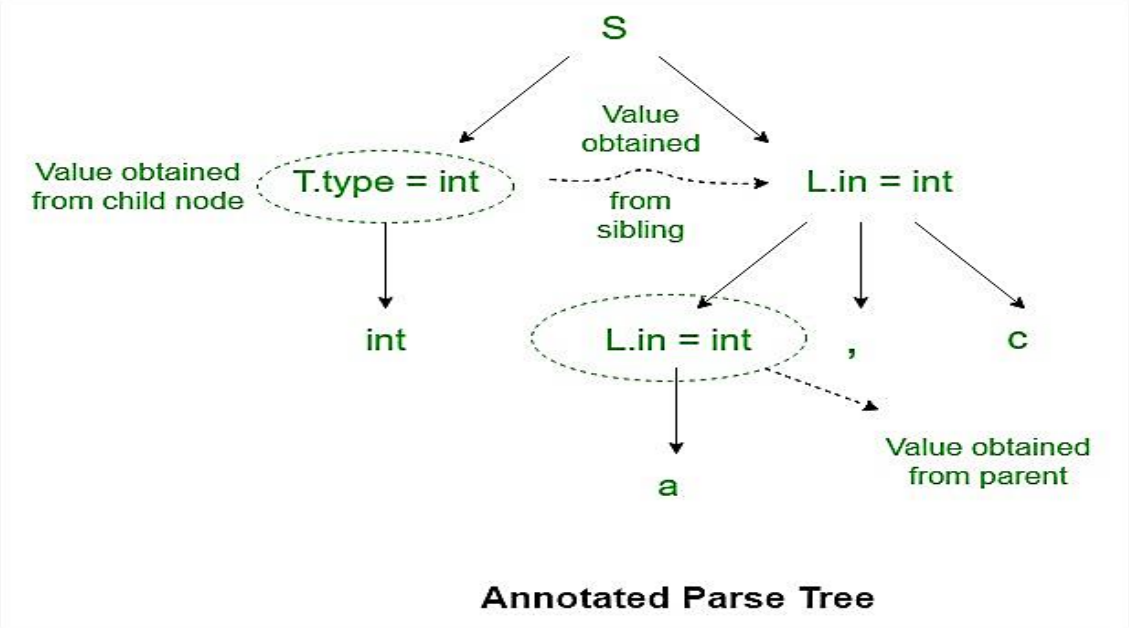
Consider the following grammar:

```
D --> T L
T --> int
T --> float
T --> double
L --> L1, id
L --> id
```

The SDD for the above grammar can be written as follow

PRODUCTIONS	SEMANTIC RULES
D→TL	L.in = =T.type
T --> int	T.type = int
T --> float	T.type = float
T --> double	T.type = double
L --> L_1 , id	L_1.in = L.in Enter_type(id.entry , L.in)
L --> id	Entry_type(id.entry , L.in)

- Let us assume an input string **int a, c** for computing inherited attributes. The annotated parse tree for the input string is



Annotated Parse Tree

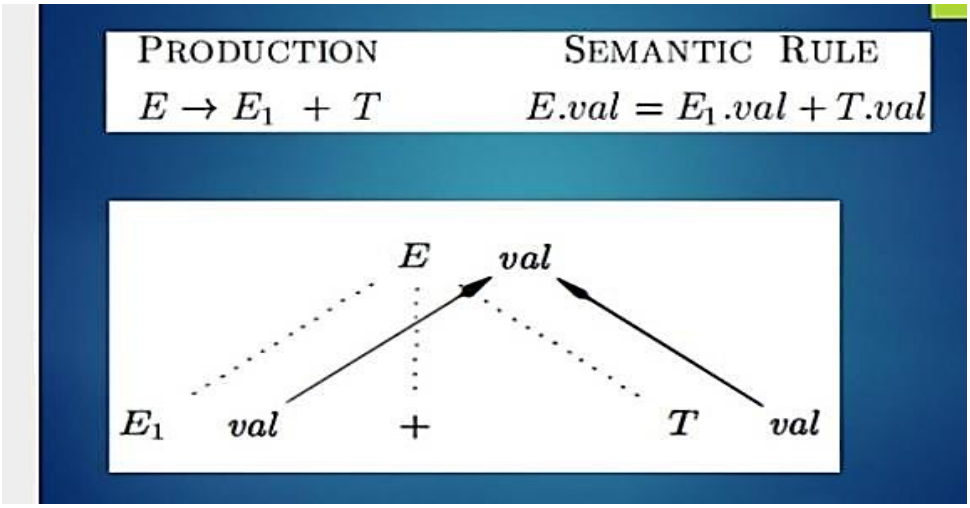
- The value of L nodes is obtained from T.type (sibling) which is basically lexical value obtained as int, float or double.
- Then L node gives type of identifiers a and c. The computation of type is done in top down manner or preorder traversal.
- Using function Enter_type the type of identifiers a and c is inserted in symbol table at corresponding id.entry.

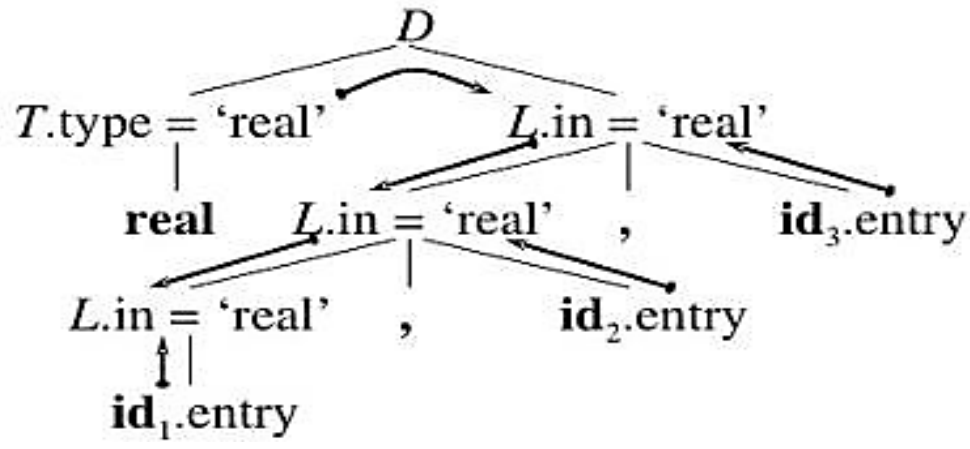
Implementing Syntax Directed Definitions

1. **Dependency Graphs**

- Implementing a Syntax Directed Definition consists primarily in finding an order for the evaluation of attributes
    - Each attribute value must be available when a computation is performed.
- Dependency Graphs are the most general technique used to evaluate syntax directed definitions with both synthesized and inherited attributes.
- Annotated parse tree shows the values of attributes, dependency graph helps to determine how those values are computed

- The interdependencies among the attributes of the various nodes of a parse-tree can be depicted by a directed graph called a dependency graph.
  - **There is a node for each attribute;**
  - **If attribute b depends on an attribute c there is a link from the node for c to the node for b (b ← c).**
- Dependency Rule: If an attribute b depends from an attribute c, then we need to find the semantic rule for c first and then the semantic rule for b.





Dependency graph for declaration statements.

2. Evaluation order

- A dependency graph characterizes the possible order in which we can calculate the attributes at various nodes of the parse tree.
- If there is an edge from node M to N, then the attribute corresponding to M first be evaluated before evaluating N.
- Thus the allowable orders of evaluation are N1,N2…..,Nk such that if there is an edge from Ni toNj then i<j
- Such an ordering embeds a directed graph into a linear order, and is called a **topological sort** of the graph.
- If there is any cycle in the graph ,then there is no topologicalsort.ie, there is no way to evaluate SDD on this parse tree.

TYPES OF SDT'S

1. **S –attributed definition**
2. **L –attributed definition**

S-attributed definition

- S stands for synthesized
- If an SDT uses only synthesized attributes, it is called as S-attributed SDT.

  EXAMPLE:

  A→BC { A.a=B.a,C.a}
- S-attributed SDTs are evaluated in bottom-up parsing, as the values of the parent nodes depend upon the values of the child nodes.
- Semantic actions are placed in rightmost place of RHS.

  A→BCD{ }.
- **Note:** (Also write SDD for desk calculator as example).

L –attributed definition

- L stands for one parse from left to right.

  - **L-Attributed Definitions** contain both synthesized and inherited attributes but do not need to build a dependency graph to evaluate them.
  - **Definition.** A syntax directed definition is *L-Attributed* if each *inherited attribute* of X_j in a production $A \rightarrow X_1 \ldots X_j \ldots X_n$, depends only on:
    1. The attributes of the symbols to the **left** (this is what L in *L-Attributed* stands for) of X_j, i.e., $X_1 X_2 \ldots X_{j-1}$, and
    2. The *inherited* attributes of A.

- Ie, If an SDT uses both synthesized attributes and inherited attributes with a restriction that inherited attribute can inherit values from parent and left siblings only, it is called as L-attributed SDT.

  EXAMPLE:

  A →BCD {B.a=A.a, C.a=B.a}

  C.a=D.a → This is not possible

- Attributes in L-attributed SDTs are evaluated by depth-first and left-to-right parsing manner.
- Semantic actions are placed anywhere in RHS.

  A→{ }BC

  B{ }C

  BC{ }

- **Note:** (Also write SDD for declaration statement as example)

If an attribute is S attributed, it is also L attributed.

Evaluation of L-attributed SDD

- **L-Attributed Definitions are a class of syntax directed definitions whose attributes can always be evaluated by single traversal of the parse-tree.**
- The following procedure evaluate L-Attributed Definitions by mixing PostOrder (synthesized) and PreOrder (inherited) traversal.

Algorithm: L-Eval(n: Node)
Input: Node of an annotated parse-tree.
Output: Attribute evaluation.

```
Begin
    For each child m of n, from left-to-right Do
    Begin
        Evaluate inherited attributes of m;
        L-Eval(m)
    End;
    Evaluate synthesized attributes of n
End.
```

SDD for desk calculator/SDD for evaluation of expressions

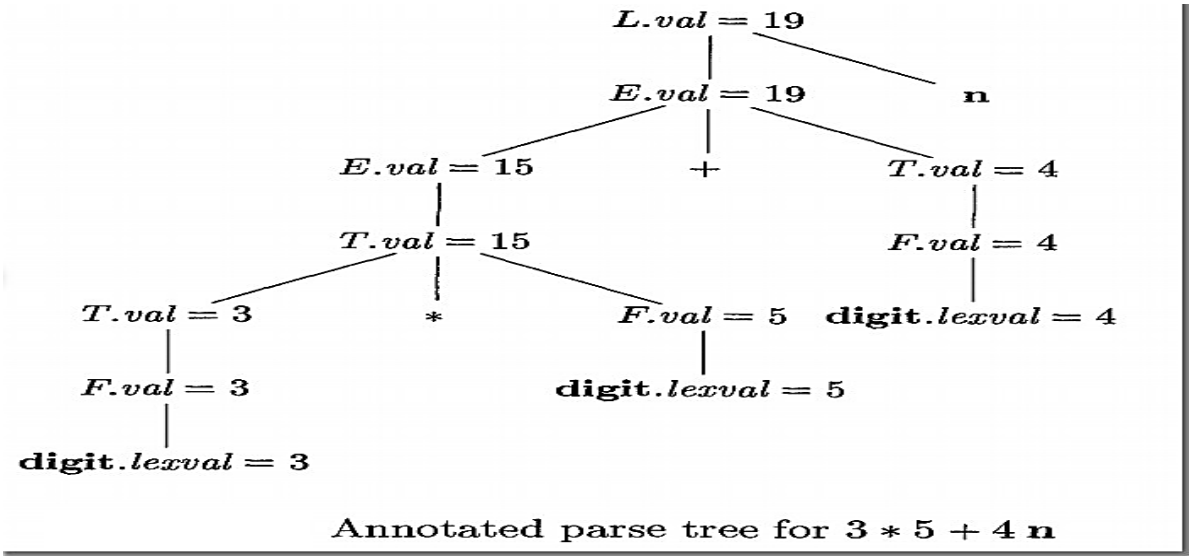
	PRODUCTION	SEMANTIC RULES
1)	$L \rightarrow E$ **n**	$L.val = E.val$
2)	$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
3)	$E \rightarrow T$	$E.val = T.val$
4)	$T \rightarrow T_1 * F$	$T.val = T_1.val \times F.val$
5)	$T \rightarrow F$	$T.val = F.val$
6)	$F \rightarrow (E)$	$F.val = E.val$
7)	$F \rightarrow$ **digit**	$F.val =$ **digit**.lexval
SSD for a desk calculator		

- **Evaluate the expression 3*5+4n using the above SDD both in bottom up and top down approach**

Solution: Bottom up evaluation for this expression is shown below

- In both case first we need to draw the parse tree.

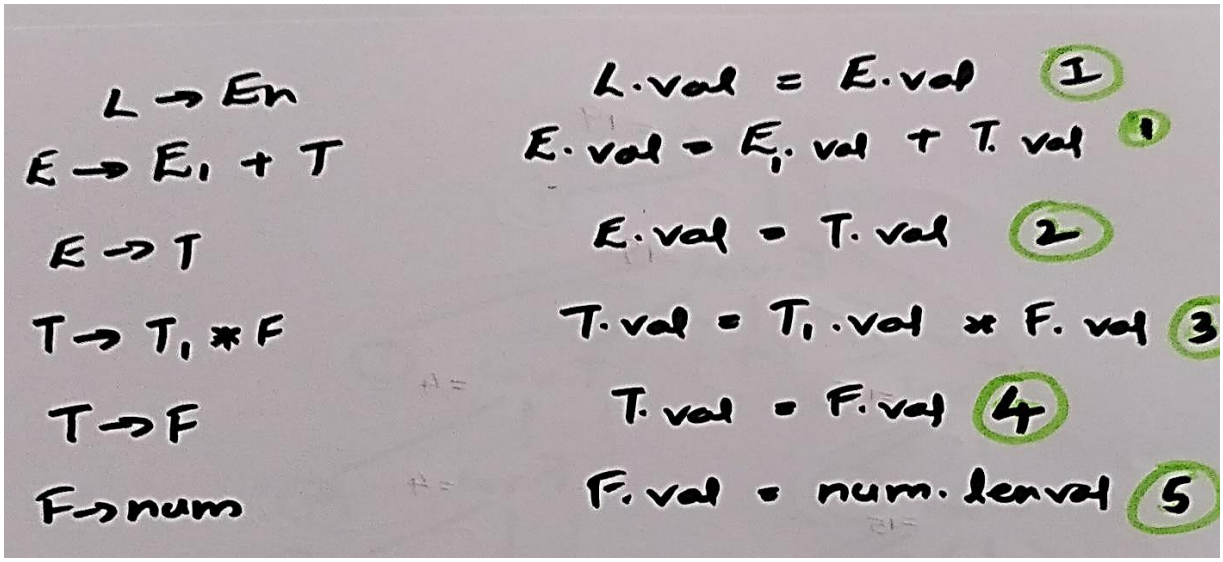
- Then traverse from top to down left to right
- In bottom up approach, whenever there is a reduction ,go to the production and carry out the action

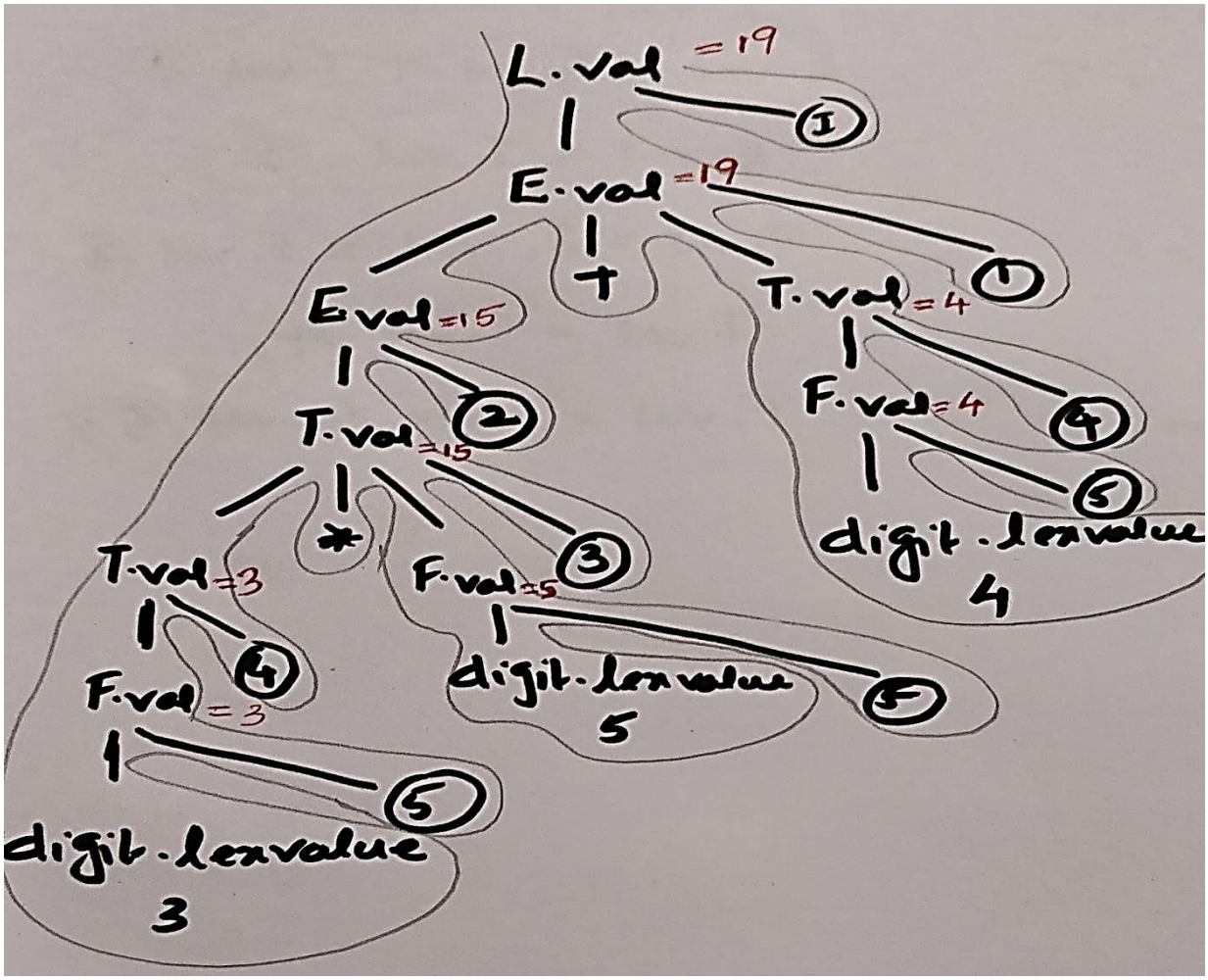


Annotated parse tree for 3 * 5 + 4 n

Solution: Top down approach:

- First we need to draw the parse tree.
- Then traverse from top to down left to right
- In Top down up approach, whenever we come across a semantic action, it is performed.





Workout problems

1. Evaluate the expression 4-6/2
2. Evaluate the expression 2+ (3*4)

Traversal

Procedure visit (n:node)

begin

for each child m of n, from left to right do

visit(m)

evaluate semantic rule at node n

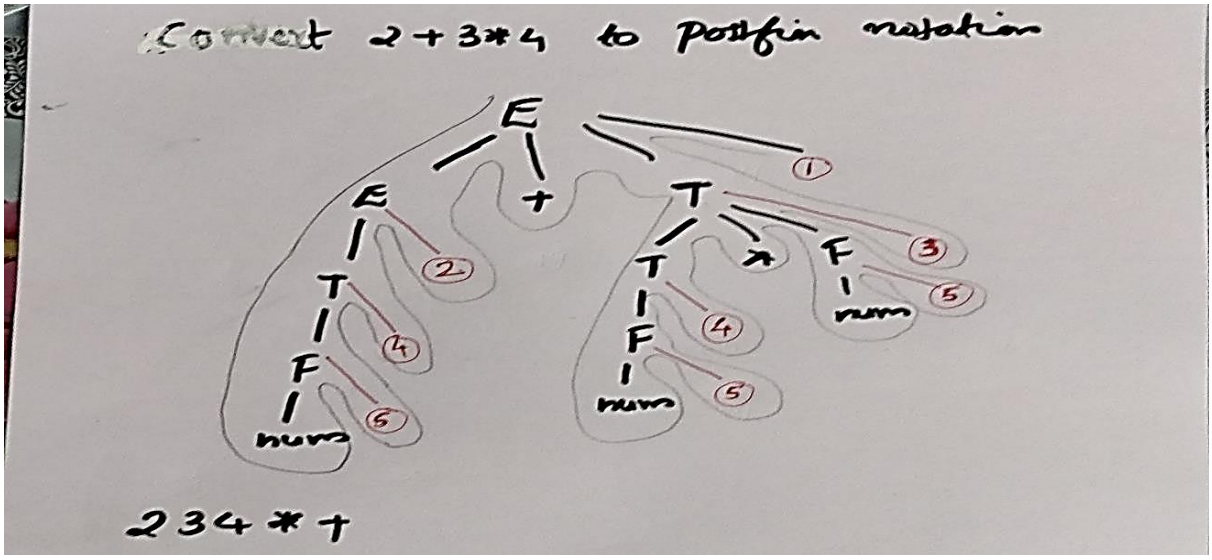
end

Syntax Directed Translation Schemes (SDT'S)

- It is a CFG with program fragments embedded with the production body.
- The program fragments are called semantic actions

SDD for infix to postfix translation

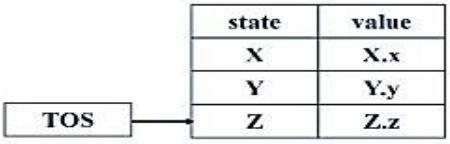
GRAMMAR	**SEMANTIC ACTIONS**
$E \rightarrow E + T$	{ Print ("+") } 1
$E \rightarrow T$	{ } 2
$T \rightarrow T * F$	{ Print ("*") } 3
$T \rightarrow F$	{ } 4
$F \rightarrow num$	{ Print (num.lvalue) } 5

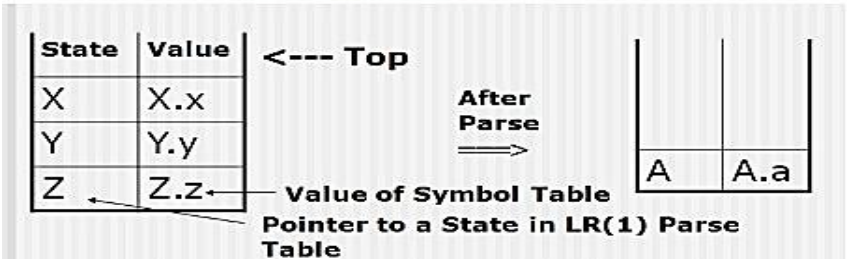


Bottom up evaluation of S-attributed definition

- Syntax-directed definitions with only synthesized attributes(S- attribute) can be evaluated by a bottom up parser (BUP) as input is parsed
- In this approach, the parser will keep the values of synthesized attributes associated with the grammar symbol on its stack.
- The stack is implemented as a pair of state and value.
- When a reduction is made ,the values of the synthesized attributes are computed from the attribute appearing on the stack for the grammar symbols
- implementation is by using an LR parser (e.g. YACC)

- e.g. A => XYZ and A.a := f (X.x, Y.y, Z.z)

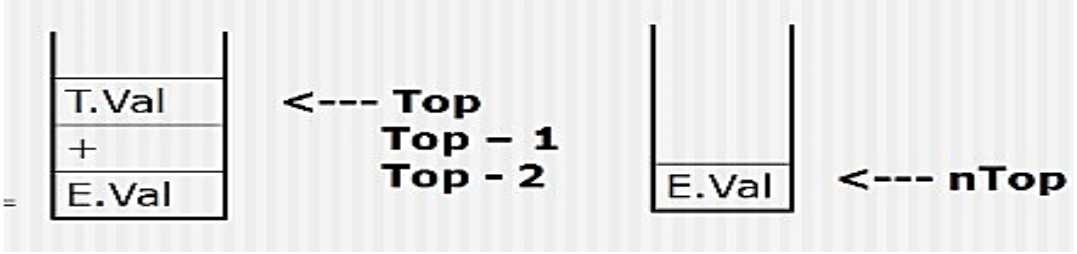




Consider again the syntax-directed definition of the desk calculator.

PRODUCTION	CODE FRAGMENT
$L \rightarrow E$ n	*print* (*val* [*top*])
$E \rightarrow E_1 + T$	*val* [*ntop*] := *val* [*top* − 2] + *val* [*top*]
$E \rightarrow T$	
$T \rightarrow T_1 * F$	*val* [*ntop*] := *val* [*top* − 2] × *val* [*top*]
$T \rightarrow F$	
$F \rightarrow (E)$	*val* [*ntop*] := *val* [*top* − 1]
$F \rightarrow$ digit	

Fig. 5.16. Implementation of a desk calculator with an LR parser.



Example :- SDD and code fragment using S attributed definition for the input 3*5+4n is as follows:

Production	Semantic Rule	Code fragment
L → En	L.val = E.val or Print(E.val)	Print(val[top])
E → E1 + T	E.val = E1.val + T.val	val[ntop] = val[top-2] + val[top]
E → T	E.val = T.val	
T → T1 * F	T.val = T1.val * F.val	val[ntop] = val[top-2] * val[top]
T → F	T.val = F.val	
F → (E)	F.val = E.val	val[ntop] = val[top-1]
F → digit	F.val = digit.lvalue.	

- The following Figure shows the moves made by the parser on input 3*5+4n.
  - Stack states are replaced by their corresponding grammar symbol;
  - Instead of the token digit the actual value is shown.

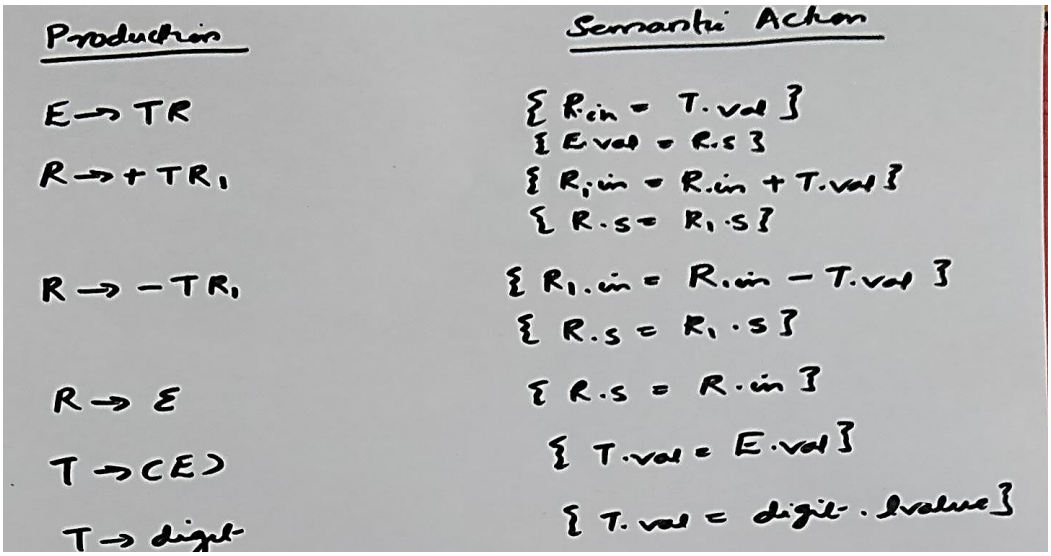
INPUT	state	val	PRODUCTION USED
3*5+4n	–	–	
*5+4n	3	3	
*5+4n	F	3	F → digit
*5+4n	T	3	T → F
5+4n	T *	3 _	
+4n	T * 5	3 _ 5	
+4n	T * F	3 _ 5	F → digit
+4n	T	15	T → T * F
+4n	E	15	E → T
4n	E +	15 _	
n	E + 4	15 _ 4	
n	E + F	15 _ 4	F → digit
n	E + T	15 _ 4	T → F
n	E	19	E → E + T
	E n	19 _	
	L	19	L → E n

Top Down Translation

- implementation of L-attribute definitions during predictive parsing using left recursion grammars and left recursion elimination algorithms
- e.g. translation schema with 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	=>	(E)	{ T.val := E.val }
T	=>	num	{ T.val := num.val }

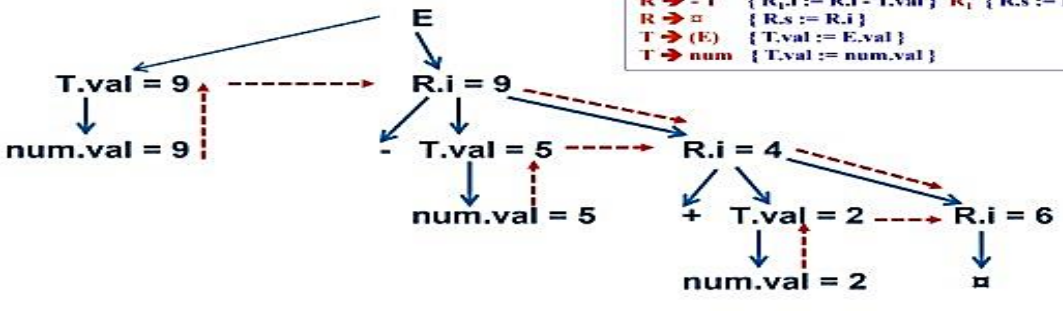
Do left recursion for this grammar



Here R represents E', i for inheritance, s for synthesised attributes.

- Evaluation of the expression 9-5+2

E → T	{ R.i := T.val }	R	{ E.val := R.s }
R → + T	{ R_1.i := R.i + T.val }	R_1	{ R.s := R_1.s }
R → - T	{ R_1.i := R.i - T.val }	R_1	{ R.s := R_1.s }
R → ¤	{ R.s := R.i }		
T → (E)	{ T.val := E.val }		
T → num	{ T.val := num.val }		



Type Checking

- The main aim of the compiler is to translate the source program to a form that can be executed on a target machine.
- For this purpose the compiler need to
    1. Check that the source program follows the syntax and semantics of the concerned language.
    2. Check the flow of data between variables.

What is type?

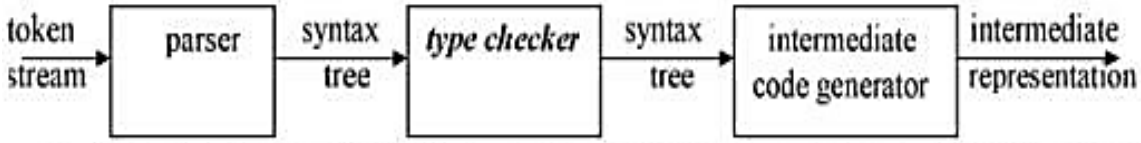
- Type is a notion or a rule that varies from language to language.
- So type checking is done to ensure that whether the source program follows the syntactic and semantic rule of that language.
- Type checking can be of two types:
    1. Static checking (Done at compile time)
    2. Dynamic checking. (Done during run time)

Static checking(Semantic check)

- Its helps in detecting and reporting program errors.
- Some of the static check includes:
    1. Type checks
    2. Flow of control checks
    3. Uniqueness check – object must be defined exactly one for some scenarios
    4. Name related check-Same name must appear two or more times
       Eg: function call and definition.

1. **Type checks** – A compiler should report an error if an operator is applied to an incompatible operand. Example: If an array variable and function variable are added together.
2. **Flow-of-control checks** – Statements that cause flow of control to leave a construct must have some place to which to transfer the flow of control. Example: An error occurs when an enclosing statement, such as break, does not exist in switch statement.

Position of type checker



- A *type checker* verifies that the type of a construct matches that expected by its context. For example : arithmetic operator *mod* in Pascal requires integer operands, so a type checker verifies that the operands of *mod* have type integer.
- Type information gathered by a type checker may be needed when code is generated.

Type expressions

- Type of a language construct will be denoted by type expression
- Type expression can be:
  1. Basic type
  2. Type expression formed by applying an operator called a type constructor to other type expression.

The following are the definitions of type expressions:

1. Basic types such as *boolean, char, integer, real* are type expressions.

   A special basic type, *type_error* , will signal an error during type checking; *void* denoting "the absence of a value" allows statements to be checked.
2. Since type expressions may be named, a type name is a type expression.
3. A type constructor applied to type expressions is a type expression.
   Constructors include:
   Arrays : If T is a type expression then *array* (I,T) is a type expression denoting the type of an array with elements of type T and index set I.

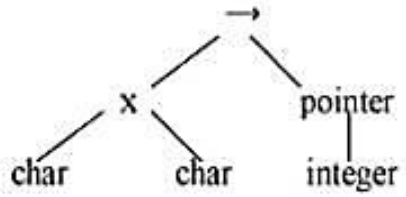
   Products : If T_1 and T_2 are type expressions, then their Cartesian product $T_1 \times T_2$ is a type expression.

Pointers : If T is a type expression, then *pointer*(T) is a type expression denoting the type "pointer to an object of type T".
For example, ***var p:*** ↑ ***row*** declares variable p to have type *pointer*(row).

Functions : A function in programming languages maps a *domain type D* to a *range type R*. The type of such function is denoted by the type expression D → R

4. Type expressions may contain variables whose values are type expressions.

Tree representation for char x char → *pointer* (integer)



Type systems

- A *type system* is a collection of rules for assigning type expressions to the various parts of a program.
- A type checker implements a type system. It is specified in a syntax-directed manner.
- Different type systems may be used by different compilers or processors of the same language.

Sound type system

A *sound* type system eliminates the need for dynamic checking for type errors because it allows us to determine statically that these errors cannot occur when the target program runs. That is, if a sound type system assigns a type other than *type_error* to a program part, then type errors cannot occur when the target code for the program part is run.

Strongly typed language

A language is strongly typed if its compiler can guarantee that the programs it accepts will execute without type errors.

Error Recovery

- Since type checking has the potential for catching errors in program, it is desirable for type checker to recover from errors, so it can check the rest of the input.
- Error handling has to be designed into the type system right from the start; the type checking rules must be prepared to cope with errors.

Specification of simple type checker

Here, we specify a type checker for a simple language in which the type of each identifier must be declared before the identifier is used. The type checker is a translation scheme that synthesizes the type of each expression from the types of its subexpressions. The type checker can handle arrays, pointers, statements and functions.

A simple language

$P \rightarrow D\ ;\ E$
$D \rightarrow D\ ;\ D \mid \text{id}\ :\ T$
$T \rightarrow \text{char} \mid \text{integer} \mid \text{array}\ [\ \text{num}\ ]\ \text{of}\ T \mid \uparrow T$
$E \rightarrow \text{literal} \mid \text{num} \mid \text{id} \mid E\ \text{mod}\ E \mid E\ [\ E\ ] \mid E \uparrow$

Figure 7.1: Grammar for source language

- This grammar will generate programs represented by the nonterminal P, consisting of a sequence of declaration D followed by a single expression E.

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

Figure 7.2: Type checking scheme

Type checking for expressions

Production	*SemanticRules*
$E \rightarrow$ literal	E.*type* = *char*
$E \rightarrow$ num	E.*type* = *integer*
$E \rightarrow$ id	E.*type* = *lookup* (id.*entry*)
$E \rightarrow E_1$ **mod** E_2	E.*type* = **if** E_1.*type* = *integer* **and** E_2.*type* = *integer* **then** *integer* **else** *type_error*
$E \rightarrow E_1[E_2]$	E.*type* = **if** E_2.*type* = *integer* **and** E_1.*type* = *array*(*s*, *t*) **then** *t* **else** *type_error*
$E \rightarrow E_1 \uparrow$	E.*type* = **if** E_1.*type* = *pointer*(*t*) **then** *t* **else** *type_error*

Type checking for statements

- Statements do not have values, therefore a special type void can be assigned to them.
- If an error occurs within a statement, the type assigned to the statement is type_error.

S→ id= E { S.type=(if id.type=E.type then void) else type error}

S→ if E then S1 { S.type=(if E.type=Boolean then S1.type) else type error}

S→While E do S1 { S.type=(if E.type=Boolean then S1.type) else type error}

S→S1;S2 { S.type=(if S1.type=void and S2.type=void then void) else type error}

Type checking for functions

E→E1(E2) {E.type=(if E2.type=S and E1.type=s→t then t) else type error}

T→T1→T2 {T.type=T1.type→T2.type}

****