Semantic Analysis

यो बाट 1 short question आउन 5 marks को लागि

Introduction: Semantic Analysis is the third phase of compiler which provides meaning to its constructs, like tokens and syntax structure.

CFG + semantic rules = Syntax Directed Definitions.

Examples of Semantic Errors:

Example1: Use of a non-initalized variable.

```
int i;
i++; //the variable i is not initialized.
```

Example2: Type incompatibility

```
int a="hello"; //the types String and int are not compatible.
```

Example3: Errors in expressions

```
char s='A';
int a=15-s; //the '-' operator does not support type char.
```

Example4: Array index out of range.

```
int a[10];
a[12]=25; //12 is not legal index, we have index 0 to 9
          for array of size 10.
```

Example5: Unknown references

```
int a;
printf("%d", a); //a is not initialized.
```

Type Checking: Compiler must check that the source program follows both the syntatic and semantic conventions of the source language. Type checking is the process of checking the data type of different variables. The design of a type checker for a language is based on information about the syntatic construct in the language, the notation of type, and the rules for assigning types to the language constructs.

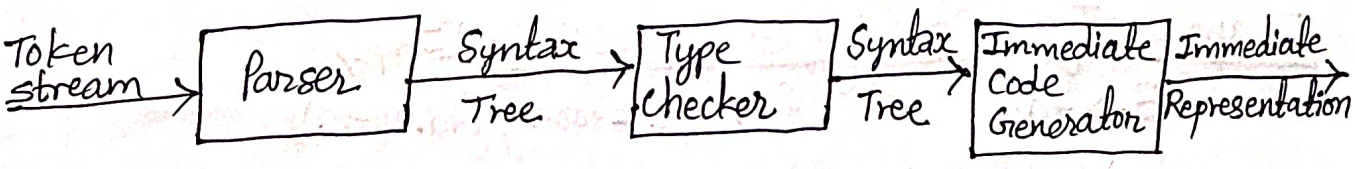


Fig: Position of Type Checker.

⊛ **Type Systems:** The collection of different data types and their associated rules to assign types to programming language constructs is known as type systems. It is an informal type system rules, for example "if both operands of addition are of type integer, then the result is of type integer." A type checker implements type system.

⊛ **Type Expressions:** The type of a language construct will be denoted by a "type expression". A type expression is either a basic type or is formed by applying an operator called a type constructor to other type expressions. The sets of basic types and constructors depend on the language to be checked.

Constructors include: ← just understand only

- **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. Example: array(0...99, int).
- **Products:** If T1 and T2 are type expressions, then their Cartesian product T1 × T2 is a type expression. Example: int × int.
- **Records:** The record type constructor will be applied to a tuple formed from field names and field types.
- **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 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. Example: int→int represents the type of a function which takes an int value as parameter, and its return type is also int.

Example: Type Checking of Expressions: [Imp]

E → id	{E.type = lookup(id.entry)}
E → charliteral	{E.type = char}
E → intliteral	{E.type = int}

$E \rightarrow E1 + E2$	{E.type = (E1.type == E2.type) ? E1.type : type_error}
$E \rightarrow E1 \uparrow$	{E.type = (E1.type == pointer(t)) ? t : type_error}
$E \rightarrow E1 [E2]$	{E.type = (E2.type == int and E1.type == array(s,t)) ? t : type_error}
$S \rightarrow id = E$	{S.type = (id.type == E.type) ? void : type_error}

(This bracket denotes index of array)

$S \rightarrow$ if E then S1	{S.type = (E.type == boolean) ? S1.type : type_error}
$S \rightarrow$ while E do S1	{S.type = (E.type == boolean) ? S1.type : type_error}
$S \rightarrow S1 ; S2$	{S.type = (S1.type == void and S2.type == void) ? void : type_error}

⊛Static vs. Dynamic Type Checking: [Imp]

Static Type Checking	Dynamic Type Checking
The type checking at compilation time is known as static type checking.	The type checking at runtime is known as dynamic type checking.
A language is statically-typed if the type of variable is known at compile time.	A language is strongly-typed, if every program it's compiler accepts will execute without type errors.
It includes languages such as C, C++, C#, Java, FORTRAN, Pascal etc.	It includes languages such as JavaScript, LISP, PHP, Python, Ruby etc.

Typical examples of static checking are:
→ Type checks
→ Flow-of-control checks
→ Uniqueness checks
→ Name-related checks
	Example: array out of bounds as below;
	`int a[10];`
	`for(int i=0; i<20; i++)`
	`a[i]=i;`

It is the process of reducing possibilities for bugs in programs and then checking that parts of program have been connected in a consistent way.	It is the process of verifying the type safety of a program at runtime.

⑧. Type Casting vs. Type Conversion:

Type Casting	Type Conversion (Coercion)
In type casting, a data type is converted into another data type by a programmer using casting operator.	In type conversion, a data type is converted into another data type by a compiler.
Type casting can be applied to compatible data types as well as incompatible data types.	Type conversion can only be applied to compatible data types.
In type casting, casting operator is needed in order to cast the data type to another data type.	In type conversion there is no need for a casting operator.
In type casting, the destination data type may be smaller than the source data type, when converting the data type to another data type.	In type conversion, the destination data type can't be smaller than source data type.
Type casting takes place during the program design by programmers.	Type conversion is done at the compile time.

⊛ Syntax-Directed Translation (STD):

(STD भनेर describe गर्नी आए यसको defn र SDD, SDTS short मा describe गर्नै)

Syntax-Directed Translation (SDT) refers to a method of compiler implementation where the source language translation is completely driven by the parser. Almost all modern compilers are syntax-directed. The general approach to Syntax-Directed Translation is to construct a parse tree or syntax tree and compute the values of attributes at the nodes of the tree by visiting them in same order. There are two ways to represent semantic rules associated with grammar symbols.

→ Syntax-Directed Definitions (SDD).
→ Syntax-Directed Translation Schemes (SDTS).

(explain this with suitable example भनेर सोधिन्छ normally. So अहिले explain गर्ने र 1,1 example गर्ने पुग्छ।)

i) Syntax-Directed Definitions (SDD): [Imp]

A syntax-directed definition (SDD) is a context-free grammar together with attributes and rules. Attributes are associated with grammar symbols and rules are associated with productions. A SDD is a generalization of a context-free grammar in which each grammar symbol is associated with a set of attributes.

Example: The syntax directed definition for a simple desk calculator.

Production	Semantic Rules
L → E return	Print (E.val)
E → E1 + T	E.val = E1.val + T.val
E → T	E.val = T.val
T → T1 * F	T.val = T1.val * F.val
T → F	T.val = F.val
F → digit	F.val = digit.lexval

(digit: any variable that does not produce further)

ii) Syntax-Directed Translation Schemes (SDTS): [Imp]

SDTS embeds program fragments called semantic actions within production bodies. The position of semantic action in a production body determines the order in which the action is executed. It is used to evaluate the order of semantic rules.

Syntax, $A \rightarrow \{\ldots\} X \{\ldots\} Y \{\ldots\}$

where, within the curly brackets we define semantic actions.

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

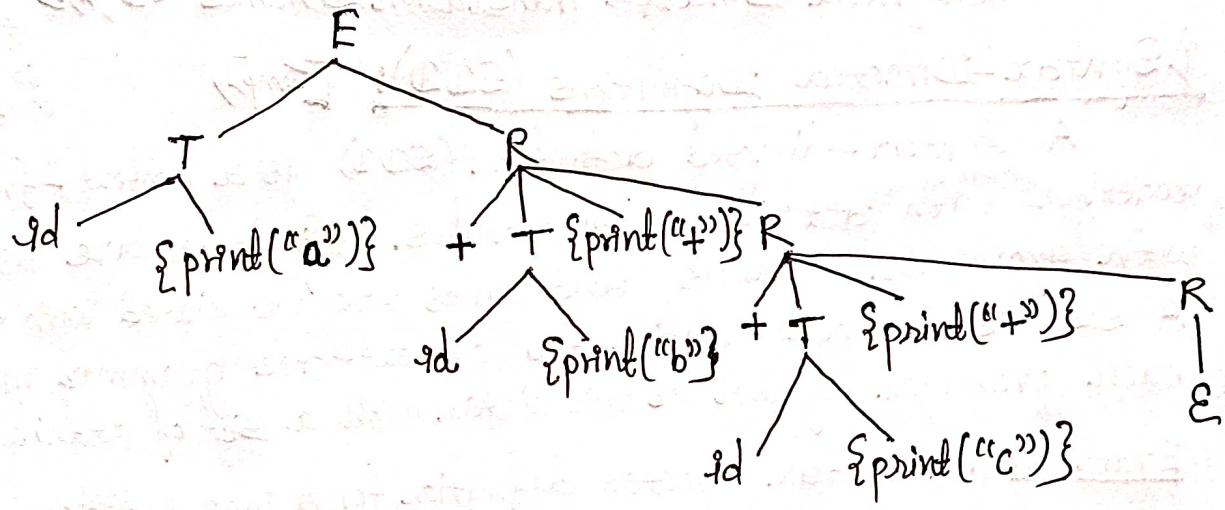
$E \rightarrow TR$

$R \rightarrow +T \{print("+")\} R_1$

$R \rightarrow \varepsilon$

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

Infix expression (a+b+c) ⟶ postfix expression (ab+c+).



⊛. ATTRIBUTE TYPES:

1) Synthesized Attribute (↑): If the value of the attribute only depends upon it's children then it is synthesized attribute. Simply we can also say that the attributes of a node that are derived from it's children nodes are called synthesized attributes. Let we have $S \rightarrow ABC$ as our production, now if S is taking values from it's child nodes (A, B, C) then it is said to be a synthesized attribute.

Example for Synthesized Attributes:

Production	Semantic Rules
L→E return	print (E.val)
E→E1+T	E.val = E1.val + T.val
E→T	E.val = T.val
T→T1*F	T.val = T1.val * F.val
F→digit	F.val = digit.lexval

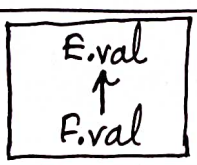
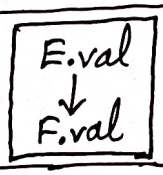
Same example as SDD

2) **Inherited Attribute (→, ↓):** A node in which attributes are derived from the parent or siblings of node is called inherited attribute of that node. If the value of the attribute depends upon it's parent or siblings then it is inherited attribute. Let we have production as $S \rightarrow ABC$, now if 'A' gets values from S, B and C, similarly if B can take values from S, A, C, likewise C can take values from S, A, and B.

Example for Inherited Attributes:

Production	Semantic Rules
D → TL	L.in = T.type
T → int	T.type = integer
T → real	T.type = real
L → L_1, id	L_1.in = L.in; addtype (id.entry, L.in)
L → id	addtype (id.entry, L.in).

⊛. Synthesized attributes vs. Inherited attributes:

Synthesized attribute	Inherited attributes.
Attributes of a node that are derived from its children nodes are called synthesized attributes.	Attributes which are derived from the parent or siblings of node are called inherited attribute.
The production must have non-terminal as its head.	The production must have non-terminal as a symbol in its body.
It can be evaluated during a single bottom-up traversal of parse tree.	It can be evaluated during a single top-down and sideways traversal of parse tree.
Synthesized attributes can be contained by both the terminals and non-terminals.	Inherited attributes can be only contained by non-terminals.
Example: E → F, E.val = F.val 	Example: E → F, E.val = F.val 

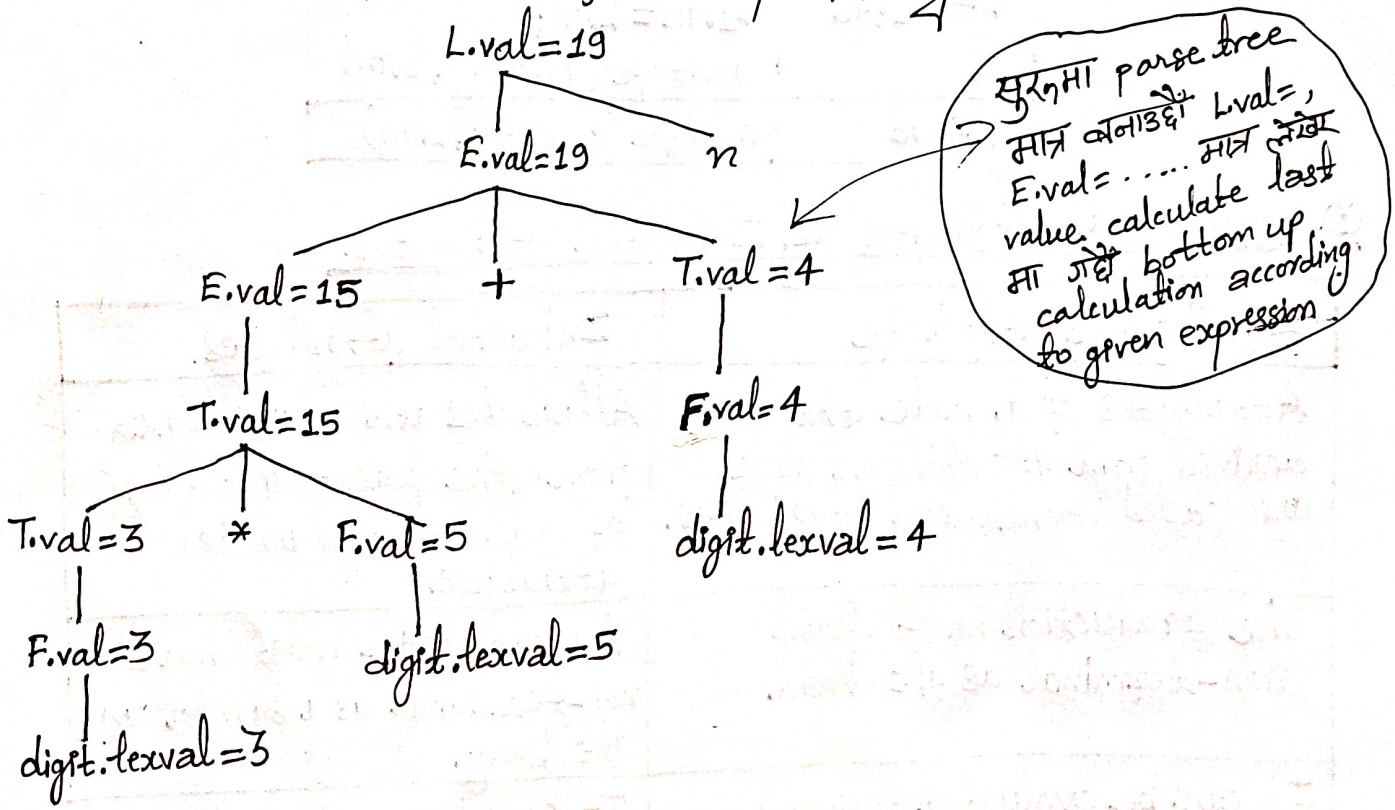
⊛ Annoted Parse Tree:

A parse tree constructing for a given input string in which each node showing the values of attributes is called an anoted parse tree. It is a parse tree showing the values of the attributes at each node. The process of computing the attribute values at the nodes is called annoting or decorating the parse tree.

Example 1: Lets take a grammar,

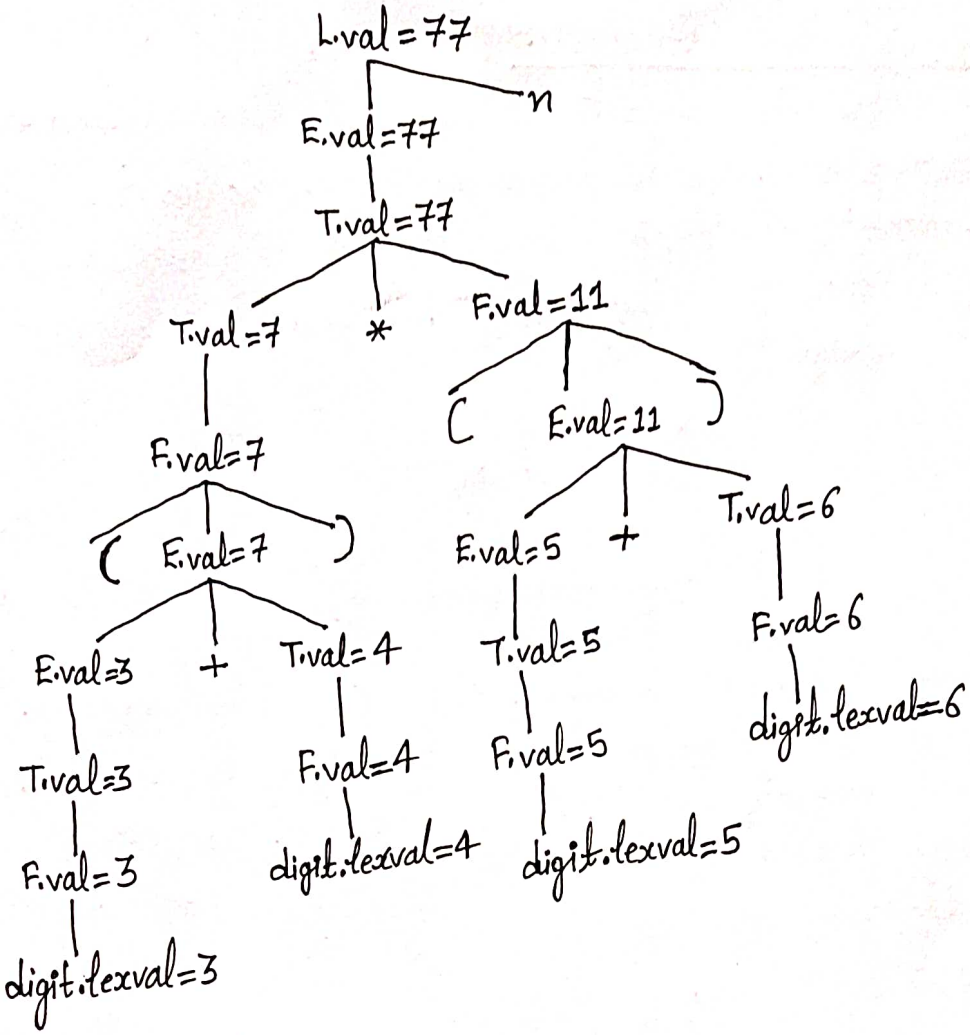
$L \to E\,n$
$E \to E1 + T$
$E \to T$
$T \to T_1 * F$
$T \to F$
$F \to (E)$
$F \to digit.$

Now the annoted parse tree for the input string 3*5+4 is,



Example 2: For the Syntax-Directed Definitions (SDD) of the grammar, $(3+4)*(5+6)n$, give annoted parse tree.

Solution:



⊛. S-Attributed Definitions:

If an SDT uses only synthesized attributes, it is called as S-attributed SDT. Attribute values for the non-terminal at the head is computed from the attribute values of the symbols at the body of the production. The attributes of an S-attributed SDD can be evaluated in bottom up order of nodes of the parse tree.

Example for S-attributed definitions:

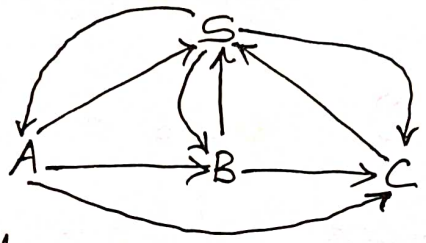
Production	Semantic Rules
L→E return	print (E.val)
E→E1+T	E.val= E1.val +T.val
E→T	E.val=T.val
T→F	T.val= F.val
F→digit	F.val =digit.lexval

Same example as before

⊛ L-Attributed Definitions:

This form of SDT uses both synthesized and inherited attributes with restriction of not taking values from right siblings. In L-attributed SDTs, a non-terminal can get values from its parent, child, and sibling nodes, as in the following production:

$S \rightarrow ABC$



S can take values from A, B, and C (synthesized). 'A' can take values from S only. B can take values from S and A. C can get values from S, A, and B. No non-terminal can get values from the sibling to its right like A does not take value from B or C. Attributes in L-attributed SDTs are evaluated by depth-first and left-to-right parsing manner.

Example:

Production	Semantic Rules
$T \rightarrow FT'$	T'.inh = F.val
$T' \rightarrow *FT_1'$	T_1'.inh = T'.inh * F.val