Chapter – 5

Syntax directed definition:

Syntax Directed Definitions (SDD) are formal methods of attaching semantic information to the syntactic structure of a programming language. SDDs improve the means of context-free **high-level** by instances, adding a semantic rule set for every production of the grammar. The rules described in these definitions state how to derive values such as types, memory locations, or fragments of code from the structure of an input object.

Syntax Directed Translation (SDT) is the action of translating a high-level language program into an intermediate language or machine language according to the semantic rules imposed by SDDs. Semantic actions in SDT, act in coordination with the parsing process in order to provide the translation of the input code. These actions are declarative and they are triggered during the parsing phase of the message to yield the result.

What is Syntax Directed Translation?

Syntax Directed Translation (SDT), is a technical manner utilized in semantics and language translation whereby a source language is translated into an intermediate language or a target language through semantic actions, and attributes attached to the grammar. The translation process follows the structure of the grammar, besides the performance of semantic actions is interwoven with the processes of input parsing.

The semantic actions are normally included in the grammatical rules and can be either performed synchronously or asynchronously with the parsing actions. Integrated development systems such as SDT offer tools to compilers such as code generation, lexical analysis, evaluation of expressions, definition, **grammar,** and checking of types among other services.

What are Annotated Parse Trees?

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

Features of Annotated Parse Trees

- High level specification
- Hides implementation details
- Explicit order of evaluation is not specified

Types of Attributes

There are two types of attributes:

1. Synthesized Attributes: These are those attributes which derive 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.

Example:

```
E --> E1 + T  { E.val = E1.val + T.val}
```

In this, E.val derive its values from E1.val and T.val

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.

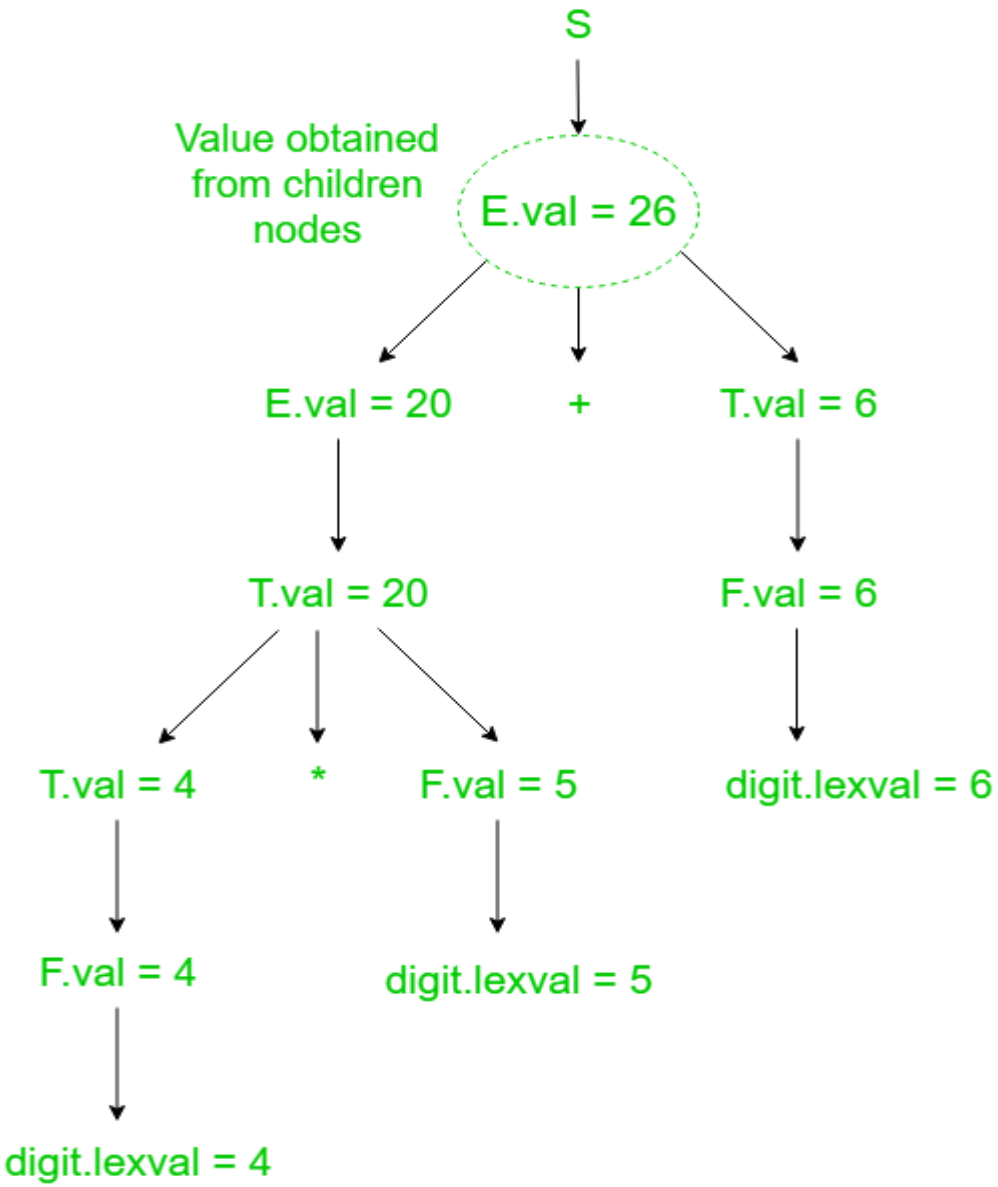
Example: Consider the following grammar

```
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

Production	Semantic Actions
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



Annotated Parse Tree

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 –> T1 * F production, the corresponding semantic action is T.val = T1.val * F.val . Hence, T.val = 4 * 5 = 20

Similarly, combination of E1.val + T.val becomes E.val i.e. E.val = E1.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.

2. Inherited Attributes: These are the attributes which derive 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 }

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.

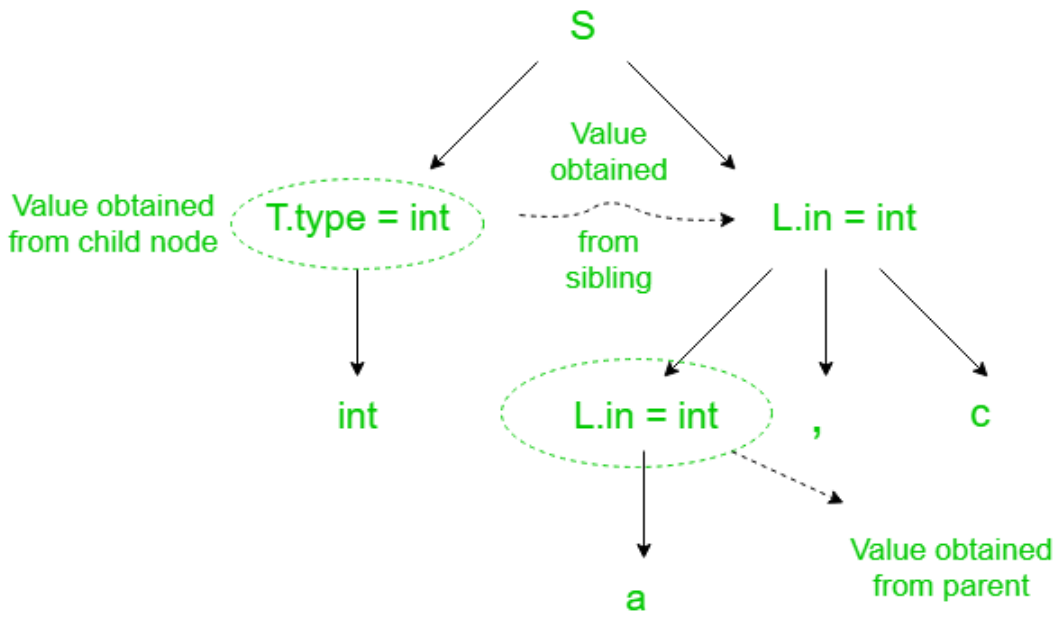
Example: Consider the following grammar

S --> T L
T --> int
T --> float
T --> double
L --> L1, id
L --> id

The SDD for the above grammar can be written as follow

Production	Semantic Actions
S --> T L	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.

S-Attributed SDD AND L-Attributed SDD:

What is S-attributed SDT?

An S-attributed SDT **(Synthesized Attributed SDT)** is one of the Syntax-Directed Translation schemes in which all attributes are synthesized. Predictive attributes are calculated as a result of attributes of the parse tree children nodes, their values are defined. Normally, the value of a synthesized attribute is produced at the leaf nodes and then passed up to the root of the parse tree.

Key Features:

- Bottom-Up Evaluation: Similarly, synthesized attributes are assessed in the bottom-up approach.
- Suitable for Bottom-Up Parsing: Thus, S-attributed SDTs are more suitable to the approaches to bottom-up parsing, including the shift-reduce parsers.
- Simple and Efficient: As all attributes are generated there are no inherited attributes involved thus making it easier to implement.

Example:

Let us consider a production role such that.

$E \rightarrow E1 + T$

Hence, the synthesized attribute E.val can be calculated as:

E.val = E1.val + T.val

What is L-attributed SDT?

An L-Attributed SDT **(Left-Attributed SDT)** also permits synthesized attributes as well as inherited attributes. Some of these attributes are forced attributes, which are inherited from the parent node, other attributes are synthetic attributes, which are calculated like S-attributed SDTs. L-attributed SDTs is an algebra of system design and the key feature of the algebra is that attributes can only be inherited on the left side of a particular production rule.

Key Features:

- Top-Down Evaluation: Evaluations of the inherited attributes are carried out in a manner that is top-down while those of the synthesized attributes are bottom–up.
- Suitable for Top-Down Parsing: L-attributed SDTs are typical for the top-down approaches to parsing such as the recursive descent parsers.
- Allows More Complex Dependencies: Since a language that has the capability of supporting both, inherent as well as synthesized attributes for its terms define more sophisticated semantic rules, then it is appropriate for a semantic network.

Example:

Consider a production rule.

S → A B

Now, the inherited attribute A.inh can be calculated as.

A.inh = f(S.inh)

likewise, B can also have synthesized attributes based on A :

B.synth = g(A.synth)

Translation Scheme:

Syntax directed translation scheme

- The Syntax directed translation scheme is a context -free grammar.
- The syntax directed translation scheme is used to evaluate the order of semantic rules.
- In translation scheme, the semantic rules are embedded within the right side of the productions.
- The position at which an action is to be executed is shown by enclosed between braces. It is written within the right side of the production.

Example

Production	Semantic Rules

S → E $	{ printE.VAL }
E → E + E	{E.VAL := E.VAL + E.VAL }
E → E * E	{E.VAL := E.VAL * E.VAL }
E → (E)	{E.VAL := E.VAL }
E → I	{E.VAL := I.VAL }
I → I digit	{I.VAL := 10 * I.VAL + LEXVAL }
I → digit	{ I.VAL:= LEXVAL}

Syntax direct translation is implemented by constructing a parse tree and performing the actions in a left to right depth first order.

SDT is implementing by parse the input and produce a parse tree as a result.

Example

Production	**Semantic Rules**
S → E $	{ printE.VAL }
E → E + E	{E.VAL := E.VAL + E.VAL }
E → E * E	{E.VAL := E.VAL * E.VAL }

E → (E)	{E.VAL := E.VAL }
E → I	{E.VAL := I.VAL }
I → I digit	{I.VAL := 10 * I.VAL + LEXVAL }
I → digit	{ I.VAL:= LEXVAL}

Parse tree for SDT:

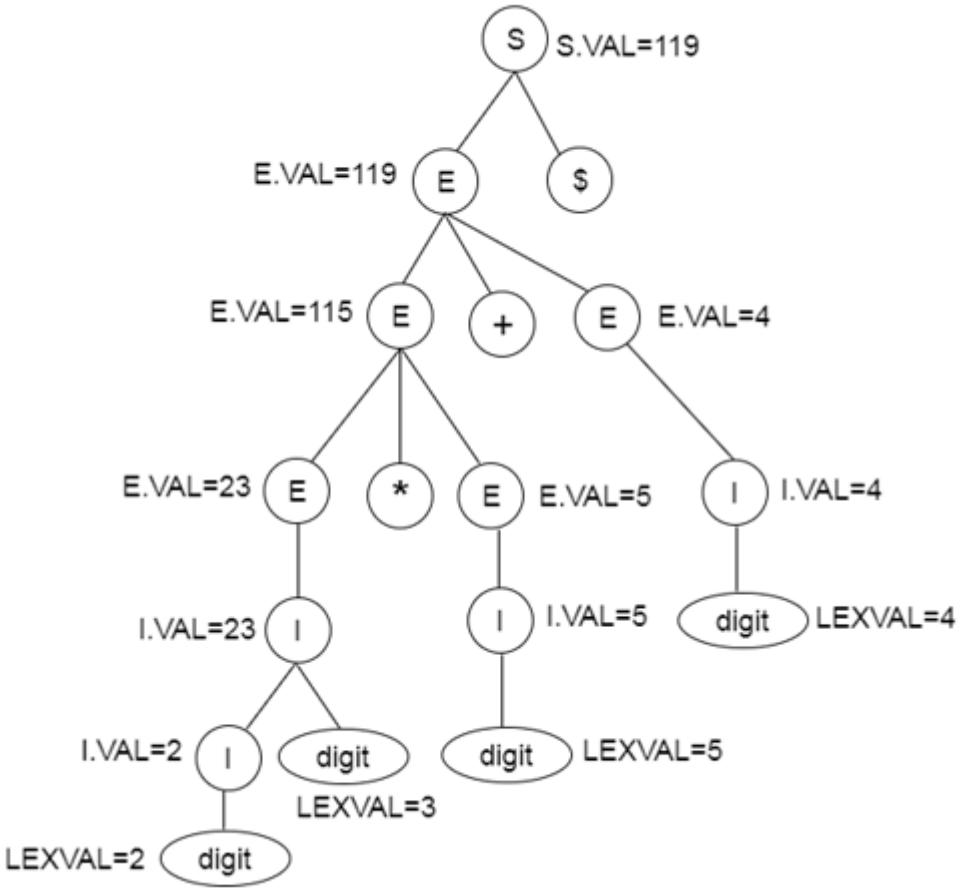
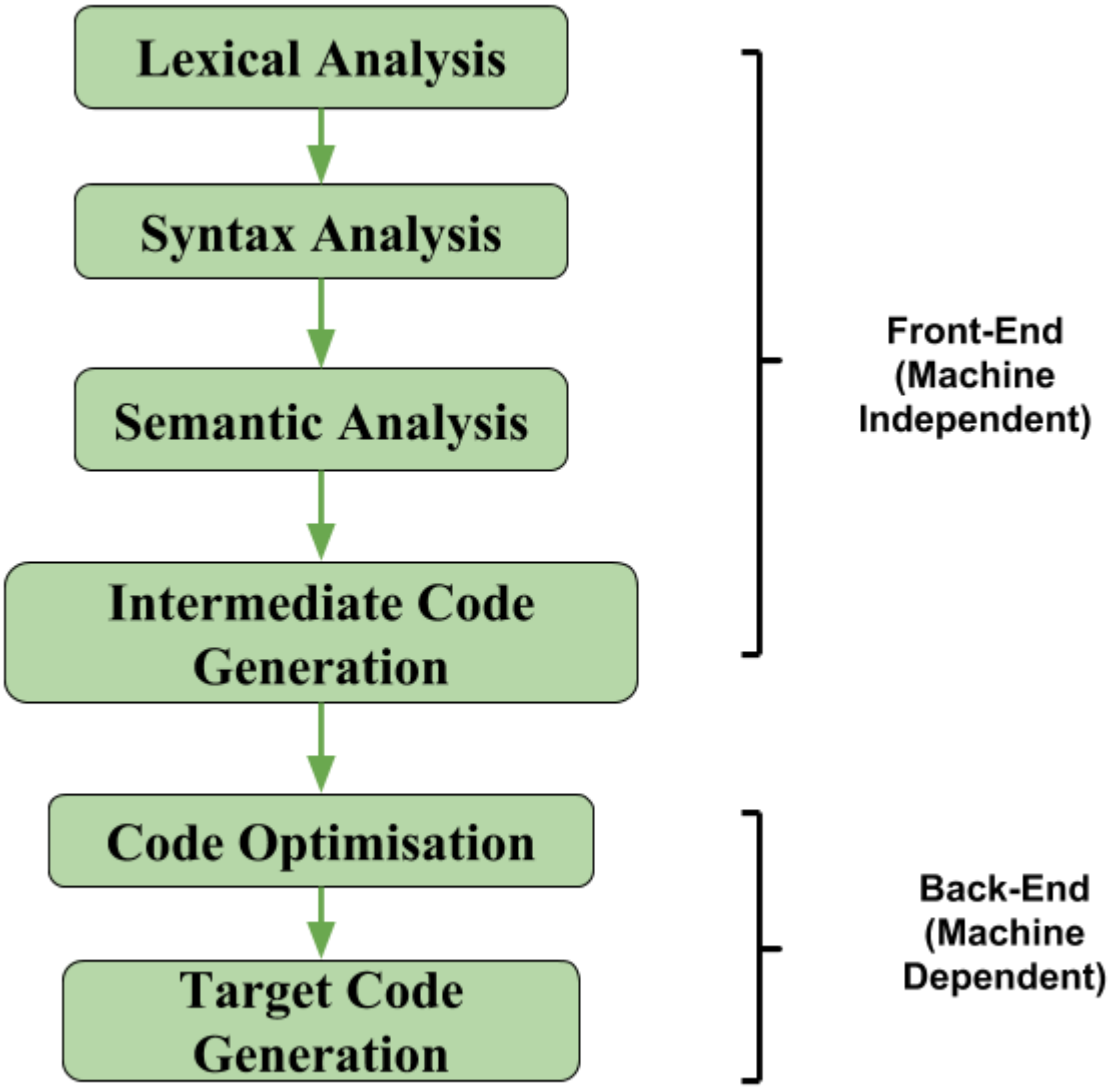


Fig: Parse tree

Intermediate Code Generation:
Intermediate Code Generation is a stage in the process of compiling a program, where the compiler translates the source code into an intermediate representation. This representation is not machine code but is simpler than the original high-level code. Here's how it works:

- **Translation:** The compiler takes the high-level code (like C or Java) and converts it into an intermediate form, which can be easier to analyze and manipulate.
- **Portability**: This intermediate code can often run on different types of machines without needing major changes, making it more versatile.
- **Optimization:** Before turning it into machine code, the compiler can optimize this intermediate code to make the final program run faster or use less memory.



If we generate machine code directly from source code then for n target machine we will have optimizers and n code generator but if we will have a machine-independent intermediate code, we will have only one optimizer. Intermediate code can be either language-specific (e.g., Bytecode for Java) or language. independent (three-address code). The following are commonly used intermediate code representations:

Postfix Notation

- Also known as reverse Polish notation or suffix notation.

- In the infix notation, the operator is placed between operands, e.g., *a + b*. Postfix notation positions the operator at the right end, as in *ab +*.
- For any postfix expressions *e1* and *e2* with a binary operator *(+)* , applying the operator yields *e1e2+*.
- Postfix notation eliminates the need for parentheses, as the operator's position and arity allow unambiguous expression decoding.
- In postfix notation, the operator consistently follows the operand.

Example 1: The postfix representation of the expression (a + b) * c is : ab + c *

Example 2: The postfix representation of the expression (a – b) * (c + d) + (a – b) is : ab – cd + *ab -+

Read more: Infix to Postfix

Three-Address Code

- A three address statement involves a maximum of three references, consisting of two for operands and one for the result.
- A sequence of three address statements collectively forms a three address code.
- The typical form of a three address statement is expressed as *x = y op z*, where *x, y*, and *z* represent memory addresses.
- Each variable *(x, y, z)* in a three address statement is associated with a specific memory location.

While a standard three address statement includes three references, there are instances where a statement may contain fewer than three references, yet it is still categorized as a three address statement.

Example: The three address code for the expression a + b * c + d : T1 = b * c T2 = a + T1 T3 = T2 + d; T 1 , T2 , T3 are temporary variables.

There are 3 ways to represent a Three-Address Code in compiler design:

i) Quadruples

ii) Triples

iii) Indirect Triples

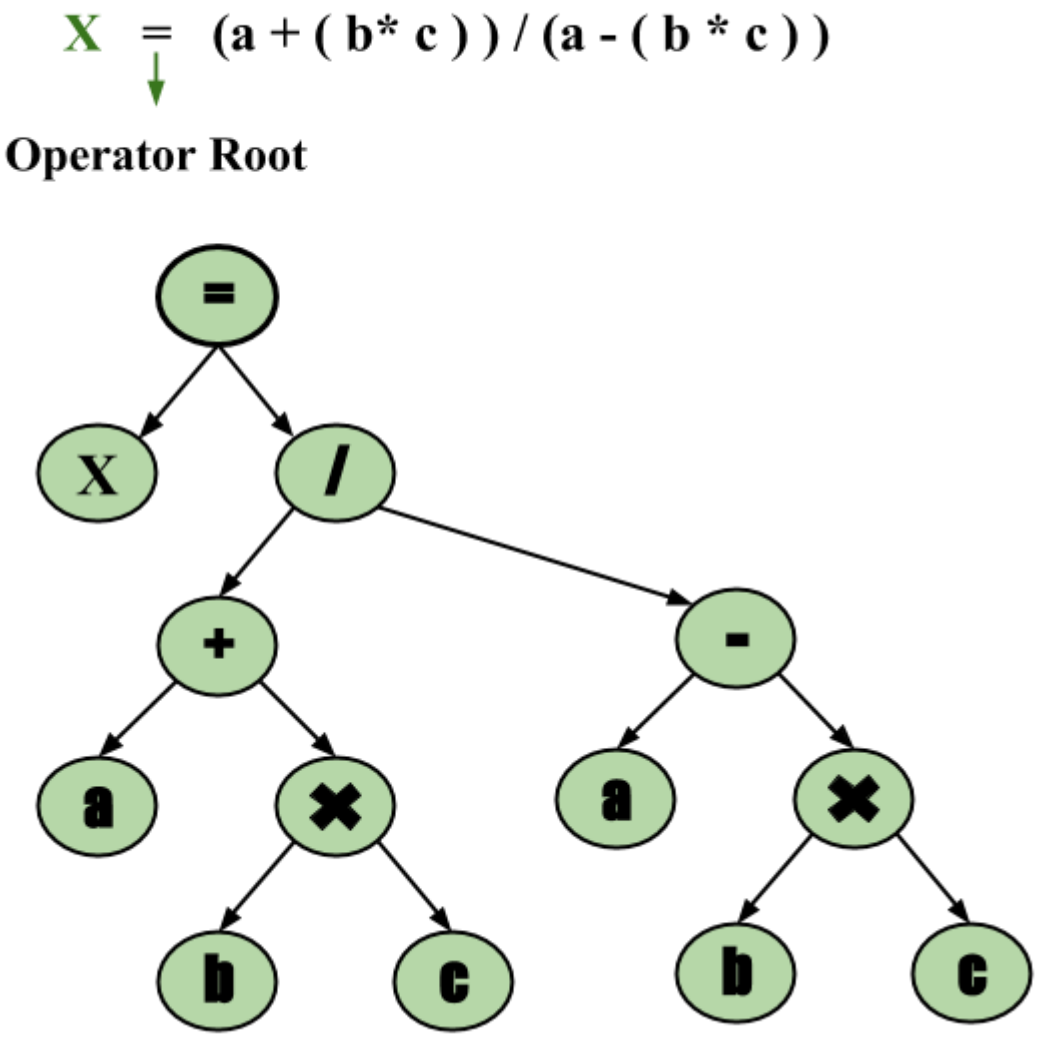
Read more: Three-address code

Syntax Tree

- A syntax tree serves as a condensed representation of a parse tree.
- The operator and keyword nodes present in the parse tree undergo a relocation process to become part of their respective parent nodes in the syntax tree. the internal nodes are operators and child nodes are operands.
- Creating a syntax tree involves strategically placing parentheses within the expression. This technique contributes to a more intuitive representation, making it easier to discern the sequence in which operands should be processed.

The syntax tree not only condenses the parse tree but also offers an improved visual representation of the program's syntactic structure,

Example: x = (a + b * c) / (a – b * c)



Advantages of Intermediate Code Generation

- **Easier to Implement:** Intermediate code generation can simplify the code generation process by reducing the complexity of the input code, making it easier to implement.
- **Facilitates Code Optimization:** Intermediate code generation can enable the use of various code optimization techniques, leading to improved performance and efficiency of the generated code.
- **Platform Independence:** Intermediate code is platform-independent, meaning that it can be translated into machine code or bytecode for any platform.
- **Code Reuse:** Intermediate code can be reused in the future to generate code for other platforms or languages.
- **Easier Debugging:** Intermediate code can be easier to debug than machine code or bytecode, as it is closer to the original source code.

Disadvantages of Intermediate Code Generation

- **Increased Compilation Time:** Intermediate code generation can significantly increase the compilation time, making it less suitable for real-time or time-critical applications.
- **Additional Memory Usage:** Intermediate code generation requires additional memory to store the intermediate representation, which can be a concern for memory-limited systems.
- **Increased Complexity:** Intermediate code generation can increase the complexity of the compiler design, making it harder to implement and maintain.
- **Reduced Performance:** The process of generating intermediate code can result in code that executes slower than code generated directly from the source code.