

UNIT – IV

Syntax Directed Definition

Syntax Directed Definition (SDD) is a kind of abstract specification. It is a generalization of context-free grammar in which each grammar production $X \rightarrow a$ is associated with a set of production rules of the form $s = f(b_1, b_2, \dots, b_k)$ where s is the attribute obtained from function f . The attribute can be a string, number, type, or memory location. Semantic rules are fragments of code that are embedded usually at the end of production and enclosed in curly braces ($\{ \}$).

Example:

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

Annotated Parse Tree—The parse tree containing the values of attributes at each node for a given input string is called an annotated or decorated parse tree.

- 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 attributes derive their values from their children's nodes; that is, the value of a synthesized attribute at a node is computed from the values of attributes at children's nodes in the parse tree.

Example:

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

In this, $E.val$ derives its values from $E_1.val$ and $T.val$

Computation of Synthesized Attributes –

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

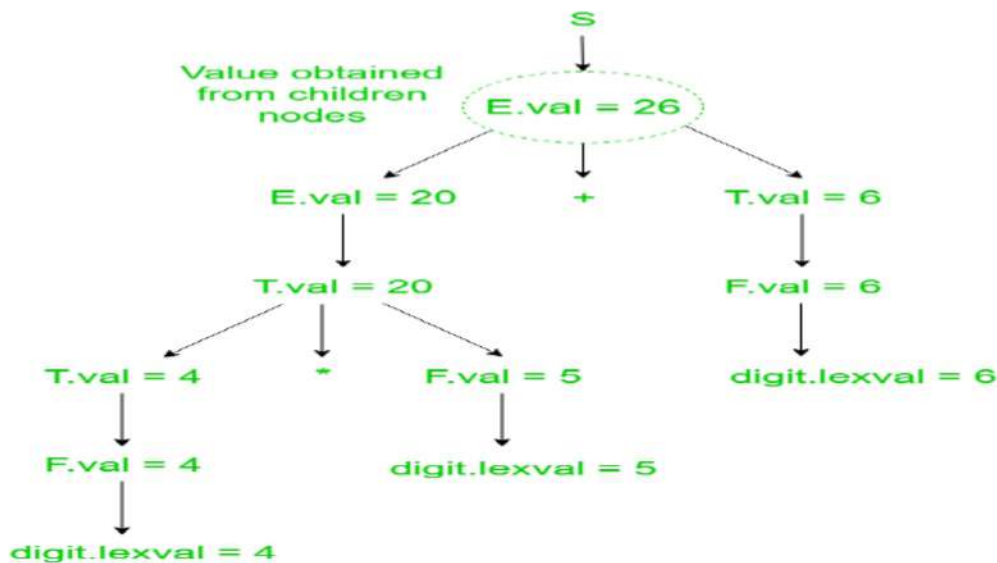
Example: Consider the following grammar

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

The SDD for the above grammar can be written as follows

Production	Semantic Actions
$S \rightarrow E$	$\text{Print}(E.\text{val})$
$E \rightarrow E_1 + T$	$E.\text{val} = E_1.\text{val} + T.\text{val}$
$E \rightarrow T$	$E.\text{val} = T.\text{val}$
$T \rightarrow T_1 * F$	$T.\text{val} = T_1.\text{val} * F.\text{val}$
$T \rightarrow F$	$T.\text{val} = F.\text{val}$
$F \rightarrow \text{digit}$	$F.\text{val} = \text{digit}.\text{lexval}$

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 the computation of attributes, we start from the leftmost bottom node. The rule $F \rightarrow \text{digit}$ is used to reduce the digit to F , and the value of the digit is obtained from the lexical analyzer, which becomes the value of F , i.e., from semantic action $F.\text{val} = \text{digit}.\text{lexval}$. Hence, $F.\text{val} = 4$, and since T is the parent node of F , we get $T.\text{val} = 4$ from the semantic action $T.\text{val} = F.\text{val}$. Then, for $T \rightarrow T_1 * F$ production, the corresponding semantic action is $T.\text{val} = T_1.\text{val} * F.\text{val}$. Hence, $T.\text{val} = 4 * 5 = 20$

Similarly, a combination of $E_1.\text{val} + T.\text{val}$ becomes $E.\text{val}$, i.e., $E.\text{val} = E_1.\text{val} + T.\text{val} = 26$. Then, the production $S \rightarrow E$ is applied to reduce $E.\text{val} = 26$, and its semantic action prints the result $E.\text{val}$. Hence, the output will be 26.

2. Inherited Attributes—These are the attributes whose values derive from their parent or sibling nodes; that is, the value of inherited attributes is computed by the value of parent or sibling nodes.

Example:

$A \rightarrow BCD \quad \{ C.\text{in} = A.\text{in}, C.\text{type} = B.\text{type} \}$

Computation of Inherited Attributes –

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

Example: Consider the following grammar

$S \rightarrow TL$

$T \rightarrow \text{int}$

$T \rightarrow \text{float}$

$T \rightarrow \text{double}$

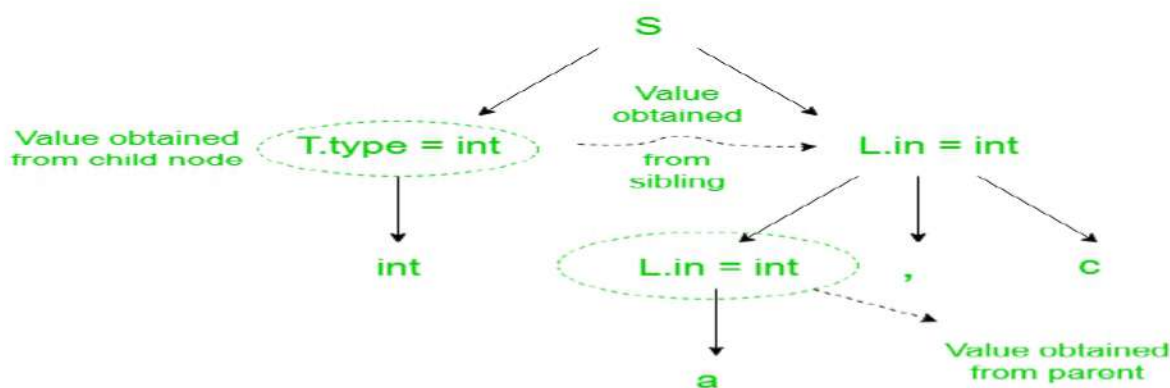
$L \rightarrow L_1, \text{id}$

$L \rightarrow \text{id}$

The SDD for the above grammar can be written as follows

Production	Semantic Actions
$S \rightarrow T L$	$L.in = T.type$
$T \rightarrow \text{int}$	$T.type = \text{int}$
$T \rightarrow \text{float}$	$T.type = \text{float}$
$T \rightarrow \text{double}$	$T.type = \text{double}$
$L \rightarrow L_1, \text{id}$	$L_1.in = L.in$ $\text{Enter_type}(\text{id.entry}, L.in)$
$L \rightarrow \text{id}$	$\text{Entry_type}(\text{id.entry}, L.in)$

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), a linguistic value obtained as int, float, or double. Then, the L node gives the type of identifiers a and c. The computation of type is done in a top-down manner or preorder traversal. Using the function Enter_type, the type of identifiers a and c is inserted in the symbol table at the corresponding id. Entry.

Translation Scheme

Syntax-Directed Translation Schemes

Syntax Directed Translation is a set of productions that have semantic rules embedded inside it. Syntax-directed translation helps in the compiler's semantic analysis phase. SDT has semantic actions along with the production in the grammar. This article concerns postfix SDT and postfix translation schemes with parser stack implementation. Postfix SDTs are SDTs that have semantic actions at the right end of the production. This article also includes SDT with actions inside the output, eliminating left recursion from SDT and SDTs for L-attributed definitions.

Postfix Translation Schemes:

- The syntax-directed translation, which has its semantic actions at the end of the production, is called the **postfix translation scheme**.
- This type of translation of SDT has its corresponding semantics at the last in the RHS of the production.
- SDTs that contain the semantic actions at the proper ends of the production are called **postfix SDTs**.

Example of Postfix SDT

$S \rightarrow A \# B \{S.val = A.val * B.val\}$

$A \rightarrow B @ 1 \{A.val = B.val + 1\}$

$B \rightarrow num \{B.Val = num.lexval\}$

Parser-Stack Implementation of Postfix SDTs:

Postfix SDTs are implemented when the semantic actions are at the right end of the production and with the bottom-up parser(LR parser or shift-reduce parser) with the non-terminals having synthesized attributes.

- The parser stack contains the record for the non-terminals in the grammar and their corresponding attributes.
- The non-terminal symbols of the production are pushed onto the parser stack.
- If the attributes are synthesized and semantic actions are at the proper ends, then the characteristics of the non-terminals are evaluated for the symbol at the top of the stack.
- When the reduction occurs at the top of the stack, the attributes are available. After the action happens, these attributes are replaced by the corresponding LHS non-terminal and its attribute.
- Now, the LHS non-terminal and its attributes are at the top of the stack.

Production

$A \rightarrow BC \{A.str = B.str . C.str\}$

$B \rightarrow a \{B.str = a\}$

$C \rightarrow b \{C.str = b\}$

Initially, the parser stack:

BC Non-terminals

B.str C.str Synthesized attributes

↑

Top of Stack

After the reduction occurs, $A \rightarrow BC$, then after B, C and their attributes are replaced by A and in the attribute. Now, the stack:

A Non-terminals

A.str Synthesized attributes



Top of stack

SDT with action inside the production:

When the semantic actions are present anywhere on the right side of the production, then it is **SDT with action inside the output**.

It is evaluated, and actions are performed immediately after the left non-terminal is processed.

This type of SDT includes both S-attributed and L-attributed SDTs.

If the SDT is parsed in a bottom-up parser, actions are performed immediately after a non-terminal occurs at the top of the parser stack.

If the SDT is parsed in a top-down parser, actions are taken before the expansion of the non-terminal or if the terminal checks for input.

Example of SDT with action inside the production

$$S \rightarrow A + \{\text{print '+'}\} B$$
$$A \rightarrow \{\text{print 'num'}\} B$$
$$B \rightarrow \text{num} \{\text{print 'num'}\}$$

Eliminating Left Recursion from SDT:

The top-down parser cannot parse the grammar with left recursion. Therefore, it should be eliminated, and the grammar can be transformed by removing it.

Grammar with Left Recursion

$$P \rightarrow Pr \mid q$$

Grammar after eliminating left recursion

$$P \rightarrow qA$$
$$A \rightarrow rA \mid \epsilon$$

SDT for L-attributed Definitions:

SDT with L-attributed definitions involves both synthesized and inherited attributes in the production.



To convert an L-attributed definition into its equivalent SDT, follow the underlying rules:

- When the attributes are inherited attributes of any non-terminal, place the action immediately before the non-terminal in the production.
- When the attributes of the non-terminal are synthesized, the action is placed at the right end of that production.

Synthesized and inherited attributes

Differences between Synthesized and Inherited Attributes

In syntax-directed definition, two attributes are used: one is a Synthesized attribute, and another is an inherited attribute. An attribute is considered a Synthesized attribute if the attribute value at child nodes determines its parse tree node value. In contrast, An attribute is an **Inherited attribute** if the attribute value at the parent and sibling node determines its parse tree node value. Now, we shall compare Synthesized Attributes and Inherited Attributes. The comparison between these two attributes is given below:

S.NO	Synthesized Attributes	Inherited Attributes
1.	An attribute is considered a Synthesized attribute if its value at child nodes determines its parse tree node value.	An attribute is said to be Inherited if its value at the parent and sibling nodes determines its parse tree node value.
2.	The production must have a non-terminal as its head.	The production must have a non-terminal as a symbol in its body.
3.	A synthesized attribute at node n is defined only by attribute values at n's children.	An Inherited attribute at node n is defined only in terms of attribute values of n's parent, n itself, and n's siblings.
4.	It can be evaluated during a single bottom-up traversal of the parse tree.	It can be assessed during a single top-down and sideways parse tree traversal.
5.	Both the terminals and non-terminals can contain synthesized attributes.	Both can't contain inherited attributes; They are only contained by non-terminals.
6.	Both S-attributed SDT and L-attributed SDT use synthesized attributes.	Only L-attributed SDT uses the inherited attribute.
7.	<p>EX:- $E.val \rightarrow F.val$</p> 	<p>EX:- $E.val = F.val$</p> 

Synthesized and Inherited Attribute are part of the semantics of a language that provides meaning to its constructs, like tokens and syntax structure. Semantics help interpret symbols, their types, and their relations with each other, and their analysis judges whether or not the syntax structure constructed in the source program derives any meaning. Now, based on features of attributes, we can distinguish between Synthesized and Inherited Attributes.

Following are the essential differences between Synthesized and Inherited Attributes.

Sr. No.	Key	Synthesized Attribute	Inherited Attribute
1	Definition	A synthesized attribute is one whose parse tree node value is determined by the attribute value at child nodes. To illustrate, assume the following	On the other hand, an attribute is said to be Inherited if the attribute value at the parent and sibling node determines its parse tree node value. In the case of

Sr. No.	Key	Synthesized Attribute	Inherited Attribute
		production: $S \rightarrow ABC$. If S takes 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.	$S \rightarrow ABC$, if A can get values from S, B, and C., B can take values from S, A, and C. Likewise, C can take values from S, A, and B. Then S is said to be an Inherited Attribute.
2	Design	As mentioned above, in the case of the Synthesized attribute, the production must have a non-terminal as its head.	On the other hand, in the case of Inherited attributes, the production must have a non-terminal as a symbol in its body.
3	Evaluation	Synthesized attributes can be evaluated during a single bottom-up parse tree traversal.	Conversely, Inherited attributes can be evaluated during a single top-down and sideways traversal of the parse tree.
4	Terminal	Both terminals and Nonterminals can contain the Synthesized attribute.	On the other hand, only Nonterminals can contain the Inherited attribute.
5	Usage	Both S-attributed SDT and L-attributed STD use the synthesized attribute.	On the other hand, the Inherited attribute is used by only L-attributed SDT.

Dependency Graph in Compiler Design

A dependency graph represents the flow of information among the attributes in a parse tree and helps determine the attributes' evaluation order in a parse tree. The main aim of dependency graphs is to help the compiler check for various types of dependencies between statements to prevent them from being executed in the incorrect sequence, i.e., in a way that affects the program's meaning. This central aspect helps identify the program's numerous parallelizable components.

It assists us in determining the impact of a change and the objects that are affected by it. Drawing edges to connect dependent actions can create a dependency graph. These arcs result in partial ordering among operations and prevent a program from running in parallel. Although use-definition chaining is a type of dependency analysis, it results in unduly cautious data reliance estimations. On a shared control route, there may be four dependencies between statements I and j.

Like other-directed networks, dependency graphs have nodes or vertices depicted as boxes or circles with names and arrows linking them in their obligatory traversal direction. Dependency graphs are commonly used in scientific literature to describe semantic links, temporal and causal dependencies between events, and the flow of electric current in electronic circuits. Drawing dependency graphs is so common in computer science that we'll want to employ tools that automate the process based on some basic textual instructions from us.

Types of dependencies:

Dependencies are broadly classified into the following categories:

1. Data Dependencies:

When a statement computes data, it is later utilized by another statement. A state in which instruction must wait for a result from a preceding instruction before it can complete its execution. A data dependence will trigger a stoppage in the flowing services of a processor pipeline or block the parallel issuing of instructions in a superscalar processor in high-performance processors using pipeline or superscalar approaches.

2. Control Dependencies:

Control Dependencies are those that come from a program's well-ordered control flow. A scenario in which a program instruction executes if the previous instruction evaluates in a fashion that permits it to perform is known as control dependence.

3. Flow Dependency:

In computer science, a flow dependence occurs when a program statement refers to the data of a previous statement.

4. Anti-dependence:

When an instruction needs a value later modified, this is known as anti-dependency or write-after-read (WAR). In the following example, instruction two is anti-dependent on instruction 3; the order of these instructions cannot be modified, nor can they be performed in parallel (potentially changing the instruction ordering) because this would adjust the final value of A.

5. Output-Dependency:

An output dependence, also known as write-after-write (WAW), occurs when the sequence in which instructions are executed impacts the variable's ultimate output value. In the example below, instructions 3 and 1 have an output dependence; altering the order of instructions would affect the final value of A, so these instructions cannot be run in parallel.

6. Control-Dependency:

If the outcome of A determines whether B should be performed, instruction B has a control dependence on a previous instruction A. The display style S 2S 2 instruction has a control reliance on the S 1S 1 instruction in the following example. However, display style S 3S 3 is not dependent on display style S 1S 1 because display style S 3S 3 is always done regardless of the result of display style S 1S 1.

Example of Dependency Graph:

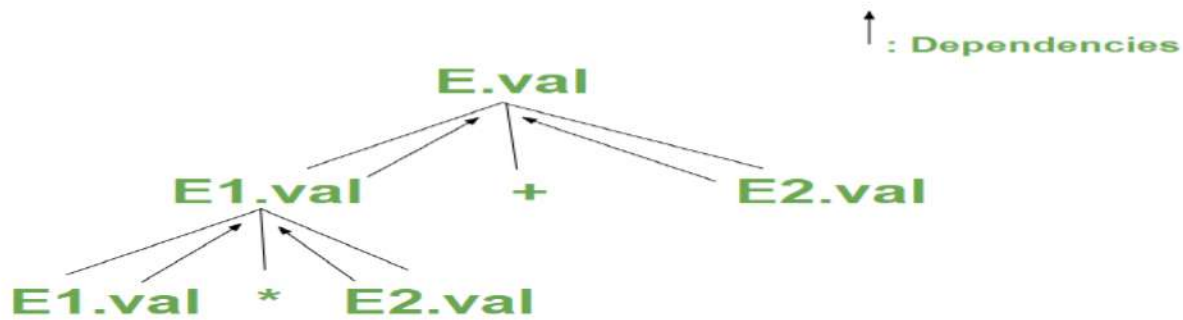
Design a dependency graph for the following grammar:

E \rightarrow E1 + E2

E \rightarrow E1 * E2

PRODUCTIONS	SEMANTIC RULES
E \rightarrow E1 + E2	E.val \rightarrow E1.val + E2.val
E \rightarrow E1 * E2	E.val \rightarrow E1.val * E2.val

The required dependency graph for the above grammar is represented as –



Dependency Graph for the above example

1. Synthesized attributes are represented by **.val**.
2. Hence, **E.val**, **E1.val**, and **E2.val** have synthesized attributes.
3. Black arrows show dependencies.
4. Arrows from E1 and E2 show that the value of E depends upon E1 and E2.

Dependency Resolution:

Dependency resolution is a two-phase procedure performed until the dependency graph is complete.

1. Perform conflict resolution when a new dependence is introduced to the graph to decide which version should be added.
2. In a specific dependence, for example, a versioned module, which is identified as part of the graph, it helps to extract its information so that it helps in adding its dependencies one by one.

When making dependency resolution, Gradle(automation tool) handles two types of conflicts:

Version conflicts:

A version conflict occurs when two components depend on the same module, but their versions are different.

For example:

Let us say that the project depends on the reacting library of Facebook, i.e., “com. google. react:18.7.0”. Here, the version is 18.7.0. We can see that It also depends on some other library, which depends on react, but version 19.0.2 is a different one altogether.

Gradle resolves this by selecting the highest version. In that case, 19.0.2 will be chosen.

But that's not the end of the store. Gradle has a notion of rich version declaration, and there are numerous ways to choose a version from various options.

Implementation conflicts:

Implementation conflicts occur when the dependency graph contains multiple modules that provide the exact implementation or capability in Gradle terminology. Gradle determines what a module offers using variations and capabilities. This is a one-of-a-kind feature that demands that its chapter fully comprehend what it entails and allows. A conflict occurs the moment two modules either:

1. Attempt to select incompatible variants,
2. Declare the same capability

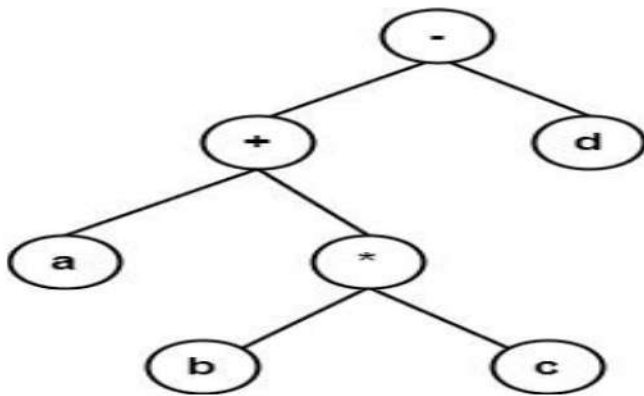
Uses of Dependency Graph:

1. The primary idea behind dependency graphs is for the compiler to check for various types of dependencies between statements to prevent them from being executed in the incorrect sequence, i.e., in a way that affects the program's meaning.
2. This aids it in identifying the program's numerous parallelizable components.
3. Automated software installers: They go around the graph seeking software packages that are needed but haven't been installed yet. The coupling of the packages determines the reliance.
4. Instructions scheduling uses dependency more broadly.
5. Dependency graphs are widely used in Dead code elimination.

Construction of syntax trees

A tree in which each leaf node describes an operand and each interior node an operator. The syntax tree is a shortened form of the **Parse Tree**.

Example 1 – Draw a Syntax Tree for the string $a + b * c - d$.



Rules for constructing a syntax tree

Each node in a syntax tree can be executed as data with multiple fields. In the node for an operator, one field recognizes the operator, and the remaining field includes a pointer to the nodes for the operands. The operator is known as the node's label. The following functions are used to create the syntax tree nodes for expressions with binary operators. Each function returns a pointer to the recently generated node.

- **mknode (op, left, suitable)** – It generates an operator node with the label op and two fields, including pointers to left and right.
- **mkleaf (id, entry)** – It generates an identifier node with label id and the field including the entry, a pointer to the symbol table entry for the identifier.
- **mkleaf (num, val)** – It generates a number node with the label num and a field including the value of the number. For example, construct a syntax tree for an expression $a - 4 + c$. In this sequence, p_1, p_2, \dots, p_5 are pointers to the symbol table entries for identifier 'a' and 'c', respectively.

p_1 – mkleaf (id, entry a);

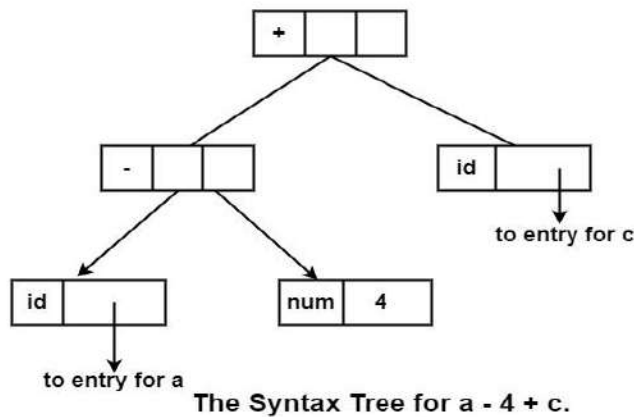
p_2 – mkleaf (num, 4);

p_3 – mknode ('-', p_1, p_2)

p_4 – mkleaf(id, entry c)

```
p5 ← mknode('+', p3, p4);
```

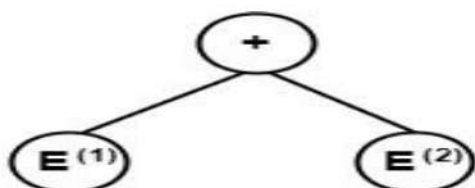
The tree is generated in a bottom-up fashion. The function calls `mkleaf (id, entry a)` and `mkleaf (num 4)` to construct the leaves for `a` and `4`. The pointers to these nodes are stored using `p1` and `p2`. The call `mknodes ('-', p1, p2)` then make the interior node with the leaves for `a` and `four` as children. The syntax tree will be



Syntax Directed Translation of Syntax Trees

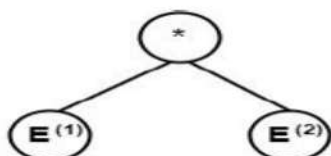
Production	Semantic Action
$E \rightarrow E^{(1)} + E^{(2)}$	$\{E. VAL = \text{Node} (+, E^{(1)}. VAL, E^{(2)}. VAL)\}$
$E \rightarrow E^{(1)} * E^{(2)}$	$\{E. VAL = \text{Node} (*, E^{(1)}. VAL, E^{(2)}. VAL)\}$
$E \rightarrow (E^{(1)})$	$\{E. VAL = E^{(1)}. VAL\}$
$E \rightarrow E^{(1)}$	$\{E. VAL = \text{UNARY} (-, E^{(1)}. VAL)\}$
$E \rightarrow \text{id}$	$\{E. VAL = \text{Leaf} (\text{id})\}$

Node (+, E⁽¹⁾. VAL, E⁽²⁾. VAL) will create a node labeled +.

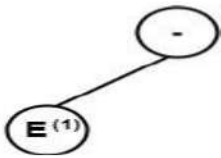


E⁽¹⁾. VAL & E⁽²⁾. VAL are left & right children of this node.

Similarly, Node (*, E⁽¹⁾. VAL, E⁽²⁾. VAL) will make the syntax as –



Function **UNARY (-, E⁽¹⁾. VAL)** will make a node – (unary minus) & E⁽¹⁾. VAL will be the only child of it.



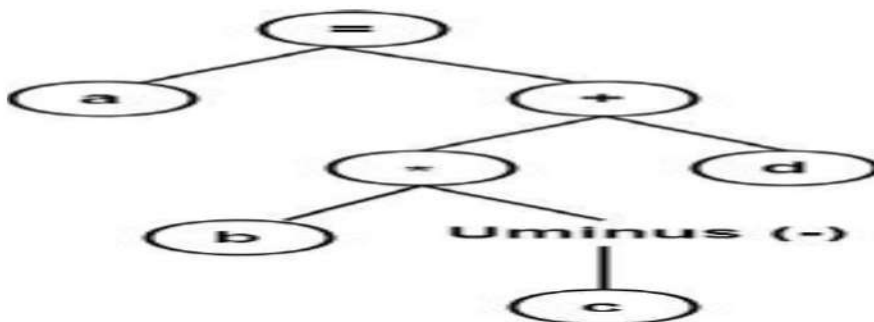
Function **LEAF (id)** will create a Leaf node with label id.



Example 2 – Construct a syntax tree for the expression.

$a = b * -c + d$

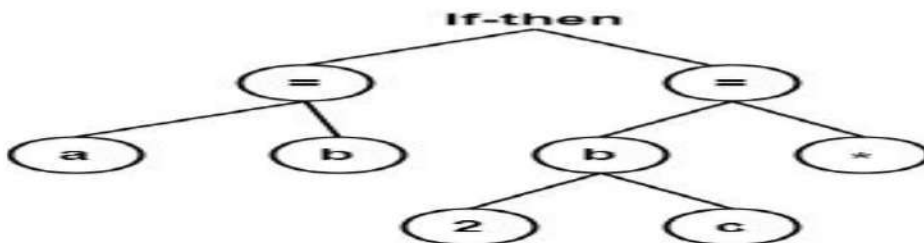
Solution



Example 3 – Construct a syntax tree for a statement.

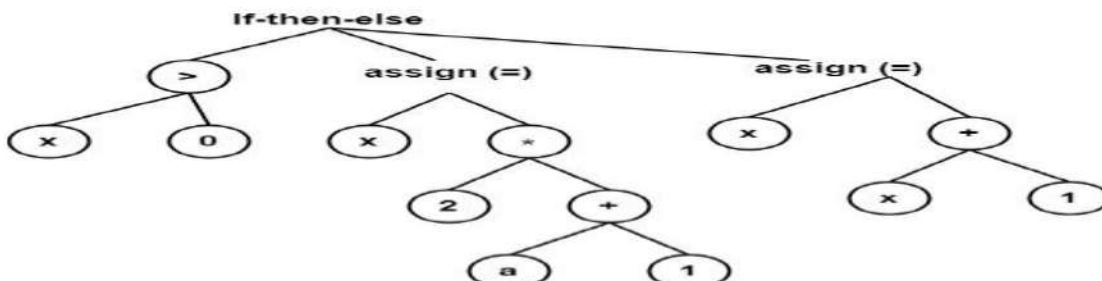
If $a = b$ then $b = 2 * c$

Solution

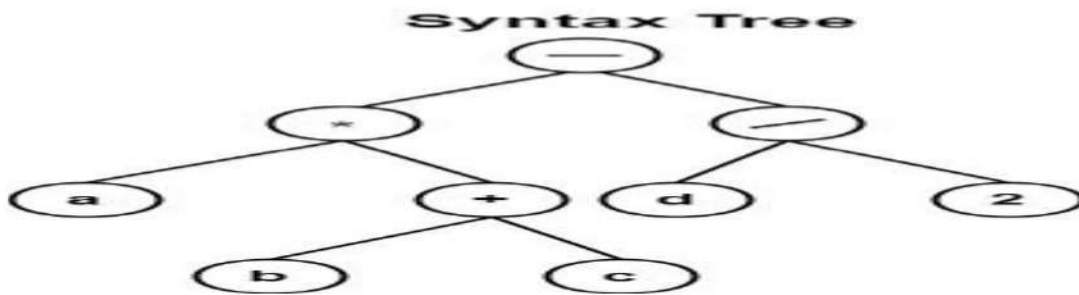


Example 4 – Consider the following code. Draw its syntax Tree.

If $x > 0$ then $x = 2 * (a + 1)$ else $x = x + 1$.



Example 5 – Draw a syntax tree for the arithmetic expression $a * (b + c) - d / 2$. Also, write the given expression in postfix form.

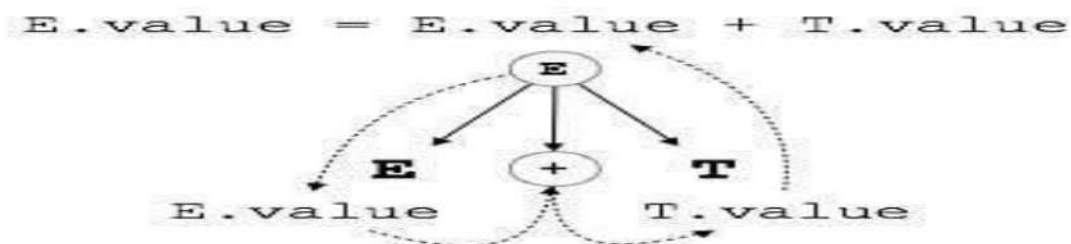


Postfix Notation

a b c + * d 2 / -

S-attributed and L-attributed definitions

S-attributed SDT If an SDT uses only synthesized attributes, it is called an S-attributed SDT. These attributes are evaluated using S-attributed SDTs whose semantic actions are written after the production (right-hand side).



S-attributed SDT As depicted above, attributes in S-attributed SDTs are evaluated in bottom-up parsing, as the values of the parent nodes depend upon the values of the child nodes.

L-attributed SDT: This form of SDT uses synthesized and inherited attributes with the restriction of not taking values from the 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.



Attributes in L-attributed SDTs are evaluated in depth-first and left-to-right parsing manners.

L-attributed SDT We may conclude that if a definition is S-attributed, it is also L-attributed as L-attributed definition encloses S-attributed definitions.

Three address codes, quadruples, triples, and indirect triples

Three-address code is an intermediate code that is easy to generate and easily converted to machine code. It uses at most three addresses and one operator to represent an expression, and the value computed at each instruction is stored in a temporary variable generated by the compiler. The compiler decides the order of operation given by the three-address code.

Three address codes are used in compiler applications:

Optimization: Three-address code is often used as an intermediate representation of code during the optimization phases of the compilation process. The three-address code allows the compiler to analyze the code and perform optimizations that can improve the performance of the generated code.

Code generation: Three address codes can also be used as an intermediate representation of code during the code generation phase of the compilation process. The three-address code allows the compiler to generate code specific to the target platform while ensuring that the generated code is correct and efficient.

Debugging: Three address codes can help debug the code generated by the compiler. Since the address code is a low-level language, it is often easier to read and understand than the final generated code. Developers can use the three address codes to trace the program's execution and identify errors or issues that may be present.

Language translation: Three address codes can also be used to translate code from one programming language to another. Translating code to a common intermediate representation makes translating the code to multiple target languages easier.

General representation –

$a = b \text{ op } c$

A, b, or c represent operands like names, constants, or compiler-generated temporaries, and op represents the operator.

Example-1: Convert the expression $a * -(b + c)$ into three address codes.

$t_1 = b + c$
 $t_2 = \text{uminus } t_1$
 $t_3 = a * t_2$

Example 2: Write three address codes for the following code

```
for(i = 1; i<=10; i++)  
{  
    a[i] = x * 5;  
}
```

Implementation of Three Address Codes –

There are three representations of three address codes, namely

1. Quadruple
2. Triples
3. Indirect Triples

```

    i = 1
L : t1 = x * 5
    t2 = &a
    t3 = sizeof(int)
    t4 = t3 * i
    t5 = t2 + t4
    *t5 = t1
    i = i + 1
    if i <= 10 goto L

```

1. Quadruple is a structure consisting of 4 fields: op, arg1, arg2, and result. Op denotes the operator arg1, and arg2 denotes the two operands, and the result is used to store the result of the expression.

Advantage –

- Easy to rearrange code for global optimization.
- One can quickly access the value of temporary variables using a symbol table.

Disadvantage –

- Contain a lot of temporaries.
- Temporary variable creation increases time and space complexity.

Example – Consider expression $a = b * -c + b * -c$. The three address codes are:

t1 = minus c

t2 = b * t1

t3 = minus c

t4 = b * t3

t5 = t2 + t4

a = t5

#	Op	Arg1	Arg2	Result
(0)	uminus	c		t1
(1)	*	t1	b	t2
(2)	uminus	c		t3
(3)	*	t3	b	t4
(4)	+	t2	t4	t5
(5)	=	t5		a

Quadruple representation

2. Triples—This representation doesn't use an extra temporary variable to represent a single operation. Instead, a pointer is used when a reference to another triple's value is needed. So, it consists of only three fields: op, arg1, and arg2.

Disadvantage –

- Temporaries are implicit and complex when rearranging code.
- It isn't easy to optimize because optimization involves moving intermediate code. When a triple is moved, any other one referring to it must also be updated. With the help of a pointer, one can directly access the symbol table entry.

Example – Consider expression $a = b * -c + b * -c$

#	Op	Arg1	Arg2
(0)	uminus	c	
(1)	*	(0)	b
(2)	uminus	c	
(3)	*	(2)	b
(4)	+	(1)	(3)
(5)	=	a	(4)

Triples representation

3. Indirect Triples—This representation uses a pointer to list all references to computations made separately and stored. It's similar in utility to quadruple representation but requires less space. Temporaries are implicit and more straightforward when rearranging code.

Example – Consider expression $a = b * -c + b * -c$

#	Op	Arg1	Arg2
(14)	uminus	c	
(15)	*	(14)	b
(16)	uminus	c	
(17)	*	(16)	b
(18)	+	(15)	(17)
(19)	=	a	(18)

List of pointers to table

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

Indirect Triples representation

Question – Write quadruple, triples, and indirect triples for the following expression : $(x + y) * (y + z) + (x + y + z)$

Explanation – The three address code is:

$t1 = x + y$

$t2 = y + z$

$t3 = t1 * t2$

$t4 = t1 + z$

$t5 = t3 + t4$

Translation of assignment statements

Translation of Assignment Statements

In syntax-directed translation, the assignment statement mainly deals with expressions. Expressions can be of type actual, integer, array, or record.

Consider the grammar

1. $S \rightarrow id := E$
2. $E \rightarrow E1 + E2$
3. $E \rightarrow E1 * E2$
4. $E \rightarrow (E1)$
5. $E \rightarrow id$

#	Op	Arg1	Arg2	Result
(1)	+	x	y	t1
(2)	+	y	z	t2
(3)	*	t1	t2	t3
(4)	+	t1	z	t4
(5)	+	t3	t4	t5

Quadruple representation

#	Op	Arg1	Arg2
(1)	+	x	y
(2)	+	y	z
(3)	*	(1)	(2)
(4)	+	(1)	z
(5)	+	(3)	(4)

Triples representation

#	Op	Arg1	Arg2
(14)	+	x	y
(15)	+	y	z
(16)	*	(14)	(15)
(17)	+	(14)	z
(18)	+	(16)	(17)

List of pointers to table

#	Statement
(1)	(14)
(2)	(15)
(3)	(16)
(4)	(17)
(5)	(18)

Indirect Triples representation

The translation scheme of the above grammar is given below:

Production rule	Semantic actions
$S \rightarrow id := E$	<pre> {p = look_up(id.name); If p ≠ nil then Emit (p = E.place) Else Error; }</pre>
$E \rightarrow E1 + E2$	<pre> {E.place = newtemp(); Emit (E.place = E1.place '+' E2.place) }</pre>

$E \rightarrow E1 * E2$	<pre>{E.place = newtemp(); Emit (E.place = E1.place '*' E2.place) }</pre>
$E \rightarrow (E1)$	<pre>{E.place = E1.place}</pre>
$E \rightarrow id$	<pre>{p = look_up(id.name); If p ≠ nil then Emit (p = E.place) Else Error; }</pre>

- The p returns the entry for id. Name in the symbol table.
- The Emit function is used to append the three address codes to the output file. Otherwise, it will report an error.
- The new temp () is a function used to generate new temporary variables.
- E.place holds the value of E.

----- X -----