UNIT – IV Syntax-Directed Translation

Syntax Directed Translation in Compiler Design

- Syntax-Directed Translation (SDT) is a method used in compiler design to convert source code into another form while analyzing its structure. It integrates syntax analysis (parsing) with semantic rules to produce intermediate code, machine code, or optimized instructions.
  In SDT, each grammar rule is linked with semantic actions that define how translation should occur. These actions help in tasks like evaluating expressions, checking types, generating code, and handling errors.

  SDT ensures a systematic and structured way of translating programs, allowing information to be processed bottom-up or top-down through the parse tree. This makes translation efficient and accurate, ensuring that every part of the input program is correctly transformed into its executable form.

  SDT relies on three key elements:
  1. **Lexical values of nodes** (such as variable names or numbers).
  2. **Constants** used in computations.
  3. **Attributes** associated with non-terminals that store intermediate results.

  The general process of SDT involves constructing a parse tree or syntax tree, then computing the values of attributes by visiting its nodes in a specific order. However, in many cases, translation can be performed directly during parsing, without explicitly building the tree.

  ### SDD v/s SDT Scheme

Syntax Directed Definition	Syntax Directed Translation
It is a context-free grammar where attributes and rules are combined and associated with grammar symbols and productions, respectively.	It refers to the translation of a string into an array of actions. This is done by adding an action to a rule of context-free grammar. It is a type of compiler interpretation.
Attribute Grammar	Translation Schemes
SDD: Specifies the values of attributes by associating semantic rules with the productions.	SDT: Embeds program fragments (also called semantic actions) within production bodies.
E → E + T { E.val := E1.val + T.val }	**E → E + T { print('+'); }**
Always written at the end of the body of production.	The position of the action defines the order in which the action is executed (in the middle of

Syntax Directed Definition	Syntax Directed Translation
	production or at the end).
More Readable	More Efficient
Used to specify the non-terminals.	Used to implement S-Attributed SDD and L-Attributed SDD.
Specifies what calculation is to be done at each production.	Specifies what calculation is to be done at each production and at what time they must be done.
Left to right evaluation.	Left to right evaluation.
Used to know the value of non-terminals.	Used to generate Intermediate Code.

Attributes in Syntax-Directed Translation

An attribute is any quantity associated with a programming construct in a parse tree. Attributes help in carrying semantic information during the compilation process.

Examples of Attributes:

- **Data types** of variables
- **Line numbers** for error handling
- **Instruction details** for code generation

Types of Attributes

1. Synthesized Attributes

- Defined by a semantic rule associated with the production at node N in the parse tree.
- Computed only using the attribute values of the children and the node itself.
- Mostly used in bottom-up evaluation.

2. Inherited Attributes

- Defined by a semantic rule associated with the parent production of node N.
- Computed using the attribute values of the parent, siblings, and the node itself.
- Used in top-down evaluation.

Read about Differences Between Synthesized and Inherited Attributes.

Attribute Grammars

An Attributed Grammar is a special type of grammar used in compiler design to add extra information (attributes) to syntax rules. This helps in semantic analysis, such as type checking, variable classification, and ensuring correctness in programming languages.

Think of it like a regular grammar with extra labels that help check things like variable types, correctness of expressions, and rule enforcement.

Example of an Attribute Grammar

Production Rule	Semantic Rule
D → T L	L.in := T.type (Passes type information)
T → int	T.type := integer (Defines type as integer)
T → real	T.type := real (Defines type as real)
L → L1 , id	L1.in := L.in addtype (id.entry, L.in) (Passes type info to child and updates symbol table)
L → id	addtype (id.entry, L.in) (Adds type info to symbol table)

- **D → T L** → The type from T is passed to L.
- **T → int / real** → Assigns type integer or real to T.
- **L → id** → Assigns type information to identifier (id).

Grammar and Translation Rules

SDT Scheme	SDD Scheme
E → E + T{ print('+') }	**E → E + T** E.code = E.code || T.code || '+'
E → E – T{ print('-') }	**E → E – T** `E.code = E.code || T.code || '-'
E → T	**E → T**E.code = T.code
T → 0{ print('0') }	**T → 0**T.code = '0'
T → 1{ print('1') }	**T → 1**T.code = '1'
…	…
T → 9{ print('9') }	**T → 9**T.code = '9'

To evaluate translation rules, we can employ one depth-first search traversal on the parse tree. This is possible only because SDT rules don't impose any specific order on evaluation until children's attributes are computed before parents for a grammar having all synthesized attributes. Otherwise, we would have to figure out the best-suited plan to traverse through the

parse tree and evaluate all the attributes in one or more traversals. For better understanding, we will move bottom-up in the left to right fashion for computing the translation rules . Syntax-Directed Translation (SDT) allows us to evaluate arithmetic expressions while parsing. It uses attributes associated with grammar symbols and rules to compute values as we process the parse tree.

The following **context-free grammar (CFG)** defines an arithmetic expression with **addition (+) and multiplication (*)**:

```
E -> E + T     { E.val = E.val + T.val }   // PR#1
E -> T         { E.val = T.val }           // PR#2
T -> T * F     { T.val = T.val * F.val }   // PR#3
T -> F         { T.val = F.val }           // PR#4
F -> INTLIT    { F.val = INTLIT.lexval }   // PR#5
```

Each production rule has a semantic action in {} that defines how values are computed.

- E, T, and F are non-terminals (expression components).
- INTLIT represents an integer literal (actual number).
- val is an attribute used to store computed values at each step.

Let's evaluate the expression: **S = 2 + 3 * 4**

Step 1: Build the Parse Tree

The **parse tree** for 2 + 3 * 4 is structured like this:

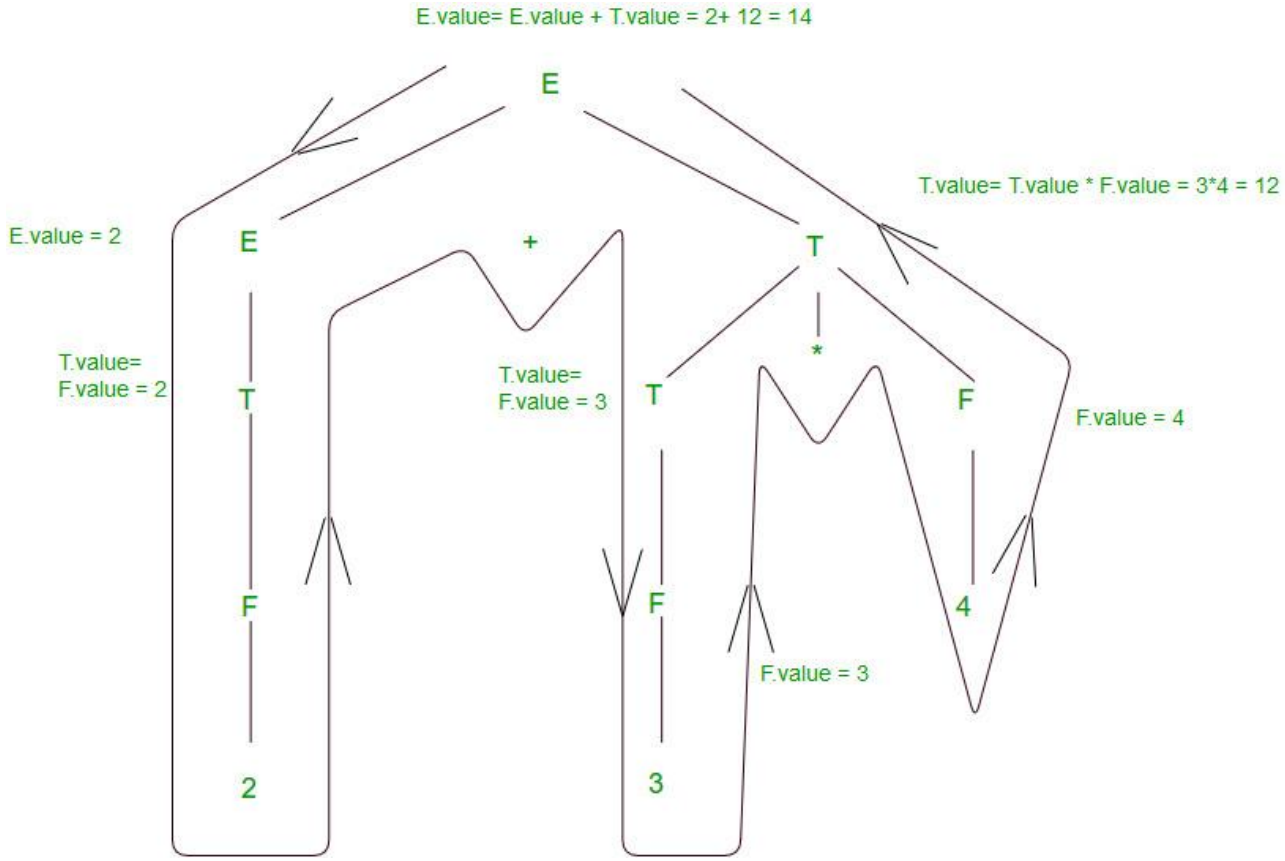
E
/|\
E + T
/ /|\
*T T * F*
/ / |
F F 4
| |
2 3

Step 2: Apply Translation Rules (Bottom-Up Evaluation)

We evaluate the expression step by step in a bottom-up manner (from leaves to root)

F → 2 → F.val = 2
F → 3 → F.val = 3
F → 4 → F.val = 4
T → F (T gets F's value) → T.val = 3
T → T * F → T.val = 3 * 4 = 12
E → T (E gets T's value) → E.val = 12
E → E + T → E.val = 2 + 12 = 14

Thus, the final computed value of 2 + 3 * 4 is **14**.



Advantages of Syntax Directed Translation:

Ease of implementation: SDT is a simple and easy-to-implement method for translating a programming language. It provides a clear and structured way to specify translation rules using grammar rules.

Separation of concerns: SDT separates the translation process from the parsing process, making it easier to modify and maintain the compiler. It also separates the translation concerns from the parsing concerns, allowing for more modular and extensible compiler designs.

Efficient code generation: SDT enables the generation of efficient code by optimizing the translation process. It allows for the use of techniques such as intermediate code generation and code optimization.

Disadvantages of Syntax Directed Translation:

Limited expressiveness: SDT has limited expressiveness in comparison to other translation methods, such as attribute grammars. This limits the types of translations that can be performed using SDT.

Inflexibility: SDT can be inflexible in situations where the translation rules are complex and cannot be easily expressed using grammar rules.

Limited error recovery: SDT is limited in its ability to recover from errors during the translation process. This can result in poor error messages and may make it difficult to locate and fix errors in the input program.

S – Attributed and L – Attributed SDTs in Syntax Directed Translation

---

In Syntax-Directed Translation (SDT), the rules are those that are used to describe how the semantic information flows from one node to the other during the parsing phase. SDTs are derived from context-free grammars where referring semantic actions are connected to grammar productions. Such action can be used in issues such as code generation like intermediate code, type checking, and so on.

There are two main types of SDTs based on how attributes are associated with grammar symbols and how information is propagated: S-attributed SDTs and L-attributed SDTs At this point, the audience of an S-attributed SDT is just the set of instructions within the same statement that uses the variable, while the audience of an L-Attributed SDT is everyone within the statement but the first instruction that uses the variable. In general, these forms of SDTs are important in the construction of efficient compilers particularly in defining how attributes are computed in the parse tree.

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)

Backpatching in Compiler Design

- Backpatching is basically a process of fulfilling unspecified information. This information is of labels. It basically uses the appropriate semantic actions during the process of code generation. It may indicate the address of the Label in goto statements while producing TACs for the given expressions.

  Here basically two passes are used because assigning the positions of these label statements in one pass is quite challenging. It can leave these addresses unidentified in the first pass and then populate them in the second round. Backpatching is the process of filling up gaps in incomplete transformations and information.

  What is Backpatching?

  Backpatching is a method to deal with jumps in the control flow constructs like if statements, loops, etc in the intermediate code generation phase of the compiler. Otherwise, as the target of these jumps may not be known until later in the compilation stages, back patching is a method to fill in these destinations located elsewhere.

  Forward jumps are very common in constructs like the if statements, while loops, switch cases. For example, in a language with goto statements, the destination of a goto may not be resolved until and unless its label appears following the goto statement. Forward references i.e; Jumps from lower addresses to higher address it is a mechanism to maintain and solve these.

  Need for Backpatching:

  Backpatching is mainly used for two purposes:

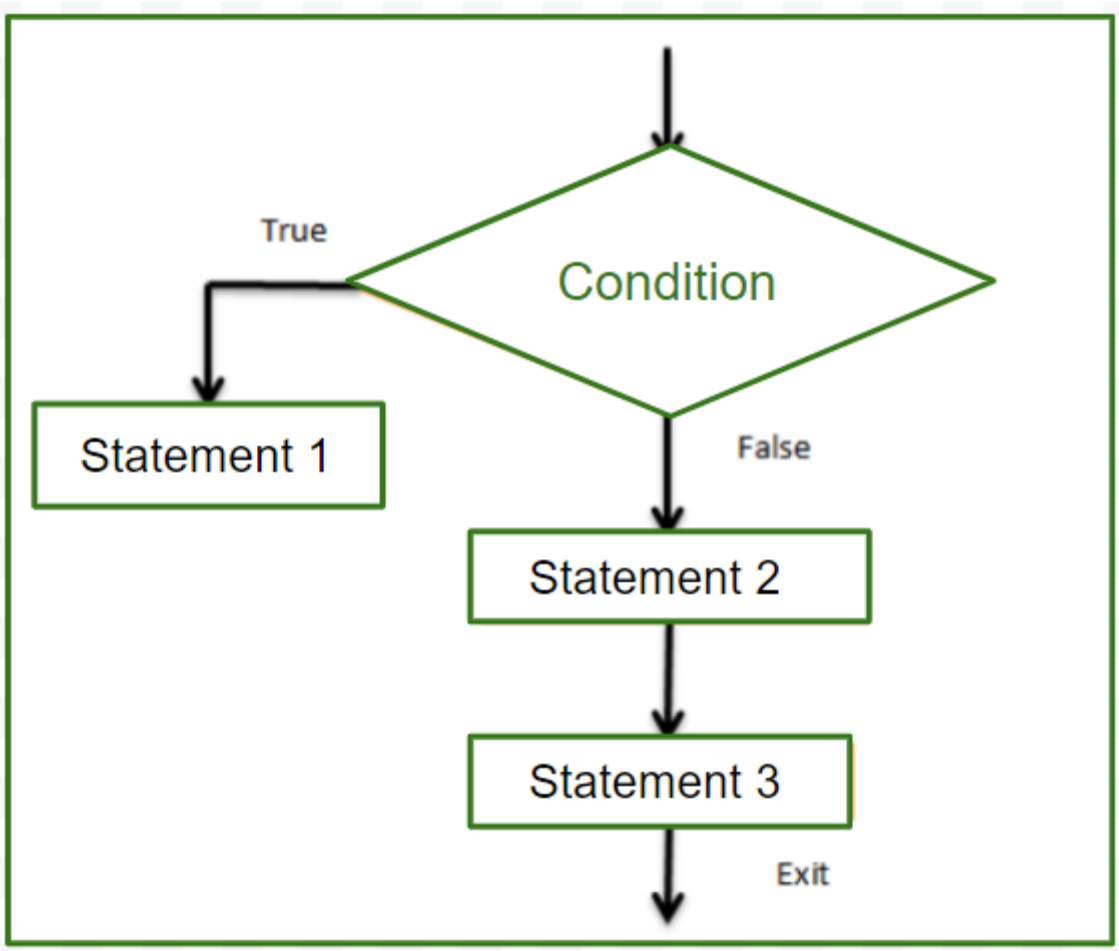
  1. Boolean Expression

  Boolean expressions are statements whose results can be either true or false. A boolean expression which is named for mathematician George Boole is an expression that evaluates to either true or false. Let's look at some common language examples:

  - My favorite color is blue. → true
  - I am afraid of mathematics. → false
  - 2 is greater than 5. → false

  2. Flow of control statements:

  The flow of control statements needs to be controlled during the execution of statements in a program. For example:



The flow of control statements

3. Labels and Gotos

The most elementary programming language construct for changing the flow of control in a program is a label and ***goto***. When a compiler encounters a statement like ***goto L***, it must check that there is exactly one statement with label ***L*** in the scope of this ***goto*** statement. If the label has already appeared, then the symbol table will have an entry giving the compiler-generated label for the first three-address instruction associated with the source statement labeled ***L***. For the translation, we generate a ***goto*** three-address statement with that compiler-generated label as a target.

When a label **L** is encountered for the first time in the source program, either in a declaration or as the target of the forward goto, we enter **L** into the symbol table and generate a symbolic table for **L**.

One-pass code generation using backpatching:

In a single pass, backpatching may be used to create a boolean expressions program as well as the flow of control statements. The synthesized properties truelist and falselist of non-terminal B are used to handle labels in jumping code for Boolean statements. The label to which control should go if B is true should be added to B.truelist, which is a list of a jump or conditional jump instructions. B.falselist is the list of instructions that eventually get the label to which control is assigned when B is false. The jumps to true and false exist, as well as the label field, are left blank when the program is generated for B. The lists B.truelist and B.falselist, respectively, contain these early jumps.

A statement S, for example, has a synthesized attribute S.nextlist, which indicates a list of jumps to the instruction immediately after the code for S. It can generate instructions into an instruction array, with labels serving as indexes. We utilize three functions to modify the list of jumps:

- **Makelist (i):** Create a new list including only i, an index into the array of instructions and the makelist also returns a pointer to the newly generated list.
- **Merge(p1,p2):** Concatenates the lists pointed to by p1, and p2 and returns a pointer to the concatenated list.
- **Backpatch (p, i):** Inserts i as the target label for each of the instructions on the record pointed to by p.

Backpatching for Boolean Expressions

Using a translation technique, it can create code for Boolean expressions during bottom-up parsing. In grammar, a non-terminal marker M creates a semantic action that picks up the index of the next instruction to be created at the proper time.

For Example, Backpatching using boolean expressions production rules table:

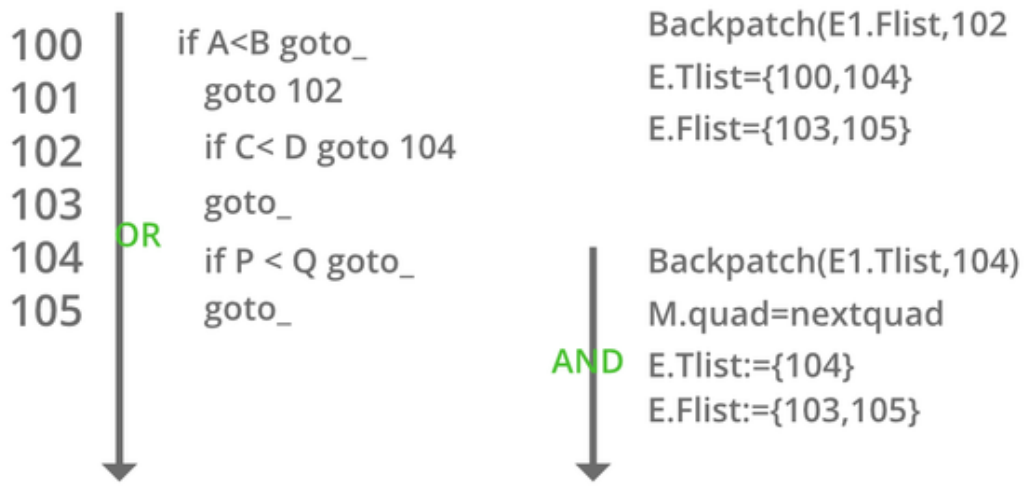
Step 1: Generation of the production table

Production Rule		Semantic action
E	E_1 OR M E_2	{backpatch (E_1.flist, M.quad); E.Tlist:=merge(E_1.Tlist, E_2.Tlist) E.Flist:= E_2.Flist}
E	E_1 AND M E_2	{backpatch (E_1.Tlist, M.quad); E.Tlist:=E_2.Tlist; E.Flist:=merge(E_1.Flist,E_2.Flist);}
E	NOT E_1	{E.Tlist:=E_1.Flist; E.Flist:=E_1.Tlist;}
E	(E_1)	{E.Tlist:=E_1.Tlist; E.Flist:=E_1.Flist;}
E	id1 relop id2	{E.Tlist:=mklist(nextstate); E.Flist:=mklist(nextstate+); Append('if' id1.place relop.op id2.place 'goto_'); Append('goto_')}
E	true	{E.Tlist:=mklist(nextstate); Append('goto_');}
E	false	{E.Flist:=mklist(nextstate); Append('goto_');}
M	ε	{m.quad:=nextquad;}

Production Table for Backpatching

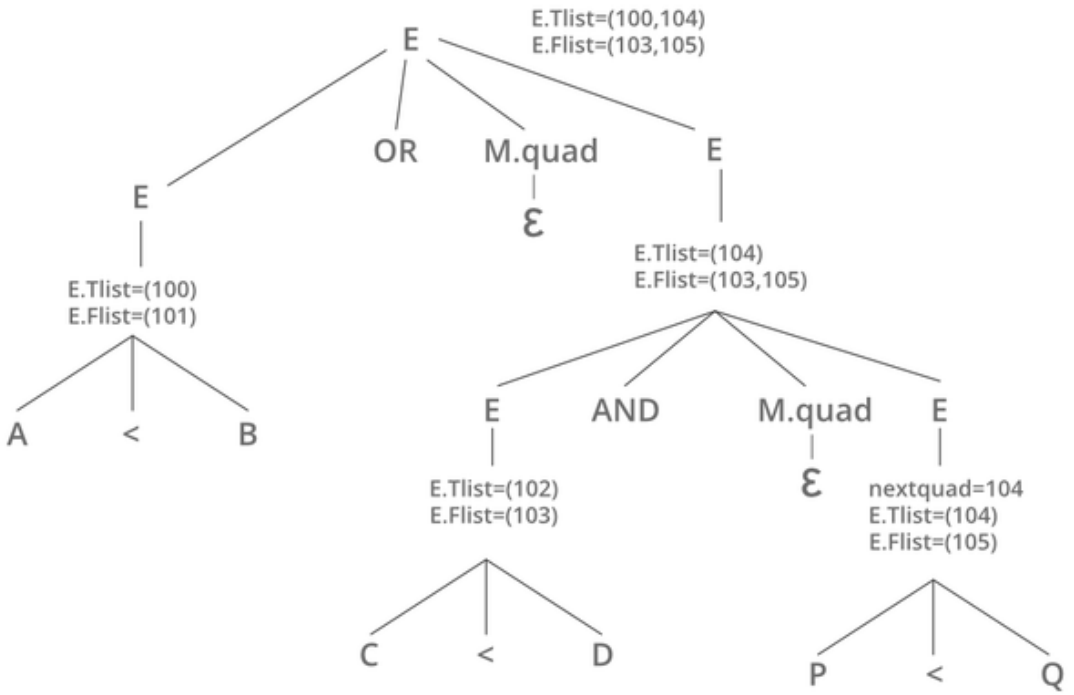
Step 2: We have to find the TAC(Three address code) for the given expression using backpatching:

A < B OR C < D AND P < Q



Three address codes for the given example

Step 3: Now we will make the parse tree for the expression:



Parse tree for the example

The flow of Control Statements:

Control statements are those that alter the order in which statements are executed. If, If-else, Switch-Case, and while-do statements are examples. Boolean expressions are often used in computer languages to

- **Alter the flow of control:** Boolean expressions are conditional expressions that change the flow of control in a statement. The value of such a Boolean statement is implicit in the program's position. For example, if (A) B, the expression A must be true if statement B is reached.
- **Compute logical values:** During bottom-up parsing, it may generate code for Boolean statements via a translation mechanism. A non-terminal marker M in the grammar establishes a semantic action that takes the index of the following instruction to be formed at the appropriate moment.

Applications of Backpatching

- Backpatching is used to translate flow-of-control statements in one pass itself.
- Backpatching is used for producing quadruples for boolean expressions during bottom-up parsing.
- It is the activity of filling up unspecified information of labels during the code generation process.
- It helps to resolve forward branches that have been planted in the code.