

## Syntax Directed Translation and Intermediate Code Generation

### What is SDD?.

Syntax Directed Definition (SDD) is a formalism used in compiler design to specify the translation of source code into target code, often accompanied by the specification of attributes. An SDD extends the context-free grammar (CFG) with semantic rules or actions associated with each production rule.

Here's an explanation of SDD with examples:

**Example 1:** Consider the following production rule:

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

In this example, **E.val** is an attribute associated with the non-terminal **E**. The semantic action **{ E.val = E1.val + T.val }** specifies that the value of **E.val** is derived from the values of **E1.val** and **T.val**.

**Example 2:** Consider the following grammar:

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

Here, **E.val**, **T.val**, and **F.val** are synthesized attributes, which means their values are computed bottom-up in the parse tree. For example, to compute **T.val**, we need the values of **T1.val** and **F.val** according to the semantic rules associated with each production.

**Synthesized Attributes:** Synthesized attributes derive their values from the values of their children nodes in the parse tree. They are computed bottom-up.

**Inherited Attributes:** Inherited attributes derive their values from their parent or sibling nodes in the parse tree. They are computed top-down.

**Computation of Synthesized Attributes:**



1. Write the SDD with appropriate semantic rules for each production in the grammar.
2. Generate the annotated parse tree.
3. Compute attribute values bottom-up in the parse tree.
4. The value obtained at the root node is the final output.

**Computation of Inherited Attributes:**

1. Construct the SDD with semantic actions.
2. Generate the annotated parse tree.
3. Compute attribute values top-down in the parse tree.

SDDs provide a high-level specification of translation rules, hiding implementation details and allowing for clear specification of attribute dependencies. They are commonly used in compiler design to specify the translation of source code into intermediate representations or target code.

**Differentiate between synthesized & inherited attributes.**

S.NO	Synthesized Attributes	Inherited Attributes
1.	An attribute is said to be Synthesized attribute if its parse tree node value is determined by the attribute value at child nodes.	An attribute is said to be Inherited attribute if its parse tree node value is determined by the attribute value at parent and/or siblings node.
2.	The production must have non-terminal as its head.	The production must have non-terminal as a symbol in its body.
3.	A synthesized attribute at node n is defined only in terms of attribute values at the children of n itself.	A 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 parse tree.	It can be evaluated during a single top-down and sideways traversal of parse tree.
5.	Synthesized attributes can be contained by both the terminals or non-terminals.	Inherited attributes can't be contained by both, It is only contained by non-terminals.
6.	Synthesized attribute is used by both S-attributed SDT and L-attributed SDT.	Inherited attribute is used by only L-attributed SDT.
7.	<p>EX:-  <math>E.val \rightarrow F.val</math></p> 	<p>EX:-  <math>E.val = F.val</math></p> 

## What is S attributed definition & L attributed definition? Explain with examples.

Before coming up to S-attributed and L-attributed SDTs, here is a brief intro to Synthesized or Inherited attributes **Types of attributes** – Attributes may be of two types – Synthesized or Inherited.

1. **Synthesized attributes** – A Synthesized attribute is an attribute of the non-terminal on the left-hand side of a production. Synthesized attributes represent information that is being passed up the parse tree. The attribute can take value only from its children (Variables in the RHS of the production). The non-terminal concerned must be in the head (LHS) of production. For e.g. let's say  $A \rightarrow BC$  is a production of a grammar, and A's attribute is dependent on B's attributes or C's attributes then it will be synthesized attribute.
2. **Inherited attributes** – An attribute of a nonterminal on the right-hand side of a production is called an inherited attribute. The attribute can take value either from its parent or from its siblings (variables in the LHS or RHS of the production). The non-terminal concerned must be in the body (RHS) of production. For example, let's say  $A \rightarrow BC$  is a production of a grammar and B's attribute is dependent on A's attributes or C's attributes then it will be inherited attribute because A is a parent here, and C is a sibling.

Now, let's discuss about S-attributed and L-attributed SDT.

### 1. S-attributed SDT :

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

### 2. L-attributed SDT:

- If an SDT uses both synthesized attributes and inherited attributes with a restriction that inherited attribute can inherit values from left siblings only, it is called as L-attributed SDT.
- Attributes in L-attributed SDTs are evaluated by depth-first and left-to-right parsing manner.
- Semantic actions are placed anywhere in RHS.
- Example :  $S \rightarrow ABC$ , Here attribute B can only obtain its value either from the parent – S or its left sibling A but It can't inherit from its right sibling C. Same goes for A & C – A can only get its value from its parent & C can get its value from S, A, & B as well because C is the rightmost attribute in the given production.

## What is intermediate code generation?

Intermediate code generation is a crucial phase in compiler design, where the front end of the compiler translates the source program into an independent intermediate representation. This intermediate code serves as a bridge between the source code and the target code, allowing for easier analysis and optimization before generating the final target code.

Here's a breakdown of intermediate code generation:

**Purpose:** The primary purpose of intermediate code generation is to create a platform-independent representation of the source code, enabling various optimizations and facilitating easier retargeting of the compiler.

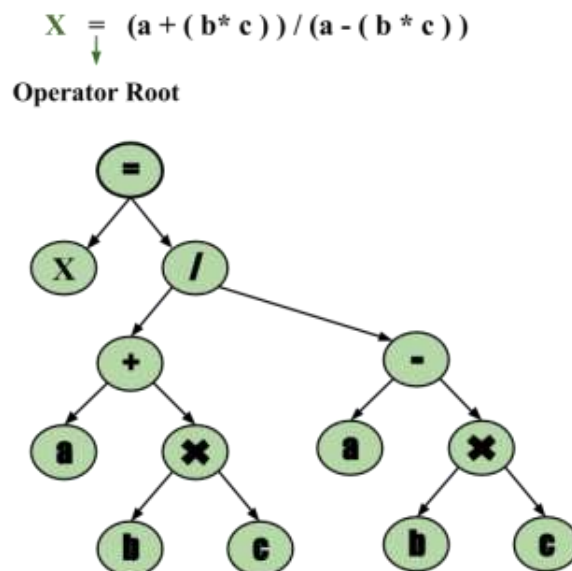
### Benefits:

1. **Portability Enhancement:** Intermediate code allows for enhanced portability of the compiler by providing a machine-independent representation of the source program. This reduces the need for separate compilers for each target machine.
2. **Retargeting Facilitation:** With intermediate code, retargeting the compiler to generate code for different architectures becomes more straightforward, as the optimizations and analysis are done on the intermediate representation.
3. **Performance Improvement:** Intermediate code generation enables the application of various optimization techniques, leading to improved performance and efficiency of the generated code.
4. **Easier Debugging:** Debugging intermediate code can be easier compared to debugging machine code or bytecode, as it retains some of the high-level constructs of the source code.

### Representation Forms:

1. **Postfix Notation:** Also known as reverse Polish notation, it eliminates the need for parentheses by positioning the operator after its operands. Example:  **$ab+c^*$** .
2. **Three-Address Code:** Each statement involves a maximum of three references (two operands and one result), facilitating easy translation into machine code. Example:  **$T1 = b * c, T2 = a + T1, T3 = T2 + d$** .

3. **Syntax Tree:** Represents the syntactic structure of the program in a condensed form, aiding in code analysis and optimization. Example:  $x = (a + b * c) / (a - b * c)$ .



#### Advantages:

- Simplification of code generation process.
- Facilitation of code optimization techniques.
- Platform independence.
- Code reuse for different platforms or languages.
- Easier debugging compared to machine code.

#### Disadvantages:

- Increased compilation time.
- Additional memory usage.
- Increased complexity of compiler design.
- Potential reduction in performance due to the overhead of intermediate code generation.

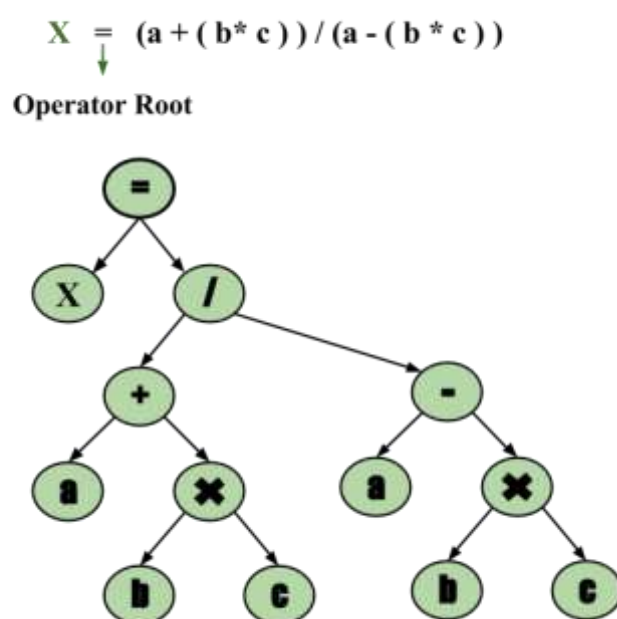
Explain following:

- a. Syntax tree
- b. Postfix notation
- c. Three address code

### 1. 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:**  $x = (a + b * c) / (a - b * c)$





## 2. 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](#)

## 3. 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  $x = y \text{ op } z$ , 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  $a + b * c + d$  :

$T1 = b * c$   $T2 = a + T1$   $T3 = T2 + d$ ;  $T1, T2, T3$  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](#)

## Differentiate between parse tree & syntax tree.

### Parse Tree :

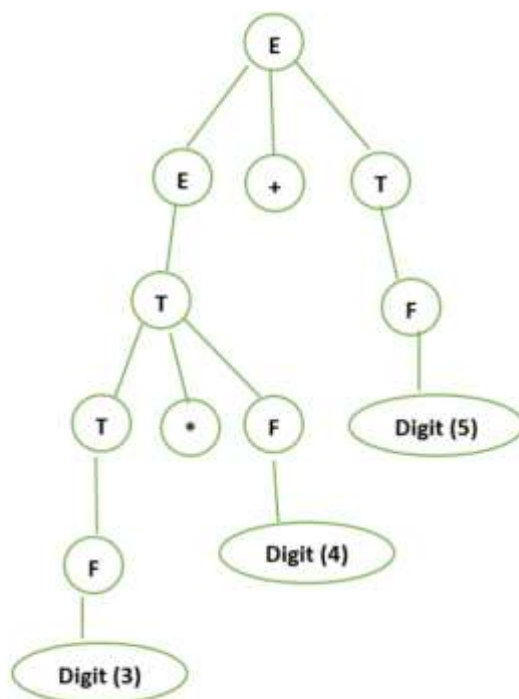
A parse tree is a visual representation of the syntactic structure of a piece of source code, as produced by a parser. It shows the hierarchy of the elements in the code and the relationships between them.

In compiler design, a parse tree is generated by the parser, which is a component of the compiler that processes the source code and checks it for syntactic correctness. The parse tree is then used by other components of the compiler, such as the code generator, to generate machine code or intermediate code that can be executed by the target machine.

[Parse trees](#) can be represented in different ways, such as a tree structure with nodes representing the different elements in the source code and edges representing the relationships between them, or as a graph with nodes and edges representing the same information. Parse trees are typically used as an intermediate representation in the compilation process, and are not usually intended to be read by humans.

### Example:

Here is the Parse tree for the expression,  $3*4+5$



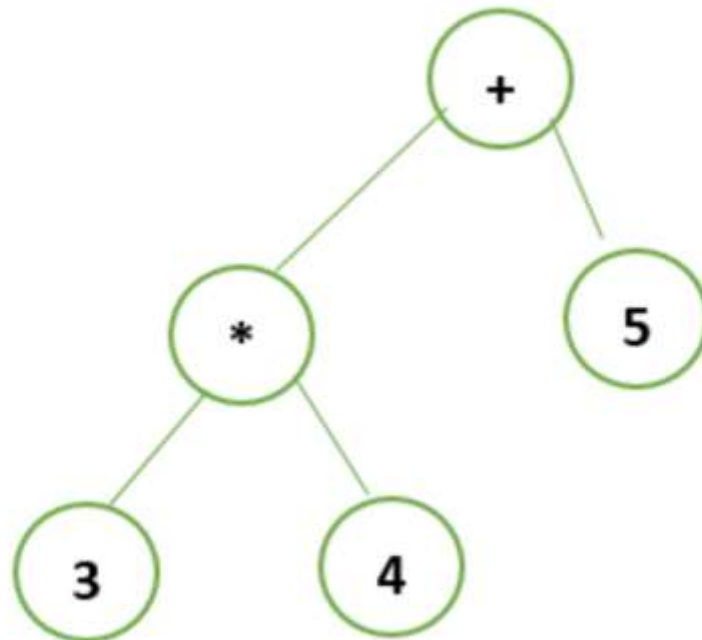
### Syntax Tree :

A syntax tree is a tree-like representation of the syntactic structure of a piece of source code. It is typically used in the process of compiler design, to represent the structure of the code in a way that is easier to analyze and manipulate.

Syntax trees are constructed by parsing the source code, which involves analyzing the code and breaking it down into its individual components, such as tokens, variables, and statements. The resulting tree is made up of nodes that correspond to these various components, with the structure of the tree reflecting the grammatical structure of the source code.

Syntax trees are useful for a variety of tasks in compiler design, such as type checking, optimization, and code generation. They can also be used to represent the structure of other types of linguistic or logical structures, such as natural language sentences or logical expressions.

Here is the Syntax tree for the expression,  $3*4+5$



## Parse Tree and Syntax Tree

Parse Tree	Syntax Tree
A parse tree is created by a parser, which is a component of a compiler that processes the source code and checks it for syntactic correctness.	A syntax tree is created by the compiler based on the parse tree after the parser has finished processing the source code.
Parse trees are typically more detailed and larger than syntax trees, as they contain more information about the source code.	Syntax trees are simpler and more abstract, as they only include the information necessary to generate machine code or intermediate code.
Parse trees are used as an intermediate representation during the compilation process.	syntax trees are the final representation used by the compiler to generate machine code or intermediate code.
Parse trees are typically represented using a tree structure with nodes representing the different elements in the source code and edges representing the relationships between them.	Syntax trees are also typically represented using a tree structure, but the nodes and edges may be arranged differently.
Parse trees can be represented in different ways, such as a tree structure, a graph, or an s-expression	Syntax trees are usually represented using a tree structure or an s-expression.
Parse trees are intended for use by the compiler and are not usually intended to be read by humans.	Syntax trees are also primarily used by the compiler, but they can also be read and understood by humans, as they provide a simplified and abstract view of the source code.

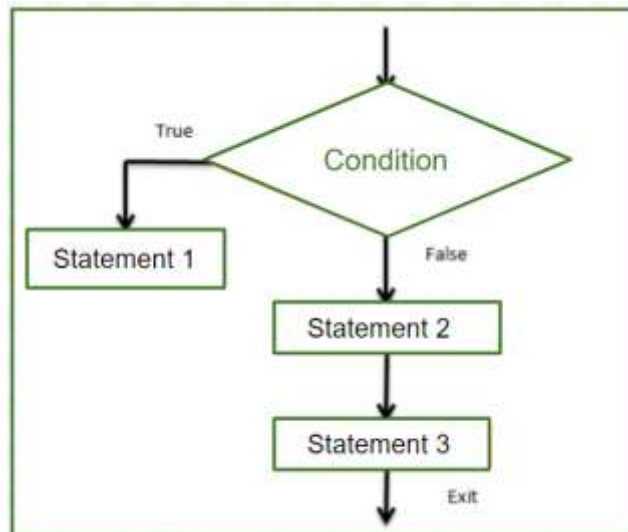
Parse Tree	Syntax Tree
<p>Parse trees include information about the source code that is not needed by the compiler, such as comments and white space.</p> <p>Parse trees may include nodes for error recovery and disambiguation, which are used by the parser to recover from errors in the source code and resolve ambiguities.</p>	<p>Syntax trees do not include this information.</p> <p>Syntax trees do not include these nodes.</p>

## What is backpatching?

Backpatching is a technique used in compiler design to handle unspecified information, particularly regarding labels, during the code generation process. It involves assigning appropriate addresses to labels, especially in the context of generating Three-Address Code (TAC) for expressions containing conditional statements like boolean expressions or flow control statements like "goto" statements.

**Purpose of Backpatching:** Backpatching serves two main purposes:

1. **Handling Boolean Expressions:** In boolean expressions, backpatching is used to manage the flow of control by determining the addresses or labels where execution should jump based on the evaluation of the expression.
2. **Flow Control Statements:** Backpatching is also essential for managing the flow of control in a program, particularly when dealing with statements like "goto" where the target label may not be known during the initial code generation phase.



**Process of Backpatching:** The process of backpatching typically involves two passes:

1. **First Pass:** During the first pass of code generation, the compiler may encounter forward branches or conditional statements where the target labels are not yet defined. Instead of attempting to resolve these branches immediately, the compiler leaves placeholders or empty spaces for the addresses.

2. **Second Pass:** In the second pass, after all labels have been defined, the compiler goes back and fills in the previously unspecified addresses or labels. This process of updating the code to replace placeholders with actual addresses is referred to as backpatching.

#### **Application of Backpatching:**

1. **Flow-of-Control Statements:** Backpatching is extensively used to handle flow-of-control statements like "goto" or conditional jumps in a program. It ensures that the correct addresses or labels are used for jumps, even if they are defined later in the code.
2. **Boolean Expressions:** Backpatching is crucial for generating code for boolean expressions during bottom-up parsing. It helps in determining the appropriate labels for true and false conditions, ensuring the correct flow of execution based on the evaluation of the expression.
3. **One-Pass Code Generation:** Backpatching allows for one-pass code generation by deferring the assignment of addresses until all labels are known. This simplifies the code generation process and enables efficient handling of forward references.



## Write Syntax Directed Translation Scheme for Assignment Statements.

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

### Translation of Assignment Statements

In the syntax directed translation, assignment statement is mainly deals with expressions. The expression can be of type real, integer, array and records.

Consider the grammar

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

The translation scheme of 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 for appending the three address code to the output file. Otherwise it will report an error.
- The newtemp() is a function used to generate new temporary variables.
- E.place holds the value of E.

### Write Syntax Directed Translation Scheme for Boolean Expression.

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

Boolean expressions have two primary purposes. They are used for computing the logical values. They are also used as conditional expression using if-then-else or while-do.

Consider the grammar

1.  $E \rightarrow E \text{ OR } E$
2.  $E \rightarrow E \text{ AND } E$
3.  $E \rightarrow \text{NOT } E$
4.  $E \rightarrow (E)$
5.  $E \rightarrow \text{id rel op id}$
6.  $E \rightarrow \text{TRUE}$
7.  $E \rightarrow \text{FALSE}$

The rel op is denoted by  $<, >, <=, >=$ .

The AND and OR are left associated. NOT has the higher precedence then AND and lastly OR.

Production rule	Semantic actions
$E \rightarrow E1 \text{ OR } E2$	$(E.place = \text{newtemp}());$ $\text{Emit } (E.place := 'E1.place \text{OR} E2.place')$ $\}$
$E \rightarrow E1 \text{ AND } E2$	$(E.place = \text{newtemp}());$ $\text{Emit } (E.place := 'E1.place \text{AND} E2.place')$ $\}$
$E \rightarrow \text{NOT } E1$	$(E.place = \text{newtemp}());$ $\text{Emit } (E.place := 'NOT E1.place')$ $\}$
$E \rightarrow (E1)$	$(E.place = E1.place)$
$E \rightarrow \text{id rel op id2}$	$(E.place = \text{newtemp}());$ $\text{Emit } (' \text{if id1.place rel op id2.place goto}$ $\text{nextstar} + 3);$ $\text{EMIT } (E.place := '0');$ $\text{EMIT } (' \text{goto} \text{ nextstar} + 2);$ $\text{EMIT } (E.place := '1');$ $\}$
$E \rightarrow \text{TRUE}$	$(E.place := \text{newtemp}());$ $\text{Emit } (E.place := '1')$ $\}$
$E \rightarrow \text{FALSE}$	$(E.place := \text{newtemp}());$ $\text{Emit } (E.place := '0')$ $\}$

The EMIT function is used to generate the three address code and the newtemp( ) function is used to generate the temporary variables.

The  $E \rightarrow \text{id rel op id2}$  contains the next\_state and it gives the index of next three address statements in the output sequence.

Here is the example which generates the three address code using the above translation scheme:

1.  $p > q$  AND  $r < s$  OR  $u > r$

100: **if**  $p > q$  **goto** 103

101:  $t1 := 0$

102: **goto** 104

103:  $t1 := 1$

104: **if**  $r < s$  **goto** 107

105:  $t2 := 0$

106: **goto** 108

107:  $t2 := 1$

108: **if**  $u > r$  **goto** 111

109:  $t3 := 0$

110: **goto** 112

111:  $t3 := 1$

112:  $t4 := t1$  AND  $t2$

113:  $t5 := t4$  OR  $t3$

