UNIT – 3 : SYNTAX-DIRECTED TRANSLATION

Syntax Directed Translation:

In syntax directed translation, along with the grammar we associate some informal notations and these notations are called as semantic rules.

So, we can say that

Grammar + semantic rule = SDT (syntax directed translation)

- In syntax directed translation, every non-terminal can get one or more than one attribute or sometimes 0 attribute depending on the type of the attribute. The value of these attributes is evaluated by the semantic rules associated with the production rule.
- In the semantic rule, attribute is VAL and an attribute may hold anything like a string, a number, a memory location and a complex record
- In Syntax directed translation, whenever a construct encounters in the programming language then it is translated according to the semantic rules define in that particular programming language.

Example:

Production	**Semantic Rules**
E → E + T	E.val := E.val + T.val
E → T	E.val := T.val
T → T * F	T.val := T.val + F.val
T → F	T.val := F.val
F → (F)	F.val := F.val
F → num	F.val := num.lexval

E.val is one of the attributes of E.

num.lexval is the attribute returned by the lexical analyzer.

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}

Implementation of Syntax Directed Translation:

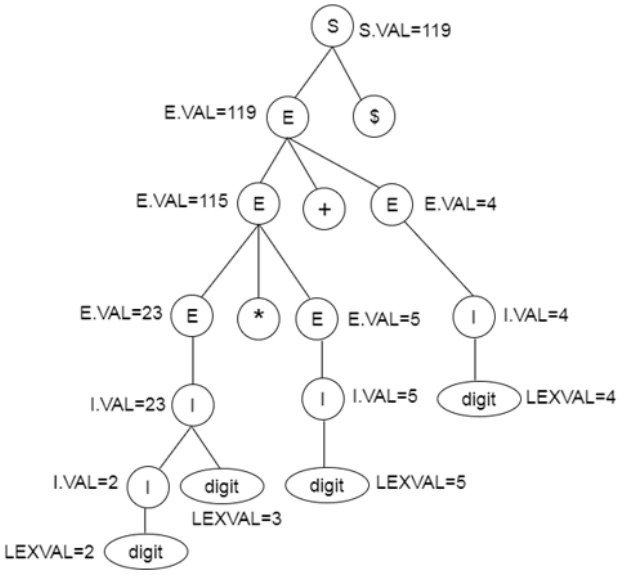
Syntax direct translation is implemented by constructing a parse tree and performing the actions in a left to right depth first order.

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}

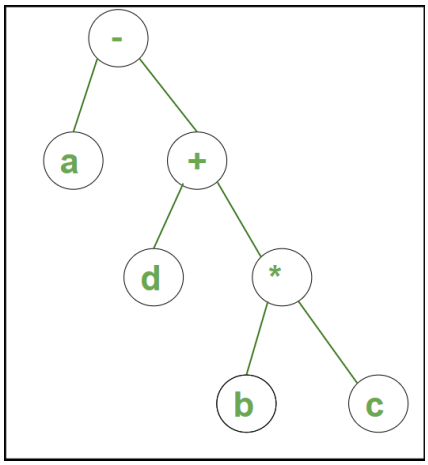
Parse tree for SDT:



Construction of Syntax Trees:

A syntax tree is a tree in which each leaf node represents an operand, while each inside node represents an operator. The Parse Tree is abbreviated as the syntax tree. The syntax tree is usually used when representing a program in a tree structure.

Example: Syntax Tree for the string **a – b ∗ c + d** is:



Rules of Constructing a Syntax Tree:

A syntax tree's nodes can all be performed as data with numerous fields. One element of the node for an operator identifies the operator, while the remaining field contains a pointer to the operand nodes. The operator is also known as the node's label. The nodes of the syntax tree for expressions with binary operators are created using the following functions. Each function returns a reference to the node that was most recently created.

1. mknode (op, left, right): It creates an operator node with the name op and two fields, containing left and right pointers.

2. mkleaf (id, entry): It creates an identifier node with the label id and the entry field, which is a reference to the identifier's symbol table entry.

3. mkleaf (num, val): It creates a number node with the name num and a field containing the number's value, val. Make a syntax tree for the expression a 4 + c, for example. p1, p2,…, p5 are pointers to the symbol table entries for identifiers 'a' and 'c', respectively, in this sequence.

Example:

p_1– mkleaf (id, entry a);

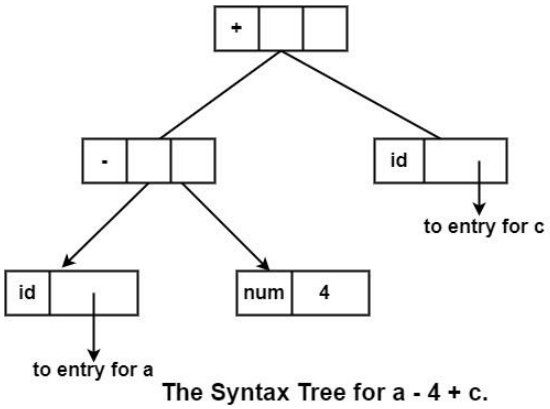
p_2– mkleaf (num, 4);

p_3– mknode ('–', p_1, p_2)

p_4– mkleaf(id, entry c)

p_5– mknode('+', p_3, p_4);

The tree is generated in a bottom-up fashion. The function calls mkleaf (id, entry a) and mkleaf (num 4) construct the leaves for a and 4. The pointers to these nodes are stored using $p_1$and p_2. The call mknodes ('–', p_1, p_2) then make the interior node with the leaves for a and 4 as children. The syntax tree will be

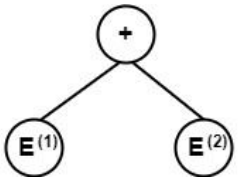


The Syntax Tree for a - 4 + c.

Syntax Directed Translation of Syntax Trees:

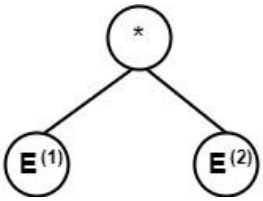
Production	Semantic Action
$E \rightarrow E^{(1)} + E^{(2)}$	{E. VAL = Node (+, $E^{(1)}$. VAL, $E^{(2)}$. VAL)}
$E \rightarrow E^{(1)} * E^{(2)}$	{E. VAL = Node (*, $E^{(1)}$. VAL, $E^{(2)}$. VAL)})
$E \rightarrow (E^{(1)})$	{E. VAL = $E^{(1)}$. VAL}
$E \rightarrow E^{(1)}$	{E. VAL = UNARY (–, $E^{(1)}$.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

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

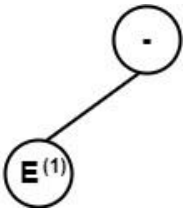


$E^{(1)}$. VAL &$E^{(2)}$. VAL **are left & right children of this node.**

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.

S – Attributed and L – Attributed:

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 -> 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 -> 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.

S-Attributed SDT:

The **S-attributed** definition is a type of syntax-directed attributes in compiler design that solely uses synthesized attributes. The symbol attribute values in the production's body are used to calculate the attribute values for the non-terminal at the head.

The nodes of the parse tree can be ranked from the bottom up when evaluating an S-attributed SDD's attributes. i.e., by conducting a post-order traverse of the parse tree and evaluating the characteristics at a node once the traversal finally leaves that node.

Let us see an example of S-attributed SDT.

Example

The grammar is given below:

S → E

E → E1+T

E → T

T → T1*F

T → F

F → digit

The S-attributed SDT of the above grammar can be written in the following way.

Production	**Semantic Rules**
S → E	S.val = E.val
E → E1 + T	E.val = E1.val + T.val
E → T	E.val = T.val
T → T1 * F	T.val = T1.val * F.val
T → F	T.val = F.val
F → digit	F.val = digit. lexval

L-Attributed SDT:

L-attributed definitions are syntax-directed attributes in compiler design in which the edges of the dependency graph for the attributes in the production body can go from left to right and not from right to left. L-attributed definitions can inherit or synthesize their attributes.

If the traits are inherited, the calculation must come from the following:

- A quality that the production head inherited.
- By a production-related attribute, either inherited or synthesized, situated to the left of the attribute being computed.
- An inherited or synthesized attribute is linked to the attribute in question in a way that prevents cycles from forming in the dependency network.

Let us see an example of L-attributed SDT.

Example

The grammar is given below:

G → TL

T → int

T → float

T → double

L → Ll, id

L → id.

The L-attributed SDT of the above grammar can be written in the following way.

Production	**Semantic Rules**
G → T L	L.in = T.type
T → int	T.type = int
T → float	T.type = float
T → double	T.type = double
L → L1, id	L1.in = L.in Enter_type(id. entry, L.in)
L → id	Entry_type(id. entry, L.in)

Attributes Flow in Compiler Design:

A **grammar** is well-defined if the rules produce a distinct set for every conceivable parse tree. Grammars are **declarative notations** that do not require an ordering. If an attribute does not depend on itself, the grammar is **noncircular**.

An **S-attributed grammar** can be decorated in the same manner as an **LR parser**, enabling the interleaving of parsing and attribute evaluation in a single pass.

An **L-attributed grammar** can be decorated like an **LL parser**, enabling the interleaving of parsing and attribute evaluation in a single pass.

Top-Down Translation:

Top-down translation in syntax-directed translation (SDT) is a method where the parse tree is generated from the root (starting symbol) down to the leaves (terminals) using a top-down approach. This method is typically integrated with parsing techniques such as recursive descent parsing.

In top-down translation, the semantic actions are executed during the parsing process as soon as the corresponding non-terminal or terminal is recognized. This immediate action allows for the translation to proceed in tandem with the parsing.

The semantic actions often involve computing attributes, which can be either synthesized or inherited. In top-down translation, inherited attributes are particularly important since they can be passed down from parent nodes to child nodes during the parse.

Typically, top-down translation is implemented using recursive descent parsers, where each non-terminal in the grammar has a corresponding recursive function. Semantic actions are embedded within these functions.

Example:

Consider a simple arithmetic expression grammar:

E → T E'

E' → + T { print('+') } E' | ε

T → int { print(int.value) }

In this example:

- The non-terminal E is expanded to T E', and E' handles the + operation.
- The semantic action { print('+') } in E' is executed when the + is recognized, immediately translating this part of the input.
- The terminal int is printed as soon as it is recognized.

Advantages:

- **Immediate Translation:** Actions are performed as soon as corresponding components are recognized, allowing for real-time processing.
- **Simplicity:** For certain grammars, this method can be simple and intuitive, especially when using recursive descent parsing.

Challenges:

- **Left Recursion:** Top-down parsers struggle with left-recursive grammars, which must be converted into a non-left-recursive form.
- **Backtracking:** Without predictive parsing (LL(1)), the parser may need to backtrack, complicating the translation process.

Intermediate Code Forms Using Postfix Notation:

Intermediate Code:

Intermediate code is used to translate the source code into the machine code. Intermediate code lies between the high-level language and the machine language.

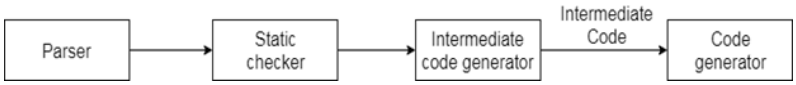


Fig: Position of intermediate code generator

- If the compiler directly translates source code into the machine code without generating intermediate code then a full native compiler is required for each new machine.
- The intermediate code keeps the analysis portion same for all the compilers that's why it doesn't need a full compiler for every unique machine.
- Intermediate code generator receives input from its predecessor phase and semantic analyzer phase. It takes input in the form of an annotated syntax tree.
- Using the intermediate code, the second phase of the compiler synthesis phase is changed according to the target machine.

Intermediate Representation:

Intermediate code can be represented in two ways:

1. High Level intermediate code: High level intermediate code can be represented as source code. To enhance performance of source code, we can easily apply code modification. But to optimize the target machine, it is less preferred.

2. Low Level intermediate code: Low level intermediate code is close to the target machine, which makes it suitable for register and memory allocation etc. it is used for machine-dependent optimizations.

Postfix Notation:

- Postfix notation is the useful form of intermediate code if the given language is expressions.
- Postfix notation is also called as 'suffix notation' and 'reverse polish'.
- Postfix notation is a linear representation of a syntax tree.
- In the postfix notation, any expression can be written unambiguously without parentheses.
- The ordinary (infix) way of writing the sum of x and y is with operator in the middle: x * y. But in the postfix notation, we place the operator at the right end as xy *.
- In postfix notation, the operator follows the operand.

Example:

Consider the arithmetic expression:

a * (b + c) - d / e

Step-by-Step Conversion to Postfix Notation:

1. b + c becomes b c +
2. a * (b + c) becomes a b c + *
3. d / e becomes d e /
4. The entire expression becomes a b c + * d e / -

Stack-Based Evaluation:

To evaluate a postfix expression, a stack is used. Operands are pushed onto the stack, and when an operator is encountered, the required number of operands is popped from the stack, the operation is performed, and the result is pushed back onto the stack.

Example:

push a

push b

push c

add

mul

push d

push e

div

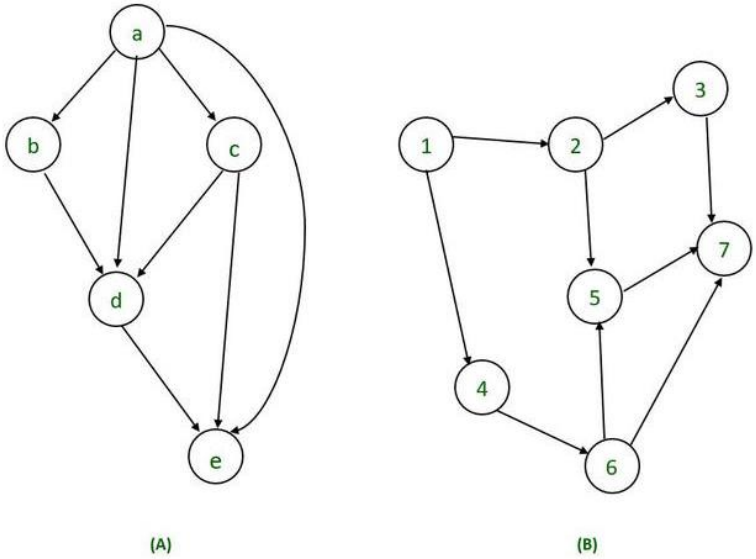
sub

Directed Acyclic Graph (DAG):

The Directed Acyclic Graph (DAG) is used to represent the structure of basic blocks, to visualize the flow of values between basic blocks, and to provide optimization techniques in the basic block. To apply an optimization technique to a basic block, a DAG is a three-address code that is generated as the result of an intermediate code generation.

- Directed acyclic graphs are a type of data structure and they are used to apply transformations to basic blocks.
- The Directed Acyclic Graph (DAG) facilitates the transformation of basic blocks.
- DAG is an efficient method for identifying common sub-expressions.
- It demonstrates how the statement's computed value is used in subsequent statements.

Examples of directed acyclic graph :



Algorithm for construction of DAG:

For constructing a DAG, the input and output are as follows.

Input- The input will contain a basic block.

Output- The output will contain the following information-

- Each node of the graph represents a label.
  - If the node is a leaf node, the label represents an identifier.
  - If the node is a non-leaf node, the label represents an operator.
- Each node contains a list of attached identifiers to hold the computed value.

There are three possible scenarios on which we can construct a DAG.

1. **Case 1: x = y op z**

   where x, y, and z are operands and op is an operator.

2. **Case 2: x = op y**

   where x and y are operands and op is an operator.

3. **Case 3: x = y**

   where x and y are operands.

Now, we will discuss the steps to draw a DAG handling the above three cases.

Steps:

To draw a DAG, follow these three steps.

1. **Step 1:** According to step 1,
   1. If, in any of the three cases, the y operand is not defined, then create a node(y).
   2. If, in case 1, the z operand is not defined, then create a node(z).
2. **Step 2:** According to step 2,
   1. For case 1, create a node(op) with node(y) as its left child and node(z) as its right child. Let the name of this node be n.
   2. For case 2, check whether there is a node(op) with one child node as node(y). If there is no such node, then create a node.
   3. For case 3, node n will be node(y).
3. **Step 3:** For a node(x), delete x from the list of identifiers. Add x to the list of attached identifiers list found in step 2. At last, set node(x) to n.

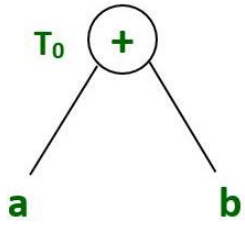
Example:

T0 = a + b —Expression 1

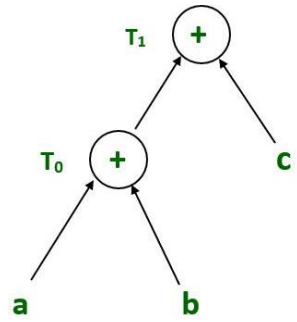
T1 = T0 + c —-Expression 2

d = T0 + T1 ——Expression 3

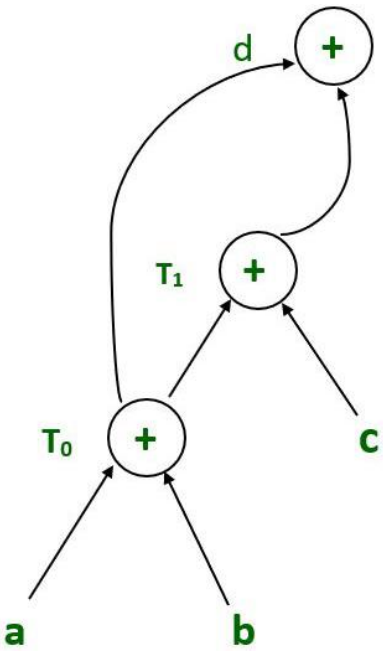
Expression 1 : T0 = a + b



Expression 2: T1 = T0 + c



Expression 3 : d = T0 + T1



Application of Dag in Compiler Design:

The applications of Directed Acyclic Graph (DAG) in Compiler Design are:

- **Expression Optimization:** DAGs are used to optimize expressions by identifying common subexpressions and representing them efficiently, reducing redundant computations.
- **Code Generation:** DAGs assist in generating efficient code by representing intermediate code and optimizing it before translating it into machine code.
- **Register Allocation:** DAGs aid in register allocation by identifying variables that can share the same register, minimizing the number of memory accesses.
- **Control Flow Analysis:** DAGs help in analyzing control flow structures and optimizing the flow of control within a program.

Advantages of Dag in Compiler Design:

The advantages of DAG in Compiler Design are:

- **Space Efficiency:** DAGs minimize memory usage by representing common subexpressions and eliminating redundancy in intermediate representations.
- **Time Efficiency:** DAG-based optimizations reduce the time required for expression evaluation and code generation, leading to faster compilation times.
- **Improved Code Quality:** By eliminating redundant computations and optimizing expressions, DAGs produce more efficient and optimized code, enhancing overall code quality and performance.
- **Simplifies Optimization Passes:** DAG-based optimizations simplify the implementation of optimization passes in compilers by providing a structured representation of intermediate code.
- **Facilitates Register Allocation:** DAGs aid in register allocation by providing insights into the usage of variables and their lifetimes, enabling efficient allocation of hardware resources.

Three Address Code:

- Three-address code is an intermediate code. It is used by the optimizing compilers.
- In three-address code, the given expression is broken down into several separate instructions. These instructions can easily translate into assembly language.
- Each Three address code instruction has at most three operands. It is a combination of assignment and a binary operator.

Example:

Given Expression: a := (-c * b) + (-c * d)

Three-address code is as follows:

t_1 := -c

t_2 := b*t_1

t_3 := -c

t_4 := d * t_3

t_5 := t_2 + t_4

a := t_5

t is used as registers in the target program.

General Representation:

a = b

a = op b

a = b op c

Where a, b or c represents operands like names, constants or compiler generated temporaries and op represents the operator

Three Address Code is Used in Compiler Applications:

1. **Optimization:** Three address code is often used as an intermediate representation of code during 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.
2. **Code generation:** Three address code 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 that is specific to the target platform, while also ensuring that the generated code is correct and efficient.
3. **Debugging:** Three address code can be helpful in debugging the code generated by the compiler. Since three 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 code to trace the execution of the program and identify errors or issues that may be present.

4. **Language translation:** Three address code can also be used to translate code from one programming language to another. By translating code to a common intermediate representation, it becomes easier to translate the code to multiple target languages.

TAC For Various Control Structures:

1. Conditional Statements (if-else):

For conditional statements like if-else, TAC uses conditional and unconditional jumps (or "goto" statements) to represent the flow of control.

Example:

```
if (a < b) {
   x = y + z;
} else {
   x = y - z;
}
```

TAC:

```
t1 = a < b
if t1 goto L1
x = y - z
goto L2
L1: x = y + z
L2:
```

Here, t1 is a temporary variable storing the result of the comparison a < b. The code then uses conditional and unconditional jumps to determine whether to execute the if block or the else block.

2. Loops (while):

For loops like while, TAC typically involves labels and conditional jumps to repeatedly execute a block of code until the condition is no longer true.

Example:

```
while (a < b) {
    a = a + 1;
}
```

TAC:

```
L1: t1 = a < b
if not t1 goto L2
a = a + 1
goto L1
L2:
```

Here, the code starts with label L1, checks the condition a < b, and if it's false, jumps to L2 to exit the loop. If true, it executes the loop body, then jumps back to L1 to recheck the condition.

3. For Loop:

The for loop is similar to the while loop but includes initialization, condition checking, and an update expression.

Example:

```
for (i = 0; i < n; i++) {
    a = a + i;
}
```

TAC:

```
i = 0
L1: t1 = i < n
if not t1 goto L2
a = a + i
i = i + 1
goto L1
L2:
```

The loop initializes i, checks if i < n, performs the loop body, increments i, and repeats until the condition fails.

4. Switch-Case Statement:

Switch-case statements are handled using a series of conditional jumps.

Example:

```
switch (a) {
   case 1: x = x + 1; break;
   case 2: x = x + 2; break;
   default: x = x + 3;
}
```

TAC:

```
if a == 1 goto L1
if a == 2 goto L2
goto L3
L1: x = x + 1
goto L4
L2: x = x + 2
goto L4
L3: x = x + 3
L4:
```

Each case is translated into a conditional jump, followed by a jump to skip the rest of the cases once a match is found.

5. Function Calls:

Function calls involve setting up the parameters, calling the function, and storing the return value.

Example:

```
x = foo(a, b);
```

TAC:

param a

param b

t1 = call foo, 2

x = t1

Here, param sets up the arguments for the function call, call foo, 2 invokes the function foo with 2 parameters, and t1 stores the return value.

6. Return Statement:

The return statement typically involves a jump to the function's exit point.

Example:

return x + y;

TAC:

t1 = x + y

return t1

7. Do-While Loop:

The do-while loop ensures the loop body executes at least once.

Example:

```
do {
    a = a + 1;
} while (a < b);
```

TAC:

L1: a = a + 1

t1 = a < b

if t1 goto L1

Representation of Three Address Code:

There are 3 representations of three address code namely

1. Quadruple
2. Triples
3. Indirect Triples

1. Quadruple:

It is a structure which consists of 4 fields namely op, arg1, arg2 and result. op denotes the operator and arg1 and arg2 denotes the two operands and result is used to store the result of the expression.

Advantage –

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

Disadvantage –

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

Example – Consider expression a = b * – c + b * – c. The three address code is:

t1 = uminus c (Unary minus operation on c)

t2 = b * t1

t3 = uminus c (Another unary minus operation on c)

t4 = b * t3

t5 = t2 + t4

a = t5 (Assignment of t5 to a)

#	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 make use of extra temporary variable to represent a single operation instead when a reference to another triple's value is needed, a pointer to that triple is used. So, it consists of only three fields namely op, arg1 and arg2.

Disadvantage –

- Temporaries are implicit and difficult to rearrange code.
- It is difficult to optimize because optimization involves moving intermediate code. When a triple is moved, any other triple referring to it must be updated also. With help of pointer one can directly access 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 makes use of pointer to the listing of all references to computations which is made separately and stored. Its similar in utility as compared to quadruple representation but requires less space than it. Temporaries are implicit and easier to rearrange code.

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

List of pointers to table

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

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

Indirect Triples representation

Boolean Expression:

A Boolean Expression is an expression that evaluates to either true or false. We generally use Boolean Expressions in conditional statements and loops in programming languages. It can be a simple Boolean Variable or a combination of Boolean Variables, Constants, and Operators.

Examples of Boolean Operators include “and,” “or,” and “not.”

In programming languages, we write boolean expressions in some specific syntax. For example, we represent “and” as “&&” in C++ and Java.

In Compiler Design, we use parsing to analyze the Boolean Expression as it allows the Compiler to understand the structure of the Expression.

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 -

E → E OR E

E → E AND E

E → NOT E

E → (E)

E → id relop id

E → TRUE

E → FALSE

The relop 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 → E1 OR E2	{E.place = newtemp(); Emit (E.place ':=' E1.place 'OR' E2.place) }
E → E1 + E2	{E.place = newtemp(); Emit (E.place ':=' E1.place 'AND' E2.place) }
E → NOT E1	{E.place = newtemp(); Emit (E.place ':=' 'NOT' E1.place) }

E → (E1)	{E.place = E1.place}
E → id relop id2	{E.place = newtemp(); Emit ('if' id1.place relop.op id2.place 'goto' nextstar + 3); EMIT (E.place ':=' '0') EMIT ('goto' nextstat + 2) EMIT (E.place ':=' '1') }
E → TRUE	{E.place := newtemp(); Emit (E.place ':=' '1') }
E → FALSE	{E.place := newtemp(); 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 → id relop 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:

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>v **goto** 111

109: t3:=0

110: **goto** 112

111: t3:= 1

112: t4:= t1 AND t2

113: t5:= t4 OR t3

Flow-of-Control Statements:

In this method, we translate the Boolean Expression into the three-address code in terms of if-then statements, if-then-else statements, and while-do statements. Here, Boolean Expressions are denoted by 'E' and the three-address statement is symbolically labelled.

These are some points you need to keep in mind while determining the Syntax-Directed Definition:

- The Control will go to the E.true label, if the Expression 'E' will be true and the Control will go to the E.false label, if the Expression 'E' will be false.
- The Control will flow from S.code to the three-address instruction.
- The label S.next is the first three-address instruction that needs to be executed after the S.code.

Syntax-Directed Definition for Flow-of-Control Statements

Production	**Semantic Rules**
S -> **if** E **then** S1	E.true : = newlabel; E.false : = S.next; S1.next : = S.next; S.code : = E.code || gen(E.true ':') || S1.code
S -> **if** E then S1 **else** S2	E.true : = newlabel; E.false : = newlabel; S1.next : = S.next; S2.next : = S.next; S.code : = E.code || gen(E.true ':') || get('goto' S.next) || gen(E.false ':') || S1.code || S2.code
S -> **While** E **do** S1	S.begin : = newlabel; E.true : = newlabel; E.false : = S.next; S1.next : = S.begin; S.code : = gen(S.begin ':') || E.code || gen(E.true ':') || S1.code || gen('goto' S.begin)

What is Backpatching?

It is a technique that involves inserting addresses or labels into the code to represent the jump to be made so that the control will go to that labelled address. To manipulate the list of labels, we use three functions:

makelist(i): It creates a new list containing only ‘i’ and returns a pointer to the list it has made.

merge(p1, p2): It concatenates the lists pointed by p1 and p2 and returns a pointer to the concatenated list.

backpatch(p, i): It inserts ‘i’ as the target label for each of the statements on the list pointed by ‘p’.

There are some notations that you need to keep in mind to understand the translation scheme of Backpatching.

- The attributes ‘truelist’ and ‘falselist’ are used to generate the jumping code for the Boolean Expression.
- The variable ‘nextquad’ holds the index of the next quadruple to follow.
- The attribute ‘M.quad’ records the number of the first statement of E.code

Translation Scheme of Backpatching:

Production	Semantic Rules

E -> E1 OR M E2	{
	backpatch (E1.falselist, M.quad);
	E.truelist : = merge(E1.truelist, E2.truelist);
	E.falselist : = E2.falselist
	}

E -> E1 AND M E2	{
	backpatch(E1.truelist, M.quad);
	E.truelist : = E2.truelist;
	E.falselist : = merge(E1.falselist, E2.falselist);
	}

E -> NOT E1	{
	E.truelist = E1.falslist;
	E.falselist = E1.truelist;
	}

E -> (E1)	{ E.truelist = E1.truelist; E.falselist = E1.falselist; }
E -> id1 relop id2	{ E.truelist : = makelist(nextquad); E.falselist : = makelist(nextquad + 1); emit('if' id1.place relop.op id2.place 'goto_') emit('goto_') }
E -> true	{ E.truelist : = makelist(nextquad); emit('goto_') }
E -> false	{ E.falselist : = makelist(nextquad); emit('goto_') }
M -> ε	{ M.quad : = nextquad }

Control Structures / Control Flow Statements:

Control flow statements are essential components of programming languages that allows executing code based on a decision. The typically available control flow statements are if-then, if- then-else, while-do statement and do – while statements. The control flow statements have an expression that needs to be evaluated and based on the true or false value of the expression the appropriate branching is taken. The context free grammar for defining control flow statements can be defined as

S à if E then S1 | if E then S1 else S2 | while E do S1 | do S1 while E

The LHS symbol 'S' stands for statement. The production defines a statement could be a simple "if Expression then Statement" or "if Expression then statement else alternate statement", "while Expression is true do the statements repeatedly" or do a statement repeatedly while the expression is true. In all these statements "E" corresponds to a Boolean expression or sometimes it could be an assignment statement.

Thus, to generate three-address code for a control flow statement, the first step is to generate code for the expression. This expression could be a sequence of expressions combined with relational and logical operators. Let us discuss each type of control flow statements and the semantic rules used for generating three-address code in the subsequent sections. (Explained in **TAC for Various Control Structures).**