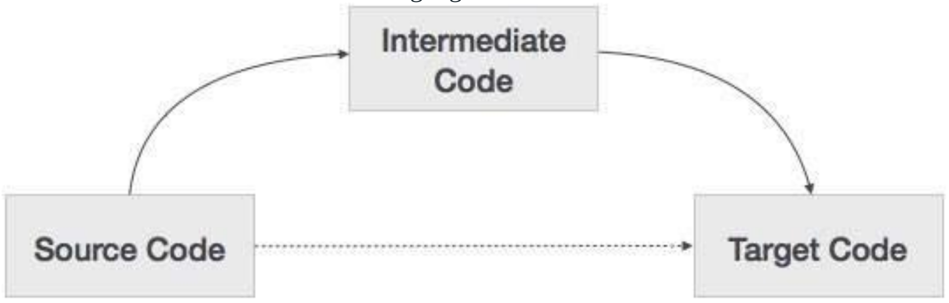
18CSC304J – COMPILER DESIGN
UNIT – IV

Topics: Intermediate Code Generation, Intermediate Languages **-** prefix – postfix, Quadruple **-** triple **-** indirect triples, Representation**-** Syntax tree**-** Evaluation of expression –three address code, Synthesized attributes – Inherited attributes**-** Intermediate languages – Declarations, Assignment Statements**-** Boolean Expressions, Case Statements, Back patching – Procedure calls.
Code Generation, Issues in the design of code generator **-** The target machine – Runtime Storage management, A simple Code generator**-** Code Generation Algorithm, Register and Address Descriptors**-** Generating Code of Assignment Statements, Cross Compiler – T diagrams**-** Issues in Cross compilers.

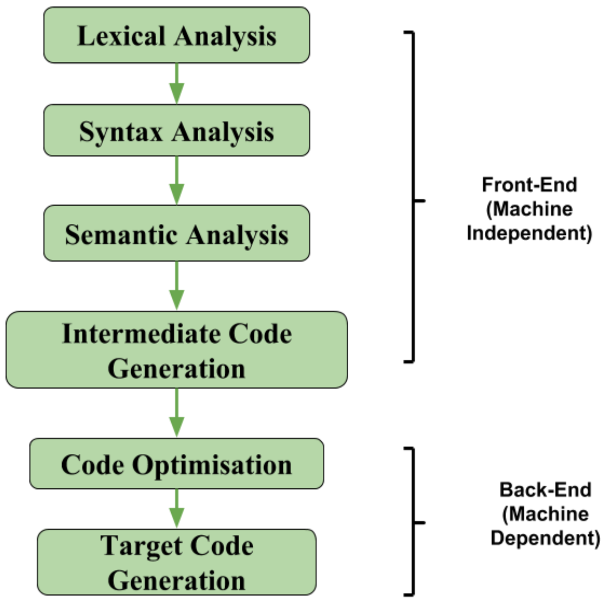
Textbook: Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman, "*Compilers: Principles, Techniques, and Tools*" Addison-Wesley, 1986.

INTERMEDIATE CODE GENERATION:

Intermediate code generator receives input from its predecessor phase, semantic analyser, in the form of an annotated syntax tree. That syntax tree then can be converted into a linear representation, e.g., postfix notation. Intermediate Code are machine independent code, but they are close to machine instruction. Syntax tree, Postfix notation, 3-address code, DAG can be used as intermediate language.

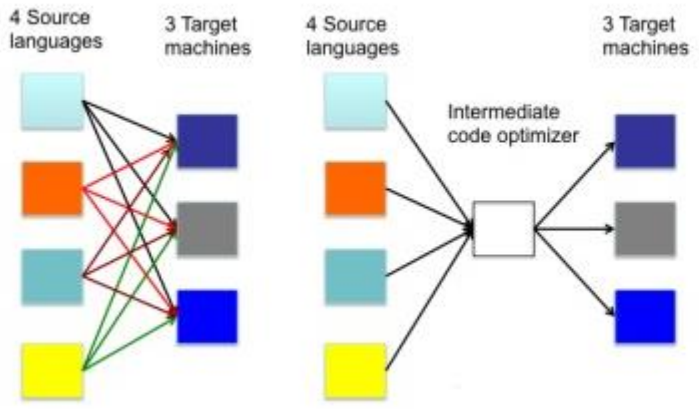


The code which is used for the conversion of source code into machine code is termed as intermediate code. Intermediate code lies in the middle of source code & machine code. The intermediate code generator, which is usually the front end of a compiler, is used to generate intermediate code. Intermediate code generation is done for the target machine & are machine-independent codes.


Lexical Analysis
Syntax Analysis
Semantic Analysis
Intermediate Code Generation
Code Optimisation
Target Code Generation
Front-End (Machine Independent)
Back-End (Machine Dependent)

NEED FOR INTERMEDIATE CODE:

1. Suppose we have x number of the source language and y number of the target language:
    1. **Without ICG –** we have to change each source language into target language directly, So, for each source-target pair we will need a compiler. Hence, we need (x*y) compilers, which can be a very large number and which is literally impossible.
    2. **With ICG –** we will need only x number of compiler to convert each source language into intermediate code. We will also need y compiler to convert the intermediate code into y target languages. So, we will need only (x+y) number of the compiler with ICG which is way lesser than x*y no of the compiler.



2. **Re-targeting is facilitated:**
   A compiler for a different machine can be created by attaching a back-end (which generate target code) for the new machine to an existing front-end (which generate intermediate code).
3. **Machine independent:**
   A Machine independent code-optimizer can be applied to the intermediate code. So, this can be run on any machine.

4. **Simplicity:**
   Intermediate code is simple enough to be easily converted to any target code. So ICG reduces the overhead of target code generation.
5. **Complexity:**
   Intermediate code is Complex enough to represent all complex structure of high-level languages.
6. **Modification:**
   We can easily modify our code to get better performance and throughput by applying optimization technique to the Intermediate code.

.

INTERMEDIATE LANGUAGE:

A language that is generated from programming source code, but that cannot be directly executed by the CPU. Also called "bytecode," "p-code," "pseudocode" or "pseudo language," the intermediate language (IL) is platform independent. It can be run in any computer environment that has a runtime engine for the language.

INTERMEDIATE CODE REPRESENTATION:

Intermediate Code Representation: In the analysis-synthesis model of a compiler, the front end of a compiler translates a source program into an independent intermediate code, then the back end of the compiler uses this intermediate code to generate the target code (which can be understood by the machine).

The benefits of using machine independent intermediate code are:

1. Intermediate code eliminates the need of a new full compiler for every unique machine by keeping the analysis portion same for all the compilers.

2. The second part of compiler, synthesis, is changed according to the target machine.

3. It becomes easier to apply the source code modifications to improve code performance by applying code optimization techniques on the intermediate code.

Intermediate codes can be represented in a variety of ways and they have their own benefits.

1. High Level IR - High-level intermediate code representation is very close to the source language itself. They can be easily generated from the source code and we can easily apply code modifications to enhance performance. But for target machine optimization, it is less preferred. 2. Low Level IR - This one is close to the target machine, which makes it suitable for register and memory allocation, instruction set selection, etc. It is good for machine-dependent optimizations.

The following are commonly used intermediate code representation:

1. Prefix and Postfix Notation:

There are two very important expression formats. Consider the infix expression A + B. What would happen if we moved the operator before the two operands? The resulting expression would be + A B. Likewise, we could move the operator to the end. We would get A B +. These changes to the position of the operator with respect to the operands create two new expression formats, **prefix** and **postfix**. Prefix expression notation requires that all operators precede the two operands that they work on. Postfix, on the other hand, requires that its operators come after the corresponding operands.

A + B * C would be written as + A * B C in prefix. The multiplication operator comes immediately before the operands B and C, denoting that * has precedence over +. The addition operator then appears before the A and the result of the multiplication.

In postfix, the expression would be A B C * +. Again, the order of operations is preserved since the * appears immediately after the B and the C, denoting that * has precedence, with + coming after. Although the operators moved and now appear either before or after their respective operands, the order of the operands stayed exactly the same relative to one another.

Example: 1

Infix Expression	**Prefix Expression**	**Postfix Expression**
A + B	+ A B	A B +
A + B * C	+ A * B C	A B C * +

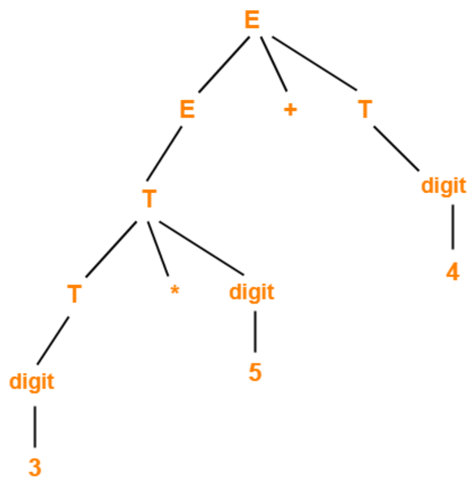
Example: 2

Infix Expression	**Prefix Expression**	**Postfix Expression**
(A + B) * C	* + A B C	A B + C *

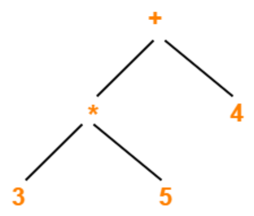
Example: 3

Infix Expression	**Prefix Expression**	**Postfix Expression**
(A + B) * (C + D)	* + A B + C D	A B + C D + *
A * B + C * D	+ * A B * C D	A B * C D * +
((A + B) * C) - D	- * + A B C D	A B + C * D -

2. Syntax Tree:
Parse tree is a graphical representation of the replacement process in a derivation. Syntax tree is nothing more than condensed form of a parse tree ie., Syntax trees are abstract or compact representation of parse trees. They are also called as **Abstract Syntax Trees**. In parse tree, each interior node represents a grammar rule and each leaf node represents a terminal. In syntax tree, each interior node represents an operator and each leaf node represents an operand. To form syntax tree put parentheses in the expression, this way it's easy to recognize which operand should come first.

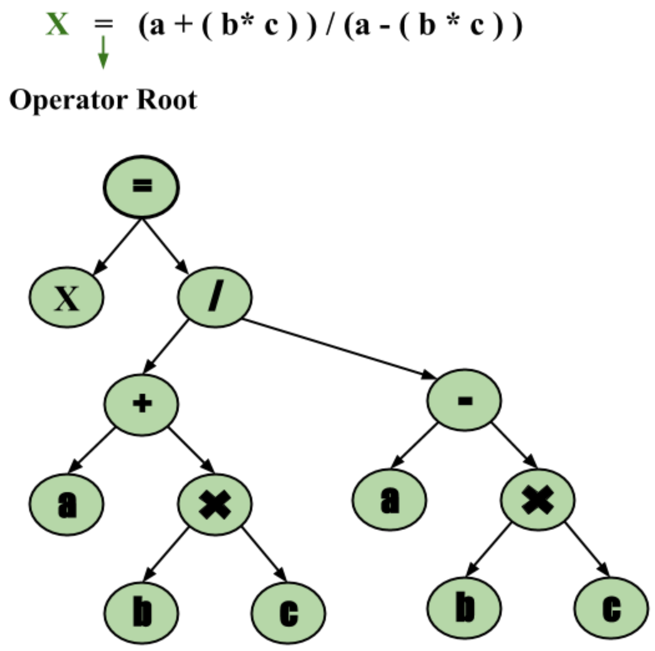


Parse Tree



Syntax Tree

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



Considering the following grammar

$$E \rightarrow E + T \mid T$$

$$T \rightarrow T * F \mid F$$

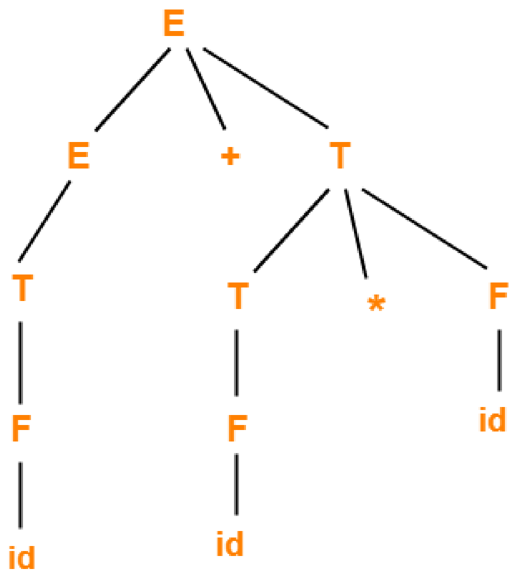
$$F \rightarrow (E) \mid id$$

Generate the following for the string id + id * id

1. Parse tree

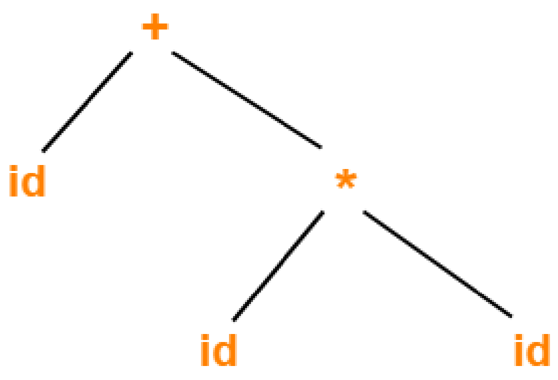
2. Syntax tree
3. Directed Acyclic Graph (DAG)

Parse Tree:



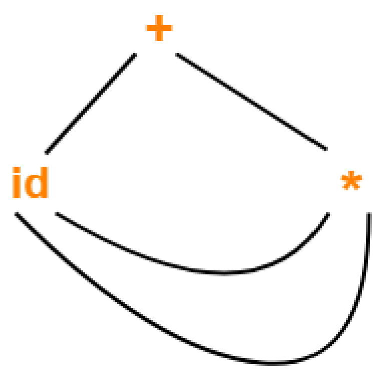
Parse Tree

Syntax Tree:



Syntax Tree

Directed Acyclic Graph:



Directed Acyclic Graph

Directed Acyclic Graph (DAG) is a special kind of Abstract Syntax Tree. Each node of it contains a unique value. It does not contain any cycles in it, hence called Acyclic.

Construction of DAGs:
Following rules are used for the construction of DAGs:
Rule: 1

- Interior nodes always represent the operators.
- Exterior nodes (leaf nodes) always represent the names, identifiers or constants.

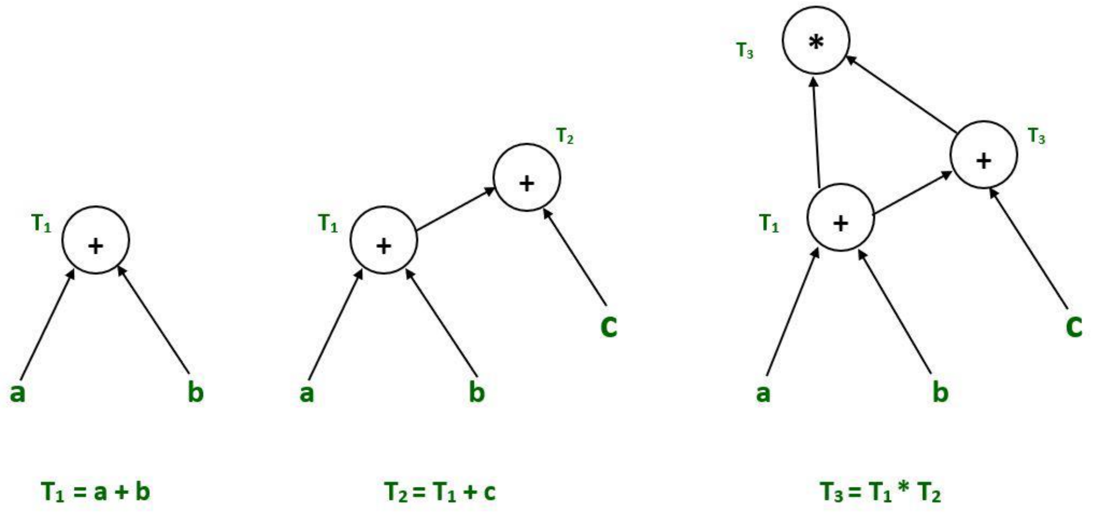
Rule: 2

- A check is made to find if there exists any node with the same value.
- A new node is created only when there does not exist any node with the same value.
- This action helps in detecting the common sub-expressions and avoiding the re-computation of the same.

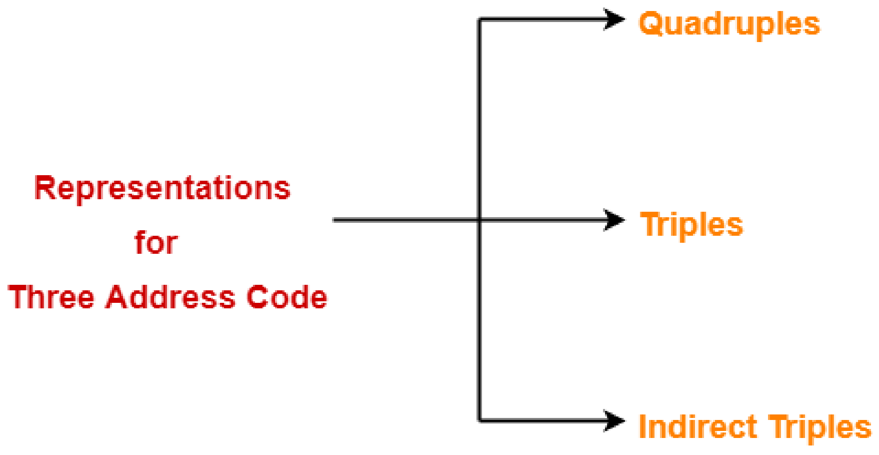
Rule: 3

- The assignment instructions of the form x:=y are not performed unless they are necessary.

Example:
Consider the expression
a + b * a + b + c
Three address code is
$T_1 = a + b$
$T_2 = T_1 + c$
$T_3 = T_1 * T_2$



3. Three-Address Code:
A statement involving no more than three references (two for operands and one for result) is known as three address statement. A sequence of three address statements is known as three address code. Three address statement is of the form x = y op z, here x, y, z will have address (memory location). Sometimes a statement might contain less than three references but it is still called three address statement.



Example:
The three-address code for the expression a + b * c + d
$T_1 = b * c$
$T_2 = a + T_1$
$T_3 = T_2 + d$

T_1, T_2 and T_3 are temporary variables.

- Three Address Code is a form of an intermediate code.
- It is generated by the compiler for implementing Code Optimization**.**
- It uses maximum three addresses to represent any statement.

The commonly used representations for implementing Three Address Code are

1. Quadruples:

In quadruples representation, each instruction is divided into the following 4 different fields

op, arg1, arg2, result

Here,

- The op field is used for storing the internal code of the operator.
- The arg1 and arg2 fields are used for storing the two operands used.
- The result field is used for storing the result of the expression.

2. Triples:

In triples representation,

- References to the instructions are made.
- Temporary variables are not used.

3. Indirect Triples:

- This representation is an enhancement over triples representation.
- It uses an additional instruction array to list the pointers to the triples in the desired order.
- Thus, instead of position, pointers are used to store the results.
- It allows the optimizers to easily re-position the sub-expression for producing the optimized code.

Example:

Translate the following expression to quadruple, triple and indirect triple-

a + b x c / e ↑ f + b x a

Three Address Code for the given expression is

T1 = e ↑ f
T2 = b x c
T3 = T2 / T1
T4 = b x a
T5 = a + T3
T6 = T5 + T4

Now, we write the required representations

Quadruple Representation:

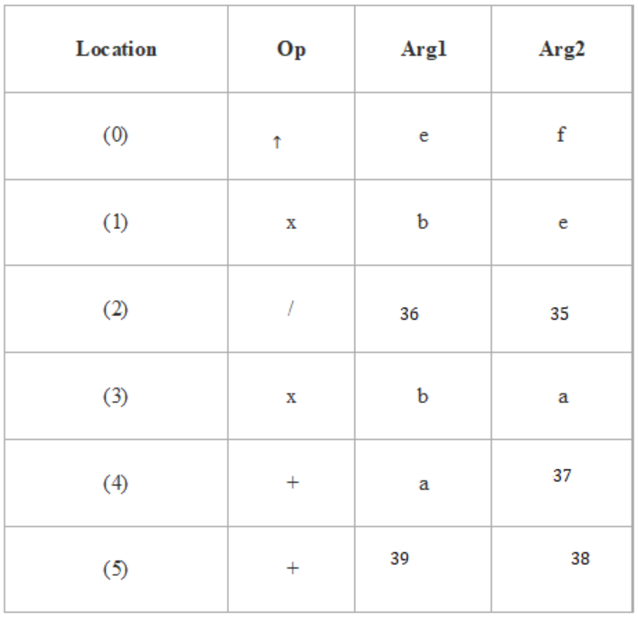
Location	**Op**	**Arg1**	**Arg2**	**Result**
(0)	↑	e	f	T1
(1)	x	b	c	T2
(2)	/	T2	T1	T3
(3)	x	b	a	T4

(4)	+	a	T3	T5
(5)	+	T5	T4	T6

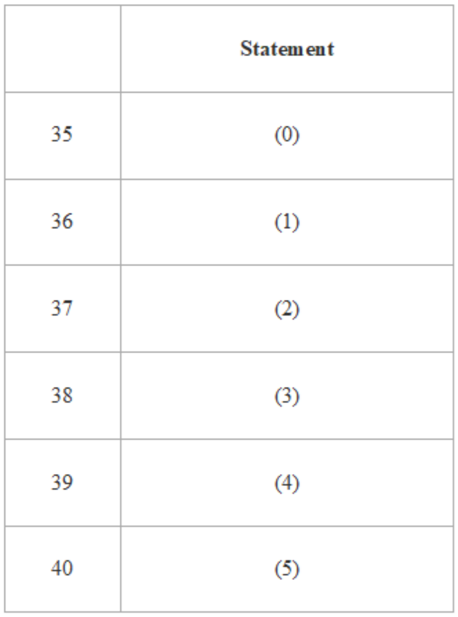
Triple Representation:

Location	Op	Arg1	Arg2
(0)	↑	e	f
(1)	x	b	c
(2)	/	(1)	(0)
(3)	x	b	a
(4)	+	a	(2)
(5)	+	(4)	(3)

Indirect Triple Representation:



Location	Op	Arg1	Arg2
(0)	↑	e	f
(1)	x	b	e
(2)	/	36	35
(3)	x	b	a
(4)	+	a	37
(5)	+	39	38



	Statement
35	(0)
36	(1)
37	(2)
38	(3)
39	(4)
40	(5)

Characteristics of Three Address instructions:

- They are generated by the compiler for implementing Code Optimization.
- They use maximum three addresses to represent any statement.
- They are implemented as a record with the address fields.

General Form:
In general, Three Address instructions are represented as-

a = b op c

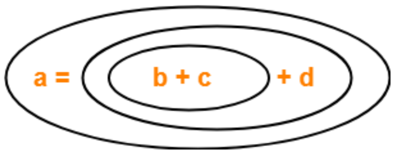
Here,

a, b and c are the operands.
Operands may be constants, names, or compiler generated temporaries.
op represents the operator.

Common Three Address Instruction Forms:
1. Assignment Statement:

x = y op z and x = op y

Here,
x, y and z are the operands.
op represents the operator.
It assigns the result obtained after solving the right-side expression of the assignment operator to the left side operand.
Example:
Write Three Address Code for the following expression:
a = b + c + d
The given expression will be solved as



Three Address Code for the given expression is
$t_1 = b + c$
$t_2 = t_1 + d$
$a = t_2$

2. Copy Statement:

x = y

Here,
x and y are the operands.
= is an assignment operator.
It copies and assigns the value of operand y to operand x.
Example:
a = 2

3. Conditional Jump:

If x relop y
goto L

Here,
x & y are the operands.
L is the tag or label of the target statement.
relop is a relational operator.
If the condition “x relop y” gets satisfied, then the control is sent directly to the location specified by label L. All the statements in between are skipped.
If the condition “x relop y” fails, then the control is not sent to the location specified by label L. The next statement appearing in the usual sequence is executed.
Example:
Write Three Address Code for the following expression:
If A < B then 1 else 0
Three Address Code for the given expression is-
if (A < B) goto 4
$t_1 = 0$
goto 5

4: t_1 = 1
5: ….

4. Unconditional Jump:

goto X

Here, X is the tag or label of the target statement.
On executing the statement, the control is sent directly to the location specified by label X.
All the statements in between are skipped.

5. Procedure Call:

param x call p return y

Here, p is a function which takes x as a parameter and returns y.

SYNTAX DIRECTED TRANSLATION:

Definition:

Syntax Directed Translation has augmented rules to the grammar that facilitate semantic analysis. SDT involves passing information bottom-up and/or top-down the parse tree in form of attributes attached to the nodes. Syntax-directed translation rules use
1) lexical values of nodes,
2) constants &
3) attributes associated with the non-terminals in their definitions.

The general approach to Syntax-Directed Translation is to construct a parse tree or syntax tree and compute the values of attributes at the nodes of the tree by visiting them in some order. In many cases, translation can be done during parsing without building an explicit tree.
Semantic analysis computes additional information related to the meaning of the program once the syntactic structure is known. Semantic analysis involves adding information to the symbol table and performing type checking. It needs both representation and implementation mechanism.
The Principle of Syntax Directed Translation states that the meaning of an input sentence is related to its syntactic structure, i.e., to its Parse-Tree.
We associate Attributes to the grammar symbols representing the language constructs. Values for attributes are computed by Semantic Rules associated with grammar productions.
Evaluation of Semantic Rules may:
• Generate Code
• Insert information into the Symbol Table
• Perform Semantic Check
• Issue error messages

Syntax Directed Definitions are a generalization of context-free grammars in which:
• Grammar symbols have an associated set of Attributes.
• Productions are associated with Semantic Rules for computing the values of attributes.
Annotated Parse-Trees where each node of the tree is a record with a field for each attribute (e.g., X.a indicates the attribute a of the grammar symbol X). The value of an attribute of a grammar symbol at a given parse-tree node is defined by a semantic rule associated with the production used at that node.

Example:

E -> E+T | T
T -> T*F | F
F -> id

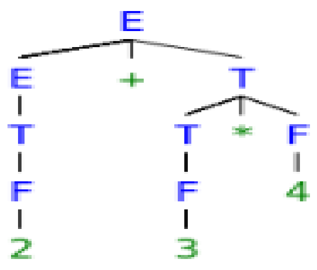
This is a grammar to syntactically validate an expression having additions and multiplications in it. Now, to carry out semantic analysis we will augment SDT rules to this grammar, in order to pass some information up the parse tree and check for semantic errors, if any. In this example, we will focus on the evaluation of the given expression, as we don't have any semantic assertions to check in this very basic example.

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

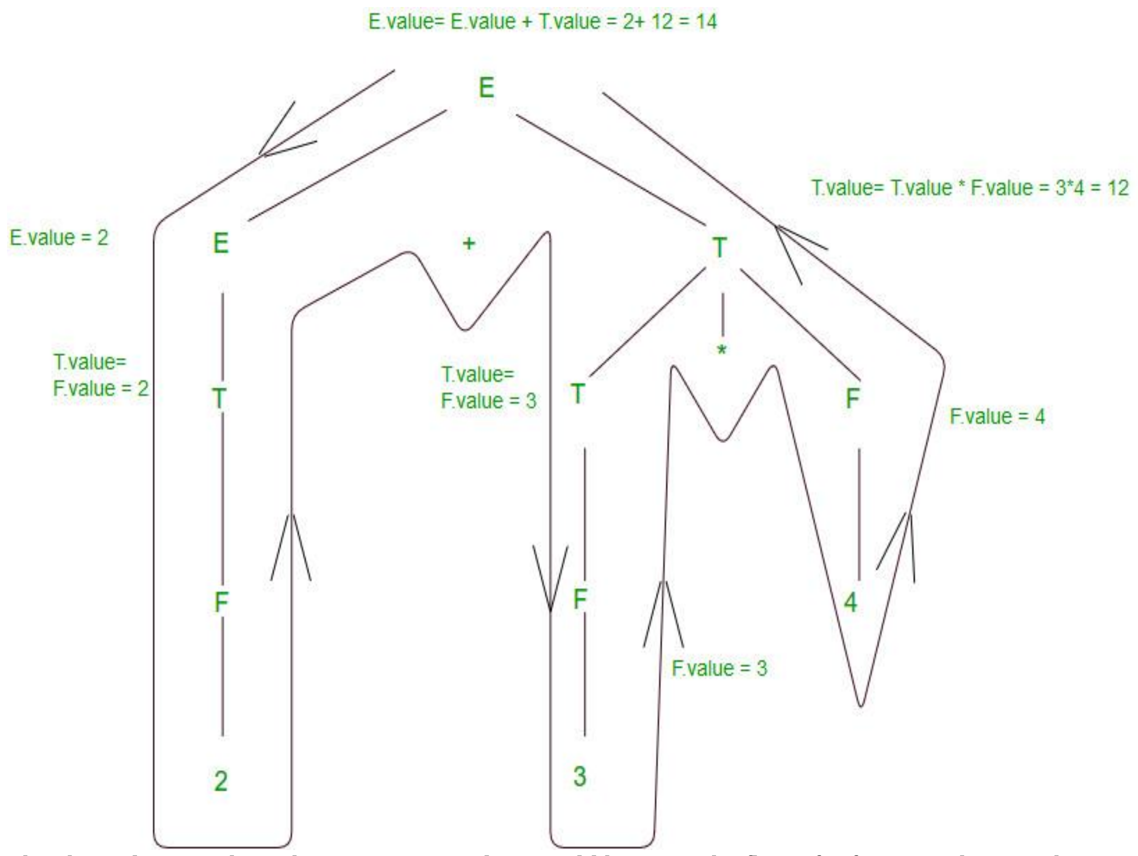
T -> T*F { T.val = T.val * F.val }
T -> F { T.val = F.val }
F -> id { F.val = id.lexval }

For understanding translation rules further, we take the first SDT augmented to [E -> E+T] production rule. The translation rule in consideration has val as an attribute for both the non-terminals – E & T. Right-hand side of the translation rule corresponds to attribute values of right-side nodes of the production rule and vice-versa.
Generalizing, SDT are augmented rules to a CFG that associate
1) set of attributes to every node of the grammar and
2) set of translation rules to every production rule using attributes, constants, and lexical values.

Let's take a string to see how semantic analysis happens – S = 2+3*4. Parse tree corresponding to S would be



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 of our example.



The above diagram shows how semantic analysis could happen. The flow of information happens bottom-up and all the children's attributes are computed before parents, as discussed above. Right-hand side nodes are sometimes annotated with subscript 1 to distinguish between children and parents.

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).

For eg. 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).

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.

TYPES OF SDT:

Attributes may be of two types – S and L attributed SDTs.

1. S-attributed SDT:

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

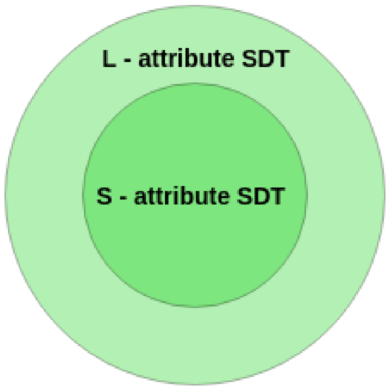
S-attributed SDTs are evaluated in bottom-up parsing, as the values of the parent nodes depend upon the values of the child nodes.

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.
For example,
A -> XYZ {Y.S = A.S, Y.S = X.S, Y.S = Z.S}
is not an L-attributed grammar since Y.S = A.S and Y.S = X.S are allowed but Y.S = Z.S violates the L-attributed SDT definition as attributed is inheriting the value from its right siblings.

If a definition is S-attributed, then it is also L-attributed but **NOT** vice-versa.



Example:
Consider the given below SDT.
P1: S -> MN {S.val= M.val + N.val}
P2: M -> PQ {M.val = P.val * Q.val and P.val =Q.val}

Select the correct option.
(A) Both P1 and P2 are S attributed.
(B) P1 is S attributed and P2 is L-attributed.
(C) P1 is L attributed but P2 is not L-attributed.
(D) None of the above

The correct answer is option C as, In P1, S is a synthesized attribute and in L-attribute definition synthesized is allowed. So P1 follows the L-attributed definition. But P2 doesn’t follow L-attributed definition as P is depending on Q which is RHS to it.

<u>DECLARATIONS:</u>

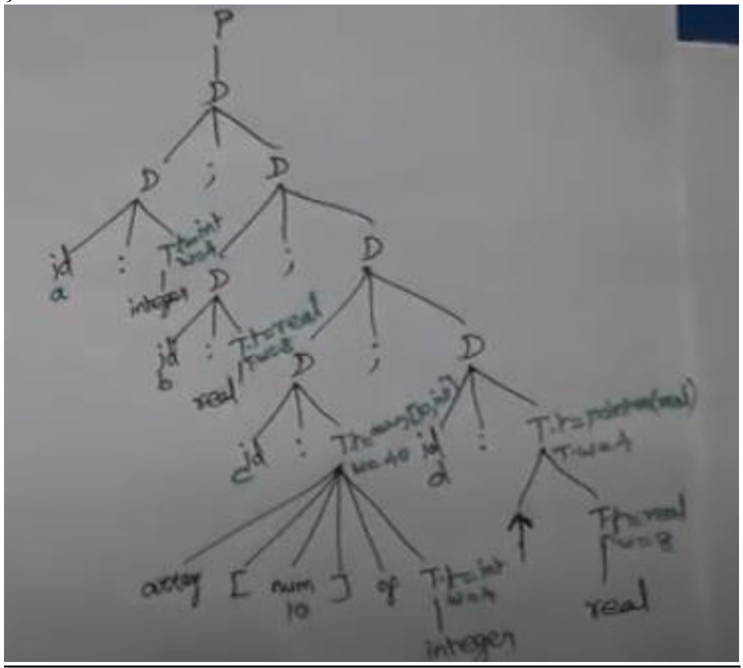
A variable or procedure has to be declared before it can be used. Declaration involves allocation of space in memory and entry of type and name in the symbol table. A program may be coded and designed keeping the target machine structure in mind, but it may not always be possible to accurately convert a source code to its target language.
Taking the whole program as a collection of procedures and sub-procedures, it becomes possible to declare all the names local to the procedure. Memory allocation is done in a consecutive manner and names are allocated to memory in the sequence they are declared in the program. We use offset variable and set it to zero {offset = 0} that denote the base address.
The source programming language and the target machine architecture may vary in the way names are stored, so relative addressing is used. While the first name is allocated memory starting from the memory location 0 {offset=0}, the next name declared later, should be allocated memory next to the first one.

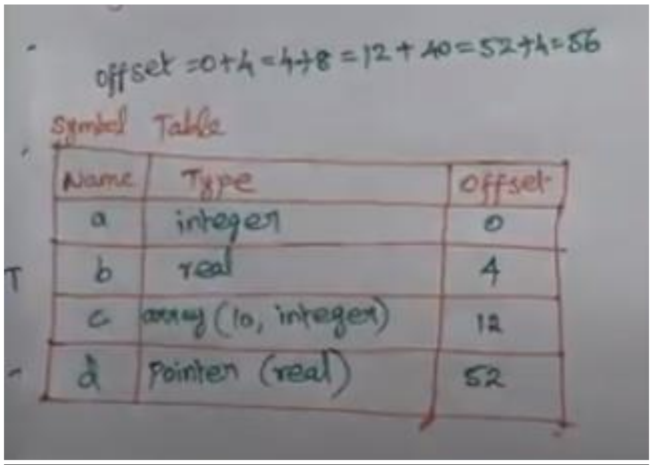
```
P→D                  { offset : = 0 }
D→D ; D
D→id : T             { enter(id.name, T.type, offset);
                     offset : = offset + T.width }
T→integer            { T.type : = integer;
                     T.width : = 4    }
T→real               { T.type : = real;
                     T.width : = 8    }
T→array [ num ] of T1          { T.type : = array(num.val, T1.type);
                               T.width : = num.val X T1.width }
T→↑T1                          { T.type : = pointer ( T1.type);
                               T.width : = 4 }
```

We take the example of C programming language where an integer variable is assigned 4 bytes of memory and a real variable is assigned 8 bytes of memory.

To enter this detail in a symbol table, a procedure *enter* can be used. This method may have the following structure:
Enter (name, type, offset)
This procedure should create an entry in the symbol table, for variable *name*, having its type set to type and relative address *offset* in its data area.
Consider the example:

```
{
a : integer;
b : real;
c : array[10] of integer;
d : ↑ real;
…
}
```




offset =0+4=4+8=12+40=52+4=56

Symbol Table

Name	Type	offset
a	integer	0
b	real	4
c	array (10, integer)	12
d	Pointer (real)	52

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
S → id := E
E → E1 + E2
E → E1 * E2
E → (E1)
E → id
The translation scheme of above grammar is given below:

Production rule	Semantic actions

S → id :=E	{p = look_up(id.name);
	If p ≠ nil then
	Emit (p = E.place)
	Else
	Error;
	}

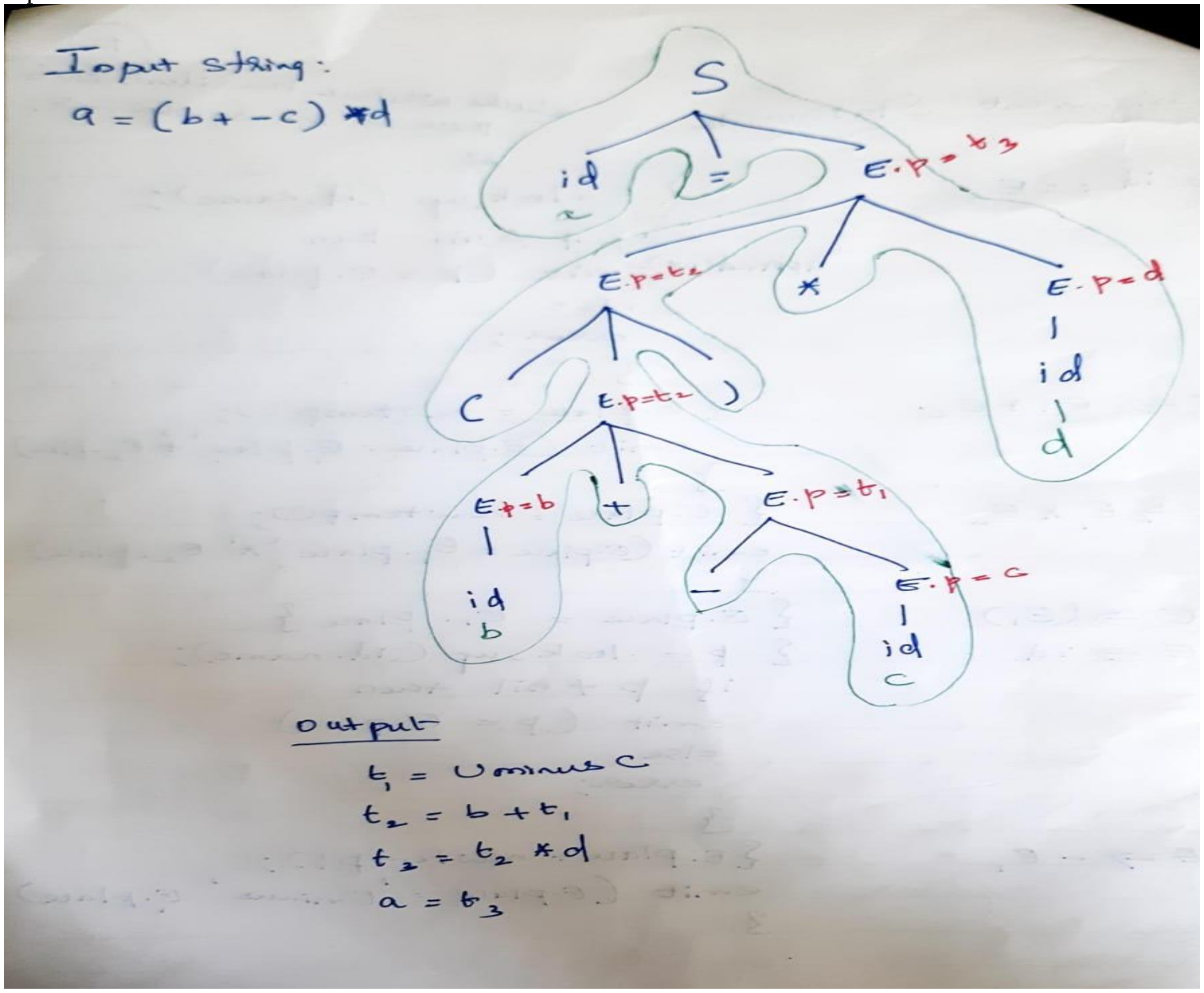
E → E1 + E2	{E.place = newtemp();
	Emit (E.place = E1.place '+' E2.place)
	}

E → E1 * E2	{E.place = newtemp();
	Emit (E.place = E1.place '*' E2.place)
	}

E → (E1)	{E.place = E1.place}

E → id	{p = look_up(id.name);
	If p ≠ nil then
	Emit (p = E.place)
	Else
	Error;
	}

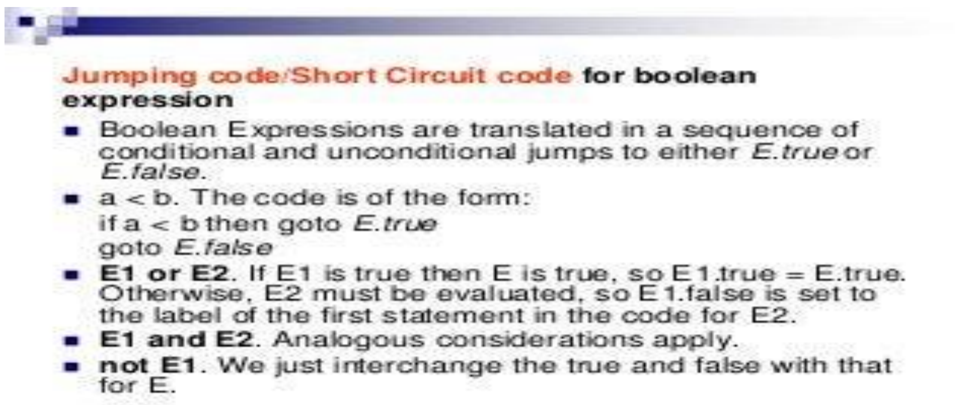
The p returns the entry for id.name in the symbol table.
The Emit function is used for appending the three-address code to the output file. Otherwise, it will report an error.
The newtemp() is a function used to generate new temporary variables.
E.place holds the value of E.



Input string:
a = (b+ -c) *d

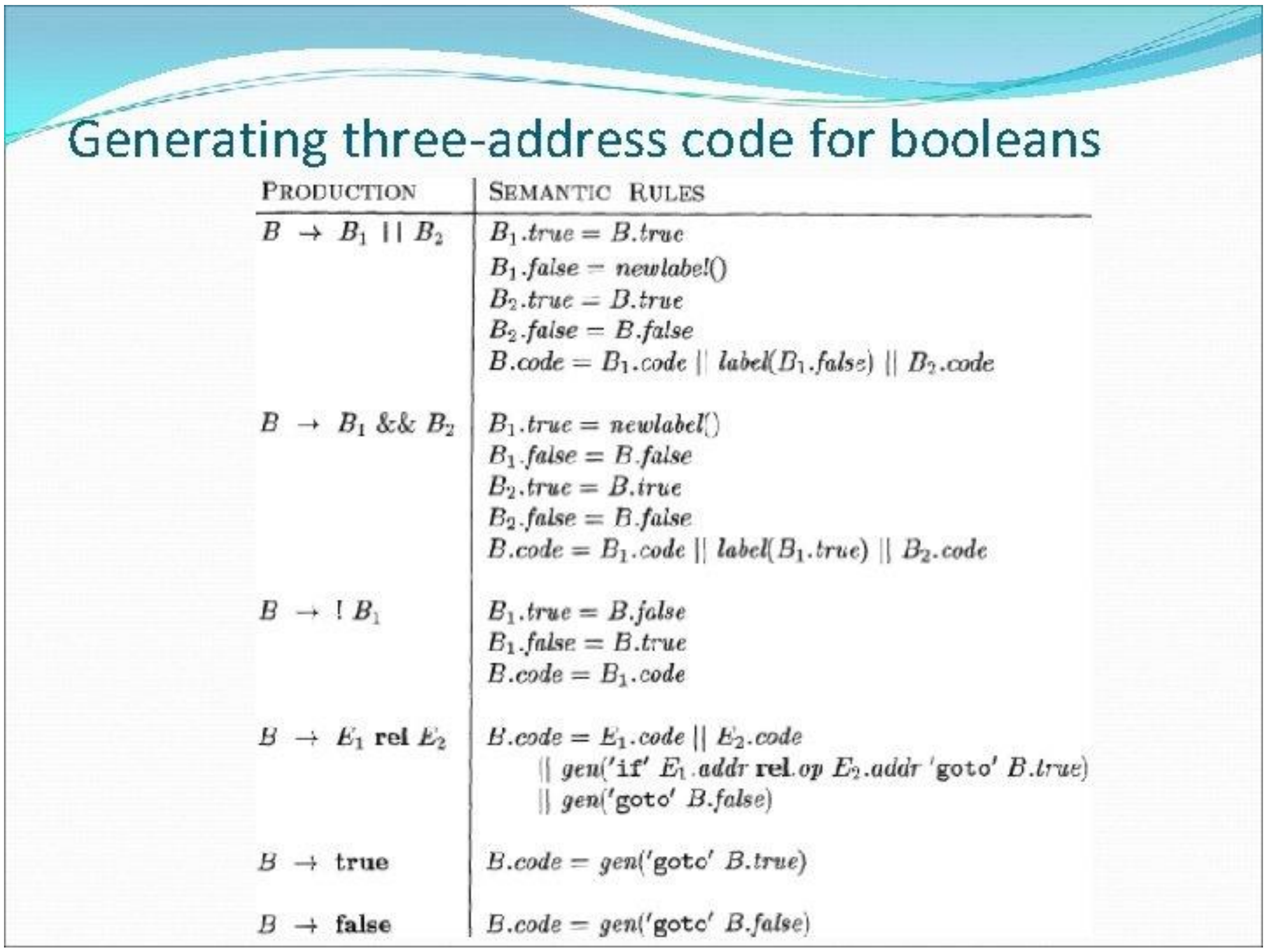
Output
```
t1 = Uminus c
t2 = b + t1
t3 = t2 * d
a = t3
```

BOOLEAN EXPRESSIONS:



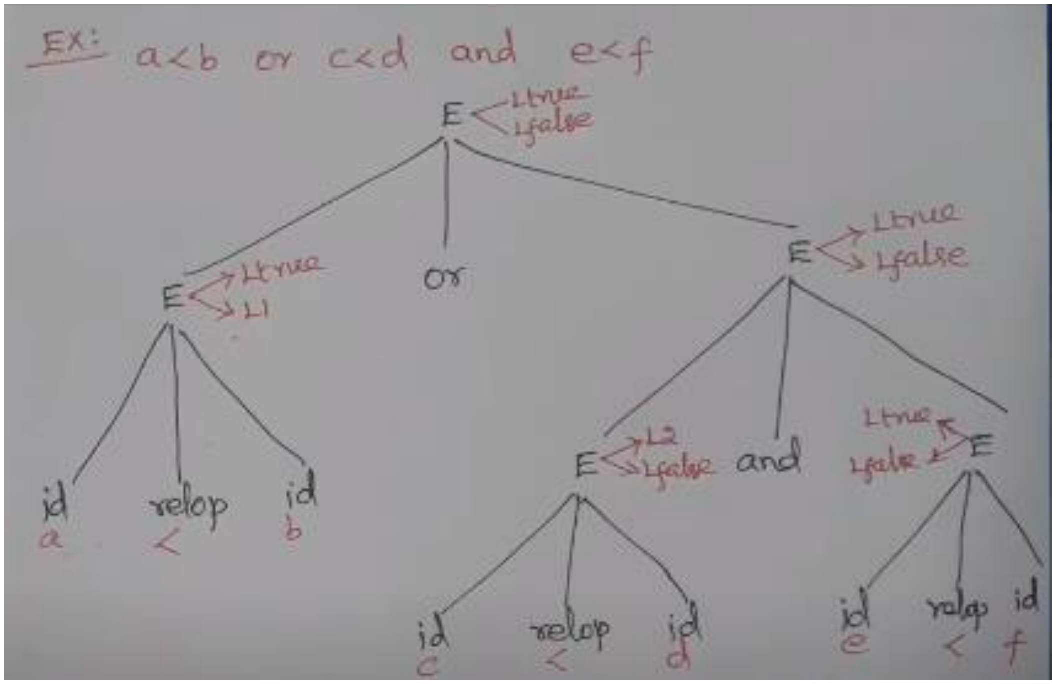
Jumping code/Short Circuit code for boolean expression

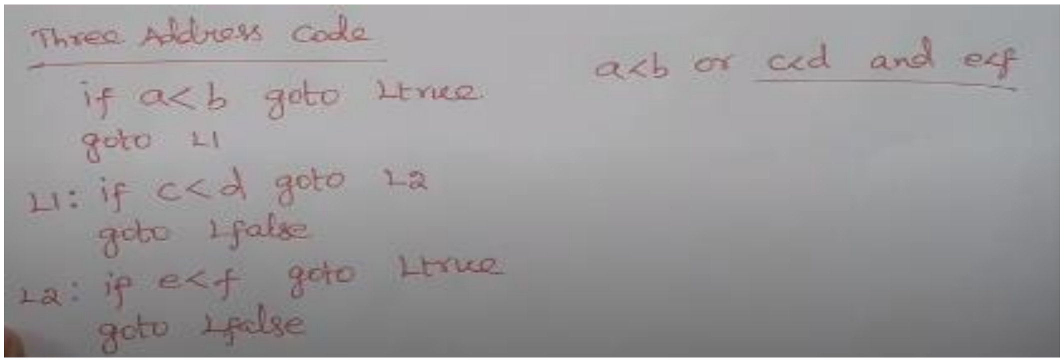
- Boolean Expressions are translated in a sequence of conditional and unconditional jumps to either *E.true* or *E.false*.
- a < b. The code is of the form:
  if a < b then goto *E.true*
  goto *E.false*
- **E1 or E2**. If E1 is true then E is true, so E1.true = E.true. Otherwise, E2 must be evaluated, so E1.false is set to the label of the first statement in the code for E2.
- **E1 and E2**. Analogous considerations apply.
- **not E1**. We just interchange the true and false with that for E.



Generating three-address code for booleans

PRODUCTION	SEMANTIC RULES
$B \rightarrow B_1 \mid\mid B_2$	$B_1.true = B.true$ $B_1.false = newlabel()$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \mid\mid label(B_1.false) \mid\mid B_2.code$
$B \rightarrow B_1 \; \&\& \; B_2$	$B_1.true = newlabel()$ $B_1.false = B.false$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \mid\mid label(B_1.true) \mid\mid B_2.code$
$B \rightarrow \; ! \; B_1$	$B_1.true = B.false$ $B_1.false = B.true$ $B.code = B_1.code$
$B \rightarrow E_1 \; \mathbf{rel} \; E_2$	$B.code = E_1.code \mid\mid E_2.code$ $\mid\mid gen(\text{'if'} \; E_1.addr \; \mathbf{rel}.op \; E_2.addr \; \text{'goto'} \; B.true)$ $\mid\mid gen(\text{'goto'} \; B.false)$
$B \rightarrow \mathbf{true}$	$B.code = gen(\text{'goto'} \; B.true)$
$B \rightarrow \mathbf{false}$	$B.code = gen(\text{'goto'} \; B.false)$





```
Three Address Code                    a<b or c<d and e<f
    if a<b goto Ltrue
    goto L1
L1: if c<d goto L2
    goto Lfalse
L2: if e<f goto Ltrue
    goto Lfalse
```

PROCEDURES CALL

Procedure is an important and frequently used programming construct for a compiler. It is used to generate good code for procedure calls and returns.

Calling sequence:

The translation for a call includes a sequence of actions taken on entry and exit from each procedure. Following actions take place in a calling sequence:

- When a procedure call occurs then space is allocated for activation record.
- Evaluate the argument of the called procedure.
- Establish the environment pointers to enable the called procedure to access data in enclosing blocks.
- Save the state of the calling procedure so that it can resume execution after the call.
- Also save the return address. It is the address of the location to which the called routine must transfer after it is finished.
- Finally generate a jump to the beginning of the code for the called procedure.

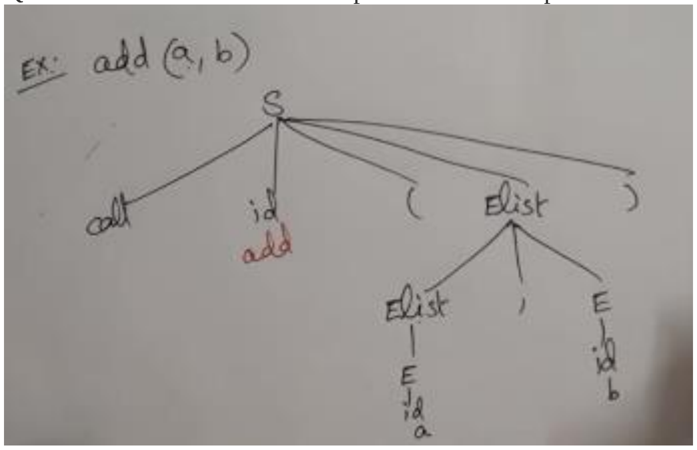
Let us consider a grammar for a simple procedure call statement

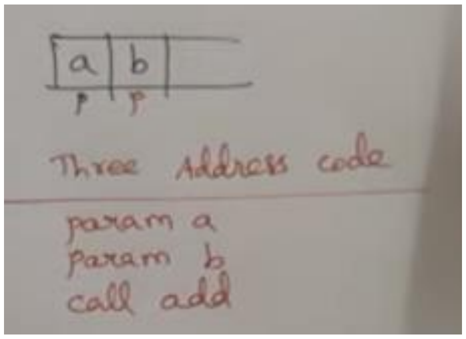
S $\rightarrow$ call id (Elist)
Elist $\rightarrow$ Elist, E
Elist $\rightarrow$ E

A suitable transition scheme for procedure call would be:

Production Rule	Semantic Action
S → call id(Elist)	for each item p on QUEUE do GEN (param p) GEN (call id.PLACE)
Elist → Elist, E	append E.PLACE to the end of QUEUE
Elist → E	initialize QUEUE to contain only E.PLACE

Queue is used to store the list of parameters in the procedure call.





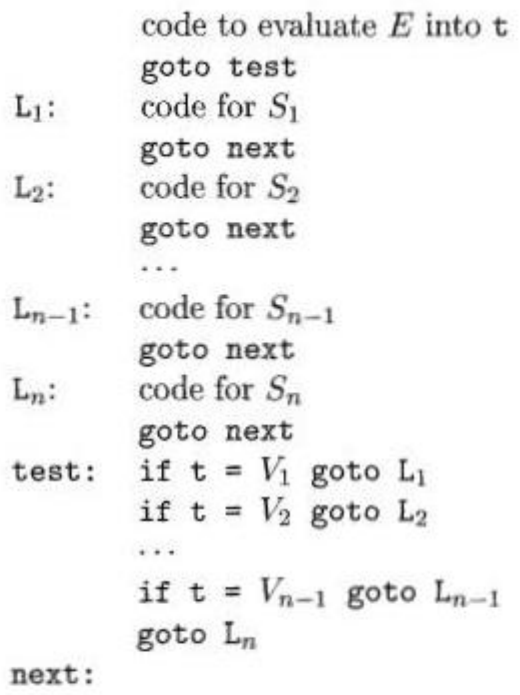
Three Address code

```
param a
param b
call add
```

CASE STATEMENTS:

```
switch ( E ) {
        case V1: S1
        case V2: S2
                ...
        case Vn-1: Sn-1
        default: Sn
}
```

Figure . Switch-statement syntax



```
            code to evaluate $E$ into t
            goto test
$L_1$:      code for $S_1$
            goto next
$L_2$:      code for $S_2$
            goto next
            ...
$L_{n-1}$:  code for $S_{n-1}$
            goto next
$L_n$:      code for $S_n$
            goto next
test:       if t = $V_1$ goto $L_1$
            if t = $V_2$ goto $L_2$
            ...
            if t = $V_{n-1}$ goto $L_{n-1}$
            goto $L_n$
next:
```

Figure Translation of a switch-statement

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.

Need for Backpatching:

Backpatching is mainly used for two purposes:

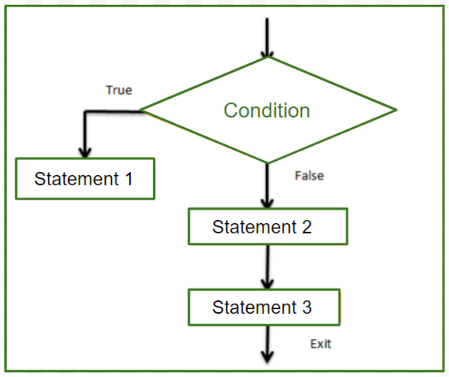
1. Boolean expression:

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

One-pass code generation using backpatching:

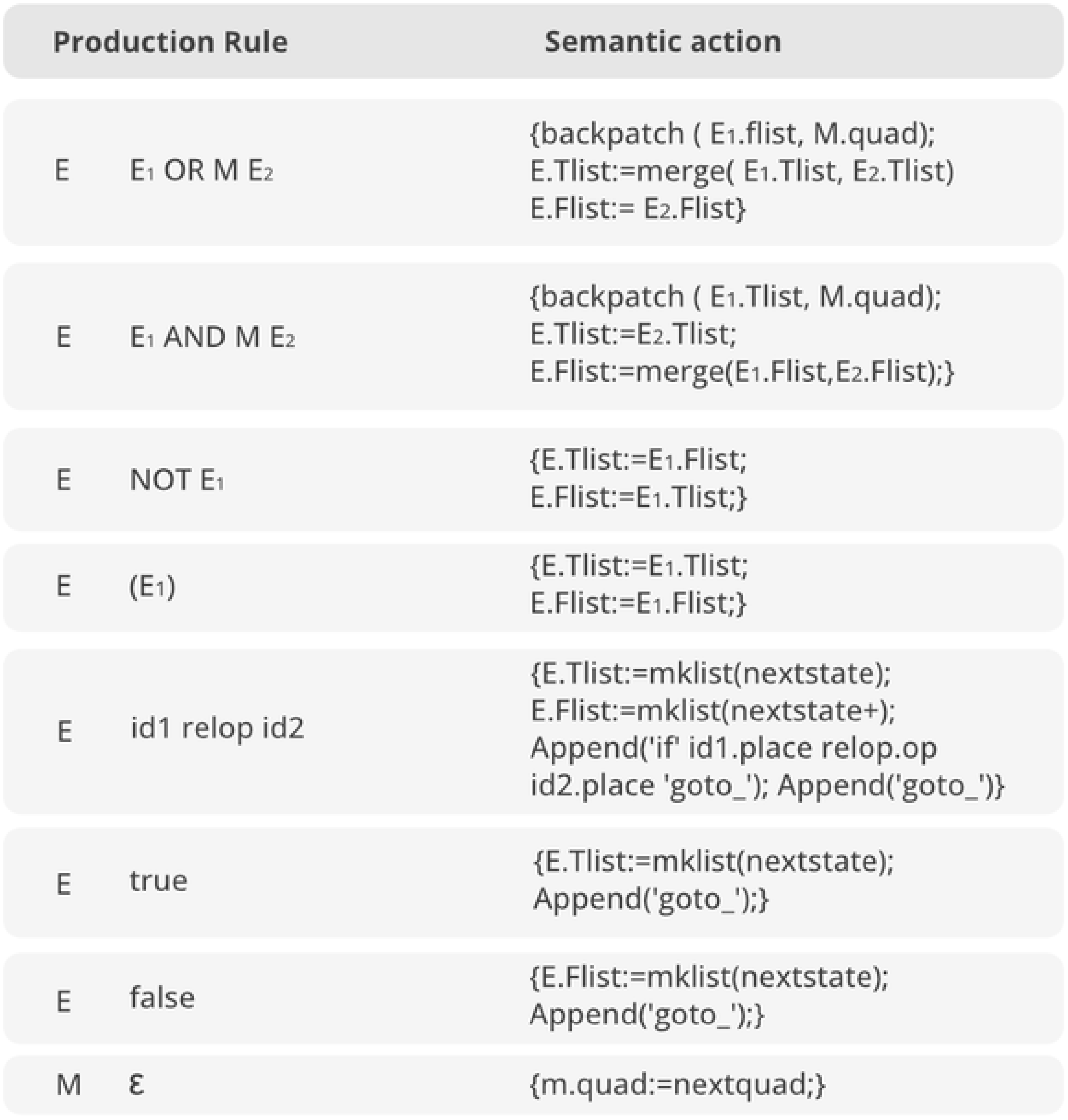
In a single pass, backpatching may be used to create a boolean expressions program as well as the flow of control statements. The synthesized properties truelist and falselist of non-terminal B are used to handle labels in jumping code for Boolean statements. The label to which control should go if B is true should be added to B.truelist, which is a list of 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:

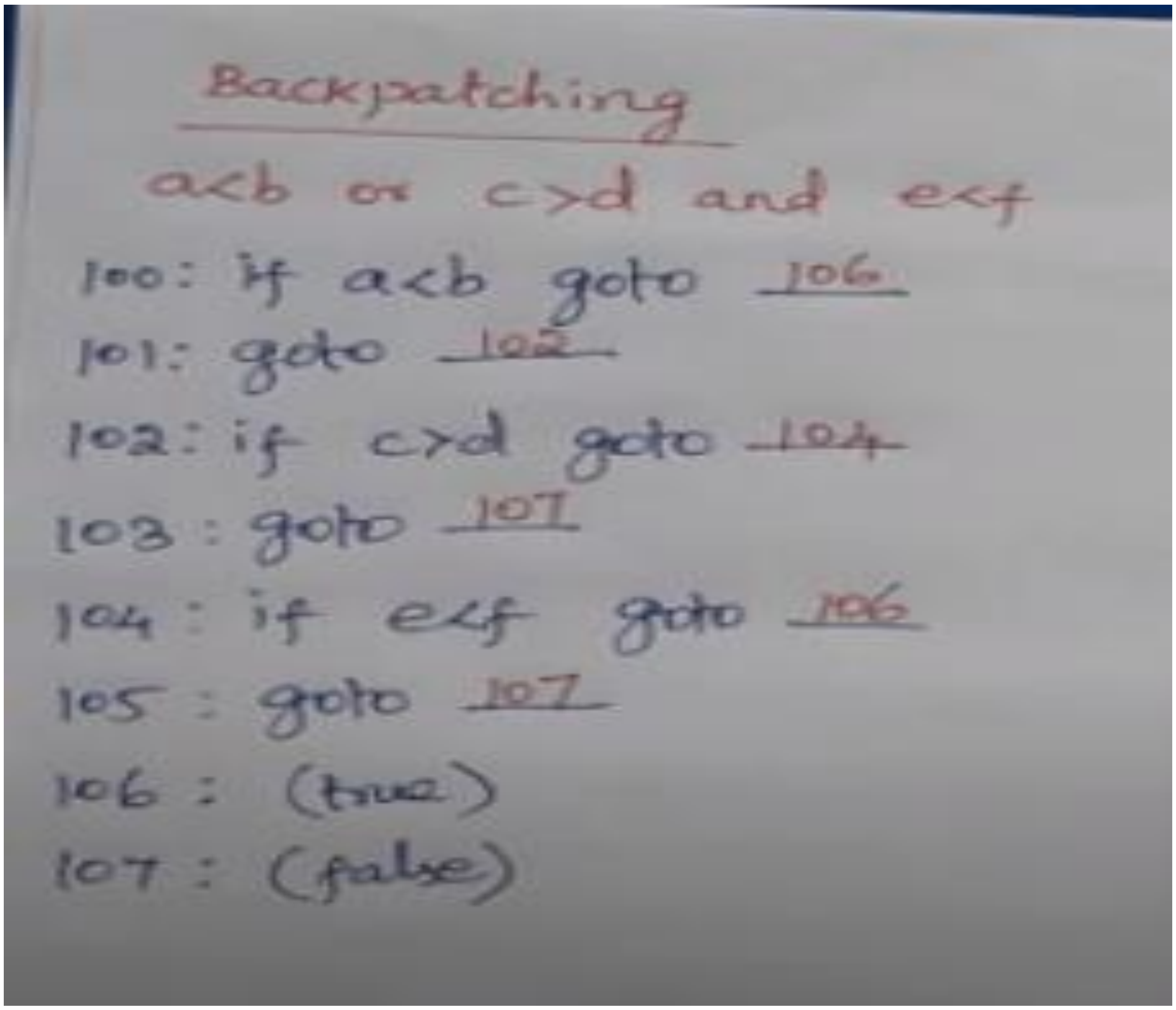
Using a translation technique, it can create code for Boolean expressions during bottom-up parsing. In grammar, a non-terminal marker M creates a semantic action that picks up the index of the next instruction to be created at the proper time.
For Example, Backpatching using boolean expressions production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



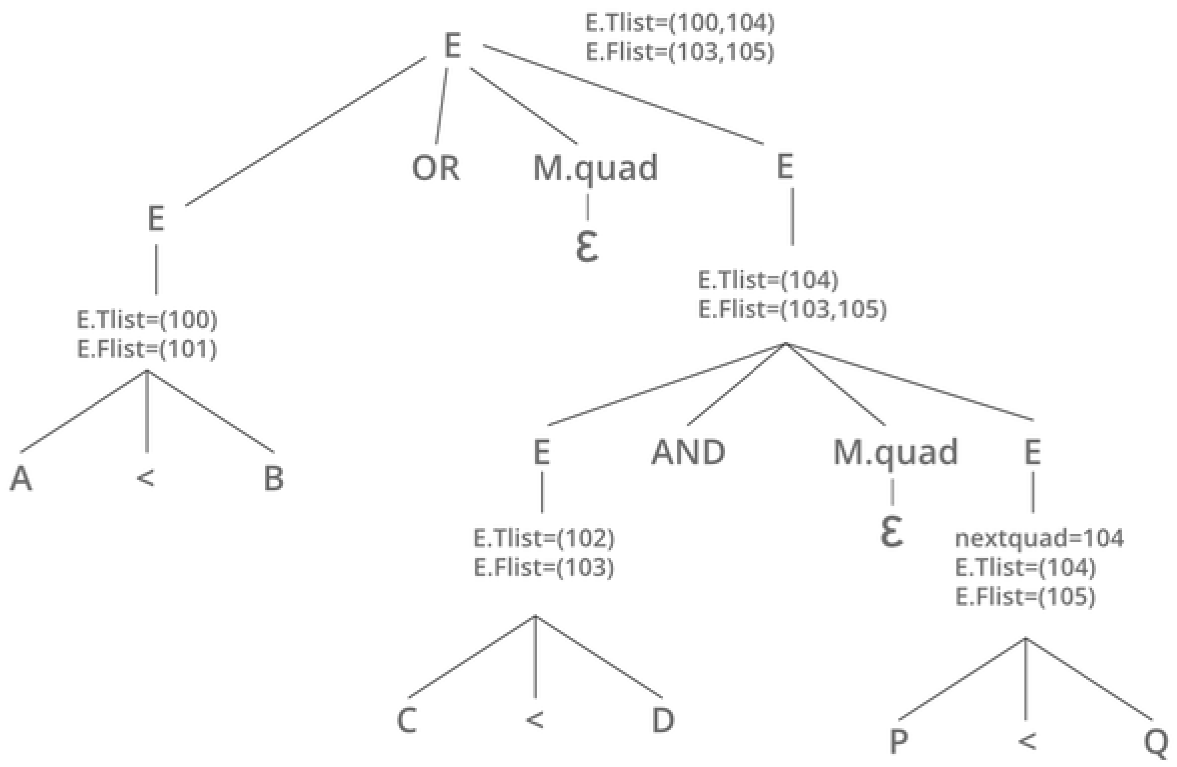
Backpatching

a<b or c>d and e<f

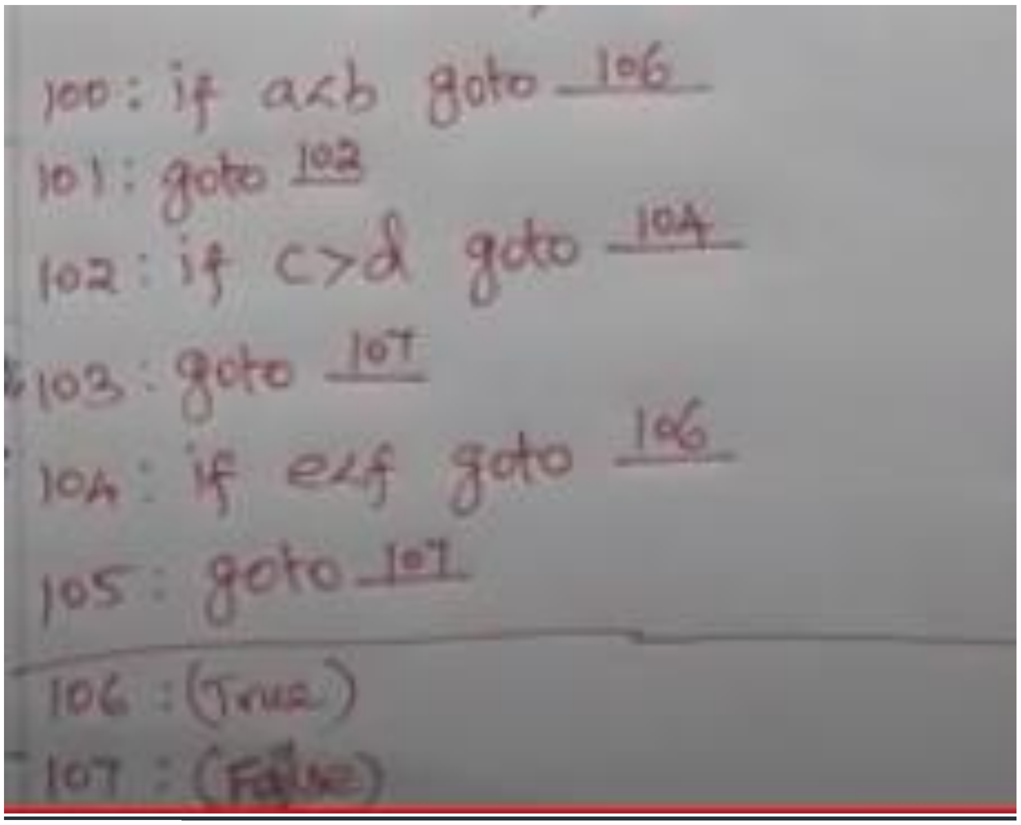
```
100: if a<b goto 106
101: goto 102
102: if c>d goto 104
103: goto 107
104: if e<f goto 106
105: goto 107
106: (true)
107: (false)
```

Three address codes for the given example

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



Parse tree for the example



```
100: if a<b goto 106
101: goto 102
102: if c>d goto 104
103: goto 107
104: if e<f goto 106
105: goto 107
106: (True)
107: (False)
```

CODE GENERATOR:

Code generator converts the intermediate representation of source code into a form that can be readily executed by the machine. A code generator is expected to generate the correct code. Designing of code generator should be done in such a way so that it can be easily implemented, tested and maintained.

The final phase in compiler model is the code generator. It takes as input an intermediate representation of the source program and produces as output an equivalent target program. The code generation techniques presented below can be used whether or not an optimizing phase occurs before code generation.

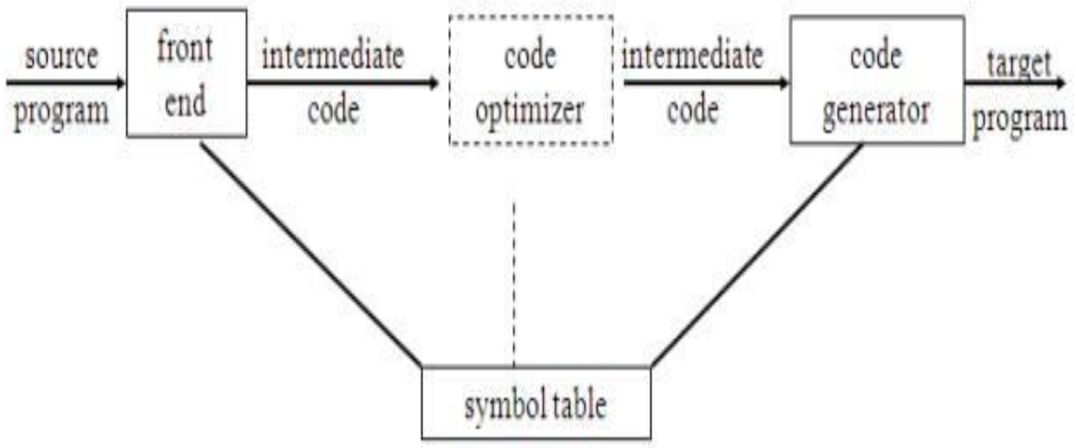


Fig. 4.1 Position of code generator

ISSUES IN THE DESIGN OF CODE GENERATION:

1. Input to code generator:

The input to code generator is the intermediate code generated by the front end, along with information in the symbol table that determines the run-time addresses of the data-objects denoted by the names in the intermediate representation. Intermediate codes may be represented mostly in

Linear representation

- Postfix notation

Three address code

- Quadruple
- Triple
- Indirect Triple

Graphical representation

- Syntax tree
- DAG.

The code generation phase just proceeds on an assumption that the input are free from all of syntactic and state semantic errors, the necessary type checking has taken place and the type-conversion operators have been inserted wherever necessary.

2. Target program:

The target program is the output of the code generator. The output may be

- Absolute machine language - as output has advantages that it can be placed in a fixed memory location and can be immediately executed.
- Relocatable machine language - as an output allows subprograms and subroutines to be compiled separately. Relocatable object modules can be linked together and loaded by linking loader. But there is added expense of linking and loading.
- Assembly language - as output makes the code generation easier. We can generate symbolic instructions and use macro-facilities of assembler in generating code. And we need an additional assembly step after code generation.

3. Memory Management:

Mapping the names in the source program to the addresses of data objects is done by the front end and the code generator. A name in the three address statements refers to the symbol table entry for name. Then from the symbol table entry, a relative address can be determined for the name.

4. Instruction selection:

Selecting the best instructions will improve the efficiency of the program. It includes the instructions that should be complete and uniform. Instruction speeds and machine idioms also plays a major role when efficiency is considered. But if we do not care about the efficiency of the target program then instruction selection is straight-forward.

For example, the respective three-address statements would be translated into the latter code sequence as shown below:

```
P:=Q+R
S:=P+T

MOV Q, R0
ADD R, R0
MOV R0, P
MOV P, R0
ADD T, R0
MOV R0, S
```

Here the fourth statement is redundant as the value of the P is loaded again in that statement that just has been stored in the previous statement. It leads to an inefficient code sequence. A given intermediate representation can be translated into many code sequences, with significant cost differences between the different implementations. A prior knowledge of instruction cost is needed in order to design good sequences, but accurate cost information is difficult to predict.

5. Register allocation issues:

Use of registers make the computations faster in comparison to that of memory, so efficient utilization of registers is important. The use of registers is subdivided into two subproblems:

During **Register allocation –** we select only those set of variables that will reside in the registers at each point in the program.
During a subsequent **Register assignment** phase, the specific register is picked to access the variable.
6. Choice of Evaluation order:
The code generator decides the order in which the instruction will be executed. The order of computations affects the efficiency of the target code. Among many computational orders, some will require only fewer registers to hold the intermediate results.

THE TARGET MACHINE:

The target computer is a type of byte-addressable machine. It has 4 bytes to a word. The target machine has n general purpose registers, R0, R1,.., Rn-1.

It also has two-address instructions of the form:

op source, destination

Where, op is used as an op-code and source and destination are used as a data field.
It has the following op-codes:

ADD (add source to destination)
SUB (subtract source from destination)
MOV (move source to destination)

The source and destination of an instruction can be specified by the combination of registers and memory location with address modes.

MODE	FORM	ADDRESS	EXAMPLE	ADDED COST
absolute	M	M	Add R0, R1	1
register	R	R	Add temp, R1	0
indexed	c(R)	C+ contents(R)	ADD 100 (R2), R1	1
indirect register	*R	contents(R)	ADD * 100	0
indirect indexed	*c(R)	contents(c+ contents(R))	(R2), R1	1
literal	#c	c	ADD #3, R1	1

Absolute addressing modes:

Mov a , R0

Mov Memory content ⇓ Register

Content (a) => R0

$$\begin{tikzcd} \overset{a}{\underset{1000}{\boxed{10}}} \arrow[r] & \overset{R_0}{\boxed{10}} \end{tikzcd}$$

Register addressing modes:

Mov R0, a

Content (R0) => a

$$\begin{tikzcd} \overset{R_0}{\underset{2000}{\boxed{5}}} \arrow[r] & \overset{a}{\boxed{5}} \end{tikzcd}$$

Indexed addressing modes:

Mov 4 (R0) , a

$$\text{Content}\ (\underset{\text{Constant}}{\underline{\underline{4}}} + \text{Content}\ (R_0)) \Rightarrow a$$

$$\rightarrow \overset{R_0}{\underset{3000}{\boxed{4000}}} + 4 \Rightarrow \underset{4004}{\boxed{20}} \Rightarrow \overset{a}{\boxed{20}}$$

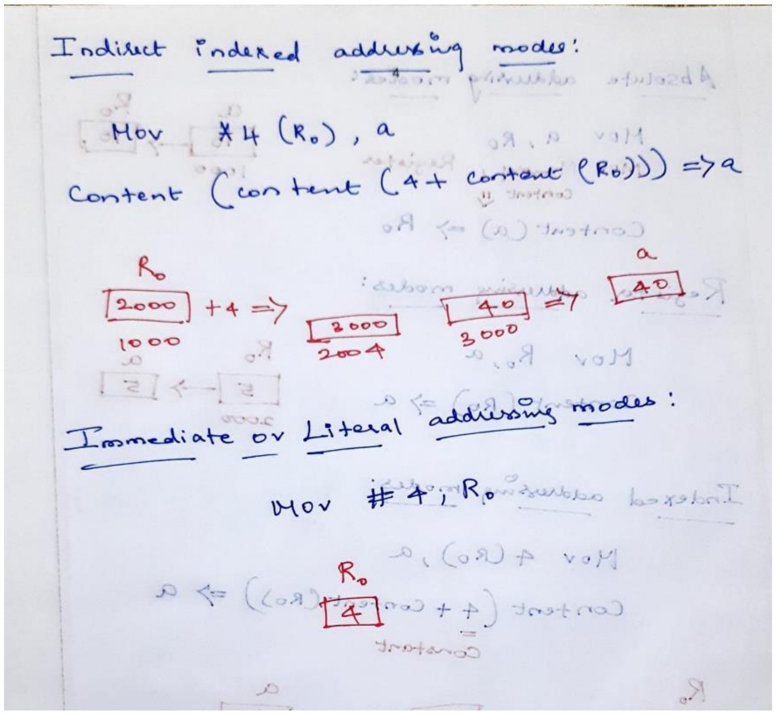
Represents address

Indirect Register addressing modes

Mov *R0, a

Content (Content (R0)) => a

$$\overset{R_0}{\underset{3000}{\boxed{5000}}} \quad \underset{5000}{\boxed{30}} \Rightarrow \overset{a}{\boxed{30}}$$


Indirect indexed addressing modes:

Mov *4 (R0) , a

Content (content (4+ content (R0))) => a

R0
2000
1000
+4 =>
2000
2004
40
3000
=>
a
40

Immediate or Literal addressing modes:

Mov #4, R0

R0
4

Here, cost 1 means that it occupies only one word of memory.
Each instruction has a cost of 1 plus added costs for the source and destination.
Instruction cost = 1 + cost is used for source and destination mode.

Examples:
MOV R0, R1
cost = 1+0+0 = 1

MOV R0, M
cost = 1+0+1 = 2

MOV *4(R0), M
cost = 1+1+1

MOV #1, R0
cost = 1+1+0 = 2

Assuming R0, R1 and R2 contain the addresses of a, b, and c
MOV *R1, *R0   cost = 1+0+0
ADD *R2, *R0   cost = 1+0+0
Therefore, cost = 2

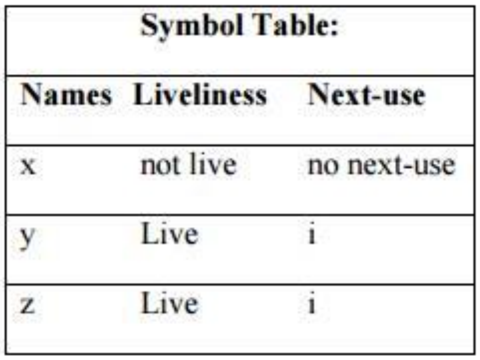
A SIMPLE CODE GENERATOR:

NEXT-USE Information:
If the name in a register is no longer needed, then we remove the name from the register and the register can be used to store some other names.
Input: Basic block B of three-address statements
Output: At each statement i: x= y op z, we attach to i the liveliness and next-uses of x, y and z.

Method: We start at the last statement of B and scan backwards.
1. Attach to statement i the information currently found in the symbol table regarding the next-use and liveliness of x, y and z.

2. In the symbol table, set x to “not live” and “no next use”.

3. In the symbol table, set y and z to “live”, and next-uses of y and z to i.



Symbol Table:		
Names	**Liveliness**	**Next-use**
x	not live	no next-use
y	Live	i
z	Live	i

A SIMPLE CODE GENERATOR
A code generator generates target code for a sequence of three- address statements and effectively uses registers to store operands of the statements.
Example: consider the three-address statement
a := b+c

It can have the following sequence of codes:
ADD Rj, Ri Cost = 1
(or)
ADD c, Ri Cost = 2
(or)
MOV c, Rj Cost = 3
ADD Rj, Ri

Register and Address Descriptors:
A register descriptor - is used to keep track of what is currently in each register. The register descriptors show that initially all the registers are empty.
An address descriptor - stores the location where the current value of the name can be found at run time.

A code-generation algorithm:
The algorithm takes as input a sequence of three-address statements constituting a basic block. For each three-address statement of the form
x : = y op z, perform the following actions:

1. Invoke a function getreg to determine the location L where the result of the computation y op z should be stored.

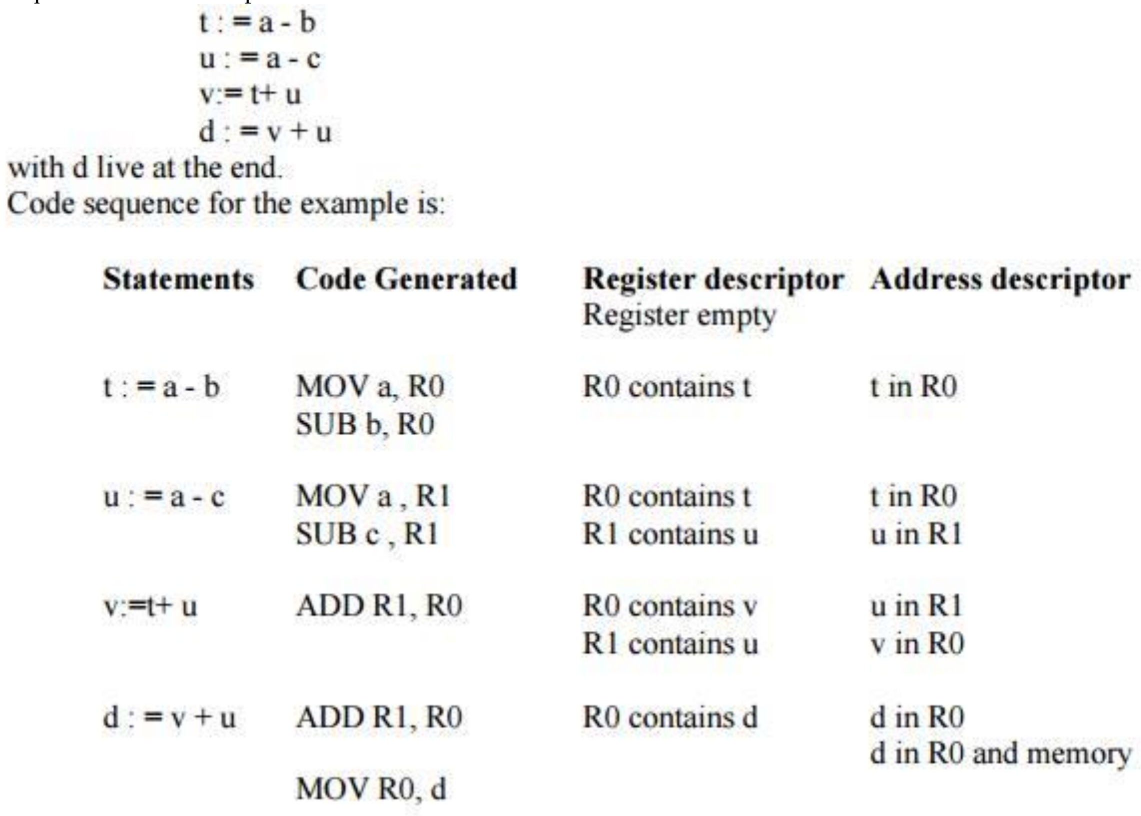
2. Consult the address descriptor for y to determine y’, the current location of y. Prefer the register for y’ if the value of y is currently both in memory and a register. If the value of y is not already in L, generate the instruction MOV y’, L to place a copy of y in L.

3. Generate the instruction OP z’, L where z’ is a current location of z. Prefer a register to a memory location if z is in both. Update the address descriptor of x to indicate that x is in location L. If x is in L, update its descriptor and remove x from all other descriptors.

4. If the current values of y or z have no next uses, are not live on exit from the block, and are in registers, alter the register descriptor to indicate that, after execution of x : = y op z , those registers will no longer contain y or z

Generating Code for Assignment Statements:
The assignment d : = (a-b) + (a-c) + (a-c) might be translated into the following three-address code sequence:

Code sequence for the example is:



```
t : = a - b
u : = a - c
v:= t+ u
d : = v + u
```

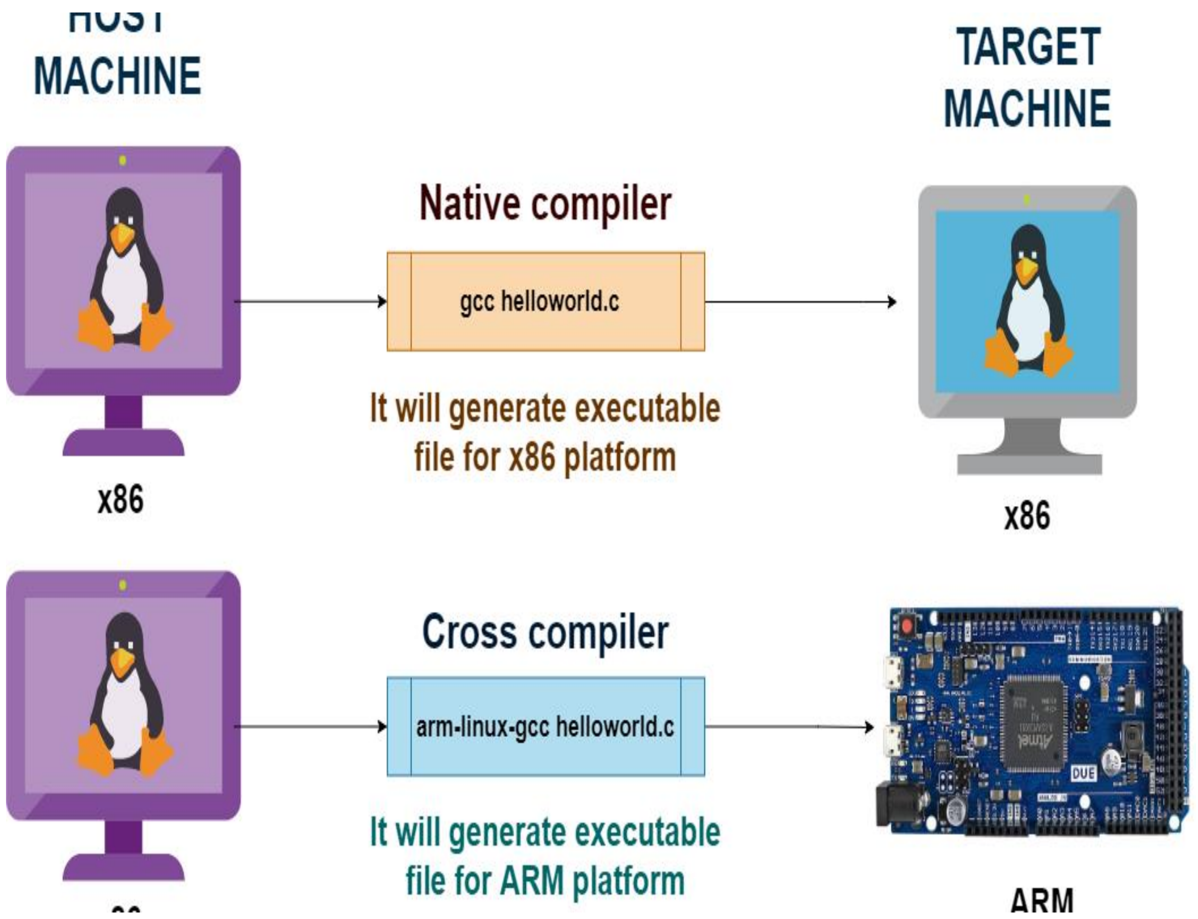
with d live at the end.

Code sequence for the example is:

Statements	Code Generated	Register descriptor	Address descriptor
		Register empty	
t : = a - b	MOV a, R0 SUB b, R0	R0 contains t	t in R0
u : = a - c	MOV a , R1 SUB c , R1	R0 contains t R1 contains u	t in R0 u in R1
v:=t+ u	ADD R1, R0	R0 contains v R1 contains u	u in R1 v in R0
d : = v + u	ADD R1, R0 MOV R0, d	R0 contains d	d in R0 d in R0 and memory

CROSS COMPILER:

A cross compiler is a compiler capable of creating executable code for a platform other than the one on which the compiler is running. For example, a compiler that runs on a Windows 7 PC but generates code that runs on Android smartphone is a cross compiler.


HOST MACHINE
TARGET MACHINE
Native compiler
gcc helloworld.c
It will generate executable file for x86 platform
x86
x86
Cross compiler
arm-linux-gcc helloworld.c
It will generate executable file for ARM platform
ARM

T DIAGRAMS

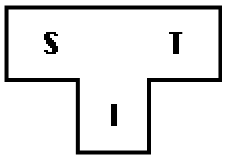
Bootstrapping:

- Bootstrapping is widely used in the compilation development.
- Bootstrapping is used to produce a self-hosting compiler. Self-hosting compiler is a type of compiler that can compile its own source code.
- Bootstrap compiler is used to compile the compiler and then you can use this compiled compiler to compile everything else as well as future versions of itself.

A compiler can be characterized by three languages:

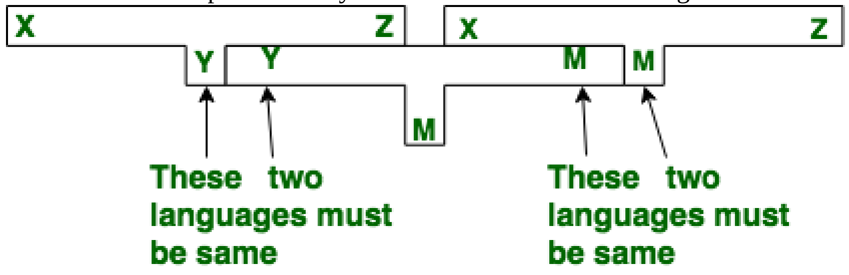
- Source Language
- Target Language
- Implementation Language

The T- diagram shows a compiler ${}^{S}C_{I}^{T}$ for Source S, Target T, implemented in I.



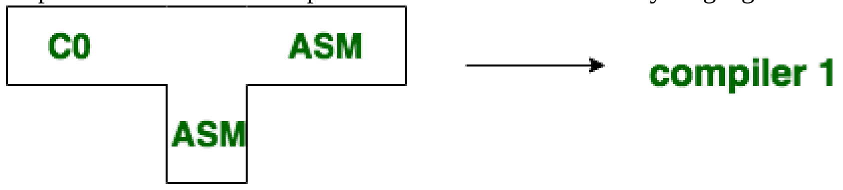
Suppose we want to write a cross compiler for new language X. The implementation language of this compiler is say Y and the target code being generated is in language Z. That is, we create XYZ. Now if existing compiler Y runs on machine M and generates code for M then it is denoted as YMM. Now if we run XYZ using YMM then we get a compiler XMZ. That means a compiler for source language X that generates a target code in language Z and which runs on machine M.

Example:
We can create compiler of many different forms. Now we will generate.

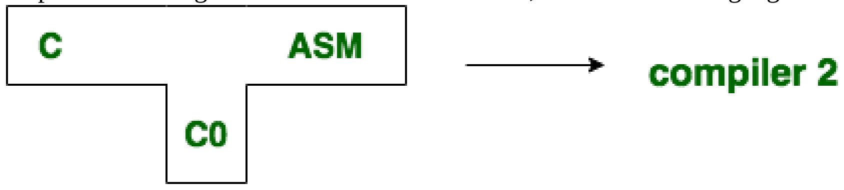

X
Z
Y
Y
M
M
X
Z
M
These two languages must be same
These two languages must be same

Compiler which takes C language and generates an assembly language as an output with the availability of a machine of assembly language.

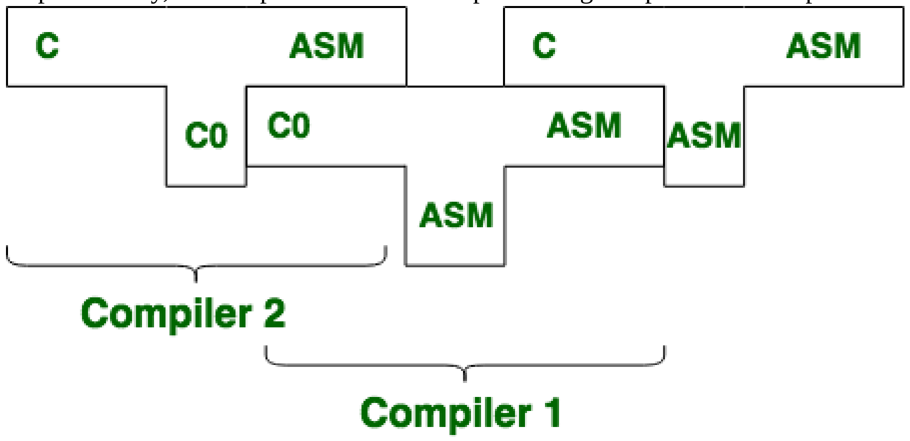
Step-1: First we write a compiler for a small of C in assembly language.



Step-2: Then using with small subset of C i.e. C0, for the source language c the compiler is written.



Step-3: Finally, we compile the second compiler. using compiler 1 the compiler 2 is compiled.



Step-4: Thus, we get a compiler written in ASM which compiles C and generates code in ASM.