Unit IV

Intermediate Code Generation

- 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:
- Because of the machine-independent intermediate code, portability will be enhanced.
- Retargeting is facilitated.
- It is easier to apply source code modification to improve the performance of source code by optimizing the intermediate code.

Intermediate Code Generation can be represented in two forms:

1. Linear form
2. Tree form

Linear Form: The following are commonly used Linear form intermediate code representations:

a. Prefix notation

b. Postfix notation

c. Three address code

Tree Form: The following are commonly used Linear form intermediate code representations:

a. Syntax tree

b. Direct acyclic graph (DAG)

- **Prefix Notation:** Also known as Polish notation or prefix notation. The ordinary (infix) way of writing the sum of a and b is with an operator in the middle: a + b The prefix notation for the same expression places the operator at the left end as + ab. In general, if e1 and e2 are any postfix expressions, and + is any binary operator, the result of applying + to the values denoted by e1 and e2 is prefix notation by + e1e2. No parentheses are needed in prefix notation because the position and arity (number of arguments) of the operators permit only one way to decode a prefix expression. In prefix notation, the operator follows the operand.
  Example 1: The prefix representation of the expression (a + b) * c is : * + abc
  Example 2: The prefix representation of the expression (a – b) * (c + d) + (a – b) is : + * -ab +cd -ab

- **Postfix Notation:** Also known as reverse Polish notation or suffix notation. The ordinary (infix) way of writing the sum of a and b is with an operator in the middle: a + b The postfix notation for the same expression places the operator at the right end as ab +. In general, if e1 and e2 are any postfix expressions, and + is any binary operator, the result of applying + to the values denoted by e1 and e2 is postfix notation by e1e2 +. No parentheses are needed in postfix notation because the position and arity (number of arguments) of the operators permit only one way to decode a postfix expression. In postfix notation, the operator 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 -+

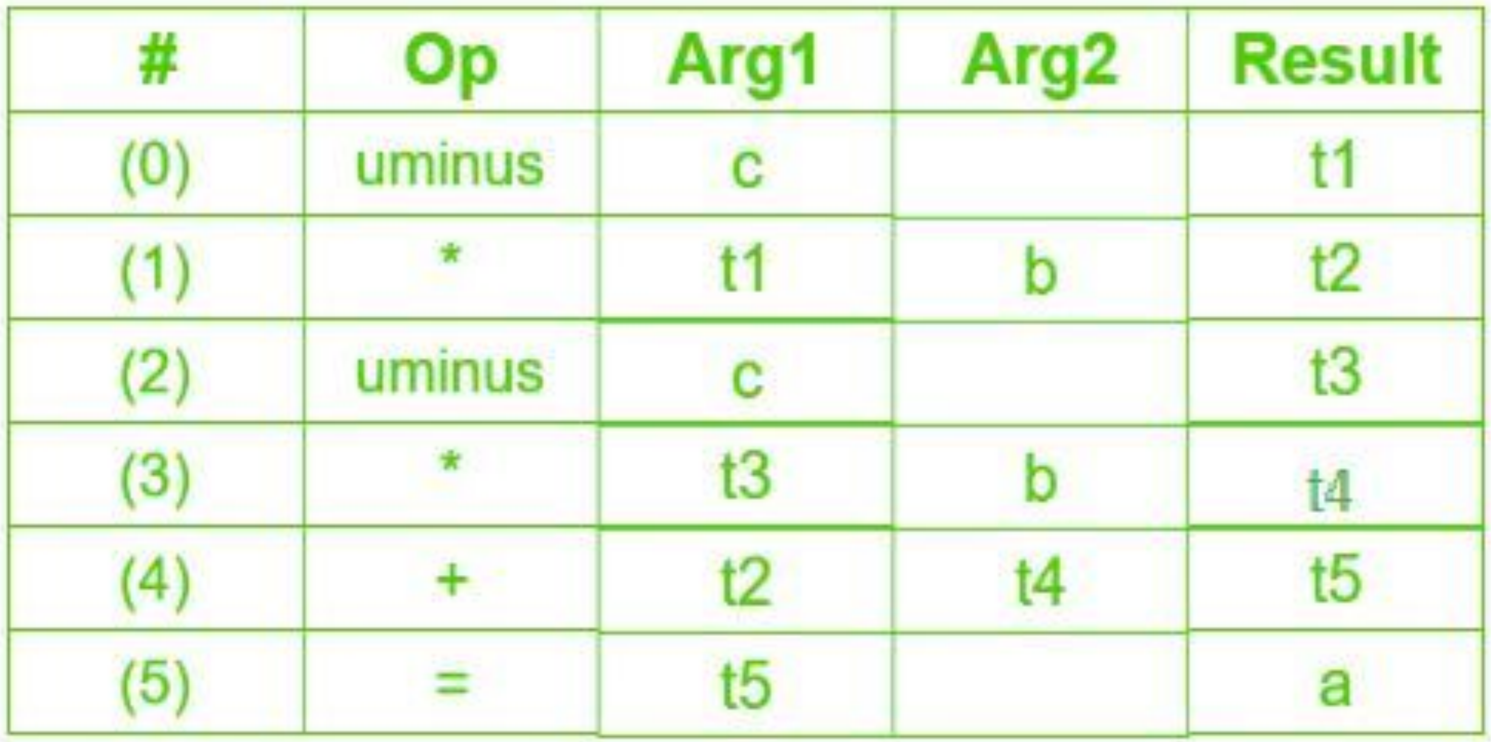
- **Three-Address Code:** A statement involving no more than three references(two for operands and one for result) is known as a three address statement.
- A sequence of three address statements is known as a three address code. Three address statement is of form **x = y op z**, where x, y, and z will have address (memory location). **At most one operator**
  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 , T 3 are temporary variables. There are 3 ways to represent a Three-Address Code in compiler design:
  i) Quadruples
  ii) Triples
  iii) 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
t2 = b * t1
t3 = uminus c
t4 = b * t3
t5 = t2 + t4
 a = t5
```



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

Quadruple representation

2. Triples – This representation doesn’t 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 consist 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)

```
t1 = uminus c
  t2 = b * t1
  t3 = uminus c
  t4 = b * t3
  t5 = t2 + t4
   a = t5
```

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)

t1 = uminus c

t2 = b * t1

t3 = uminus c

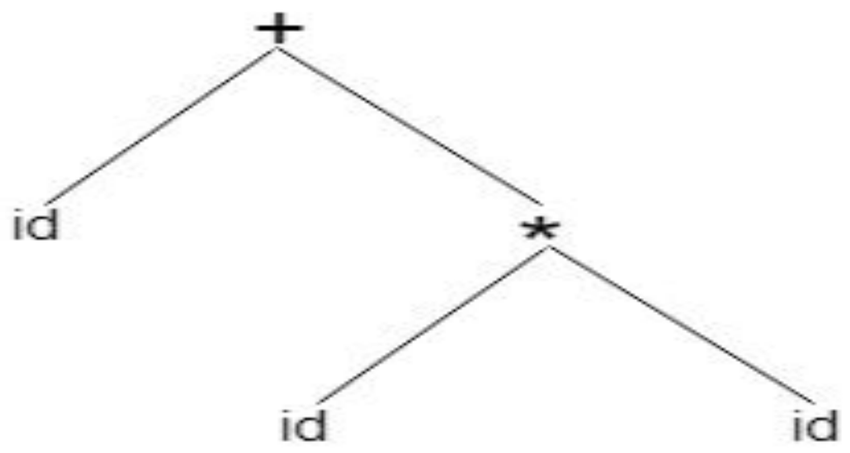
t4 = b * t3

t5 = t2 + t4

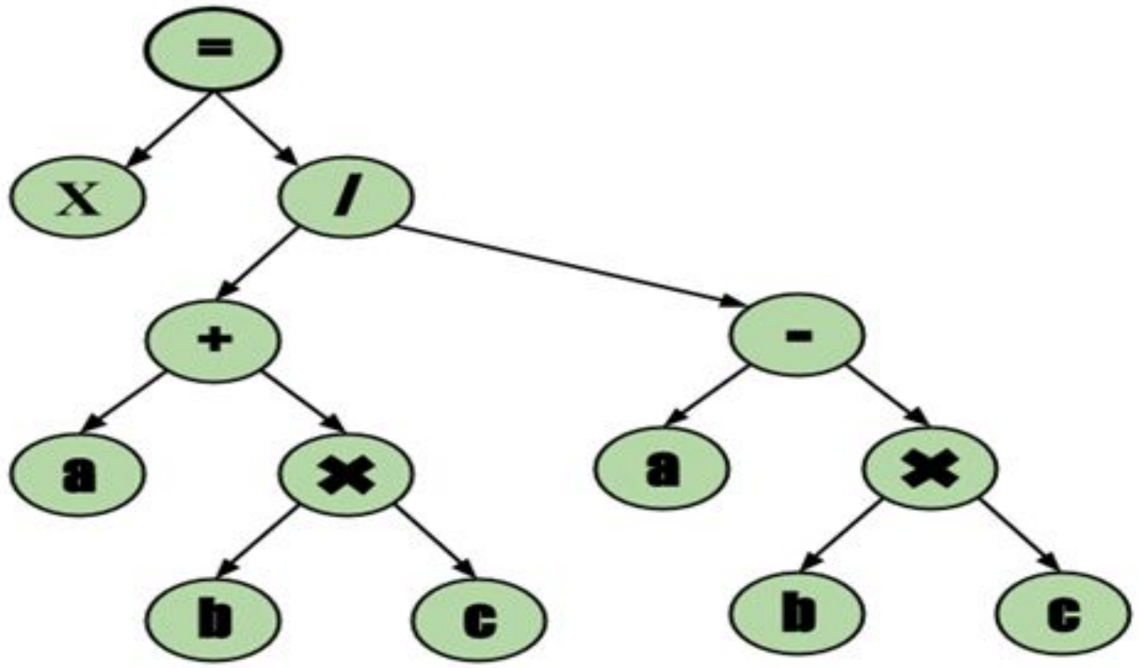
a = t5

Syntax tree

- In the parse tree, most of the leaf nodes are single child to their parent nodes.
- In the syntax tree, we can eliminate this extra information.
- Syntax tree is a variant of parse tree. In the syntax tree, interior nodes are operators and leaves are operands.
- Syntax tree is usually used when represent a program in a tree structure.
- A sentence **id + id * id** would have the following syntax tree:



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



Basic Block and Control Flow Graph:

A *basic block* is a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without halt or possibility of branching except at the end.

- The following sequence of three address code forms a basic block:

```
t1 = a * a

t2 = a * b

t3 = t1 + t2
```

- The first task is to partition a sequence of three-address code into basic blocks. A new basic block is begun with the first instruction and instructions are added until a jump or a label is met. In the absence of a jump, control moves further consecutively from one instruction to another.

Partition Algorithm for Basic Blocks :

Partitioning three-address code into basic blocks.

Input: A sequence of three address instructions.

Process: Identify leaders first. Following are the rules used for finding a leader:

1. The first three-address instruction of the intermediate code is a leader.
2. Instructions that are targets of unconditional or conditional jump/goto statements are leaders.
3. Instructions that immediately follow unconditional or conditional jump/goto statements are considered leaders.

Make basic block from leader to line before next leader.

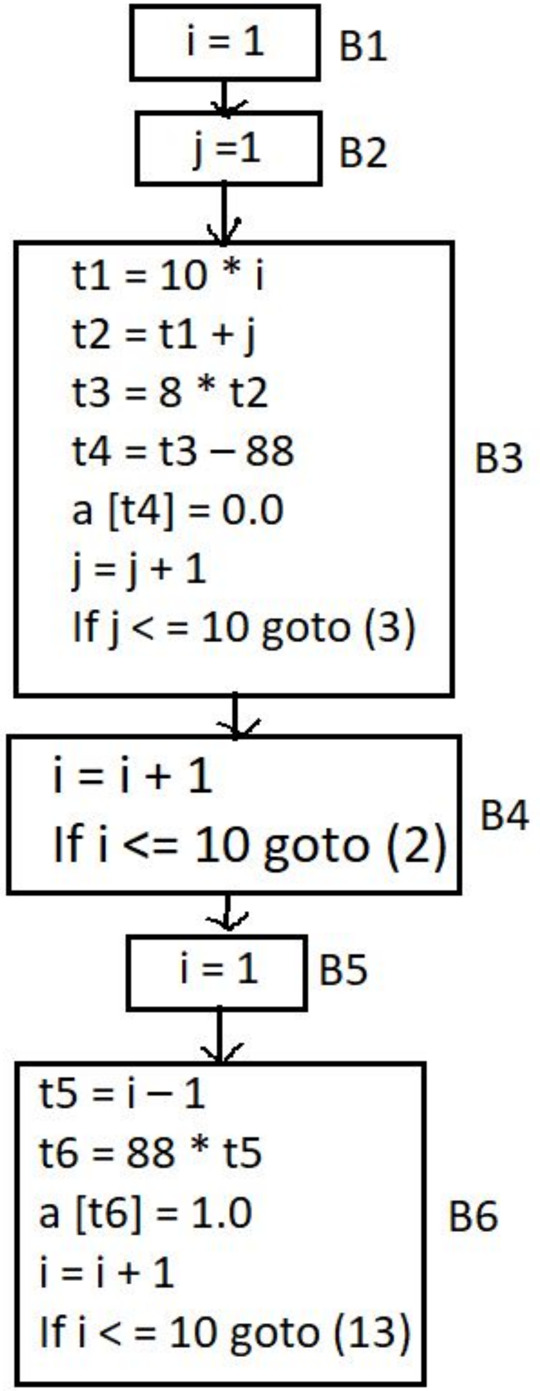
Example 1: *The following sequence of three-address statements forms a basic block:*

```
t1 := a*a
t2 := a*b
t3 := 2*t2
t4 := t1+t3
t5 := b*b
t6 := t4 +t5
```

A three address statement x:= y + z is said to define x and to use y and z. A name in a basic block is said to be live at a given point if its value is used after that point in the program, perhaps in another basic block.

Example of Basic block:

1) i = 1
2) j = 1
3) t1 = 10 * i
4) t2 = t1 + j
5) t3 = 8 * t2
6) t4 = t3 – 88
7) a [t4] = 0.0
8) j = j + 1
9) If j < = 10 goto (3)
10) i = i + 1
11) If i <= 10 goto (2)
12) i = 1
13) t5 = i – 1
14) t6 = 88 * t5
15) a [t6] = 1.0
16) i = i + 1
17) If i < = 10 goto (13)



```
B1:
i = 1

B2:
j =1

B3:
t1 = 10 * i
t2 = t1 + j
t3 = 8 * t2
t4 = t3 – 88
a [t4] = 0.0
j = j + 1
If j < = 10 goto (3)

B4:
i = i + 1
If i <= 10 goto (2)

B5:
i = 1

B6:
t5 = i – 1
t6 = 88 * t5
a [t6] = 1.0
i = i + 1
If i < = 10 goto (13)
```

$$\begin{tikzcd}
\text{start} \arrow[d] \\
B1 \arrow[d] \\
B2 \arrow[d] \\
B3 \arrow[d] \arrow[loop right] \\
B4 \arrow[d] \arrow[uu, bend right=60] \\
B5 \arrow[d] \\
B6 \arrow[d] \arrow[loop right] \\
\text{exit}
\end{tikzcd}$$

Control Flow Graph

Directed Acyclic graph

- 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.

Directed Acyclic Graph Characteristics :

A Directed Acyclic Graph for Basic Block is a directed acyclic graph with the following labels on nodes.

- The graph’s leaves each have a unique identifier, which can be variable names or constants.
- The interior nodes of the graph are labelled with an operator symbol.
- In addition, nodes are given a string of identifiers to use as labels for storing the computed value.

Method:

Step 1:

- If y operand is undefined then create node(y).
- If z operand is undefined then for case(i) create node(z).

Step 2:

- For case(i), create node(OP) whose right child is node(z) and left child is node(y).
- For case(ii), check whether there is node(OP) with one child node(y).
- For case(iii), node n will be node(y).

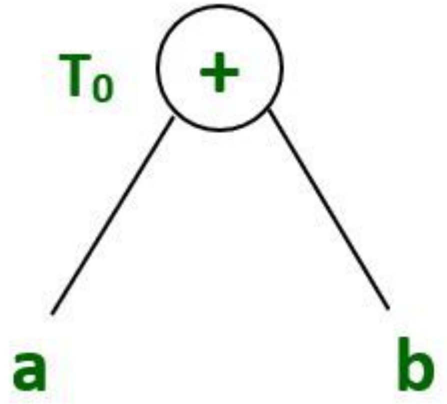
Example :

T0 = a + b —Expression 1

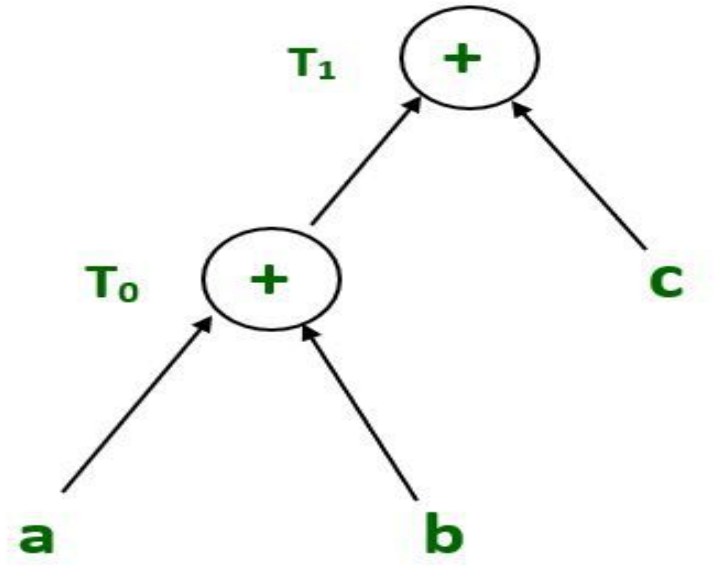
T1 = T0 + c —-Expression 2

d = T0 + T1 ——Expression 3

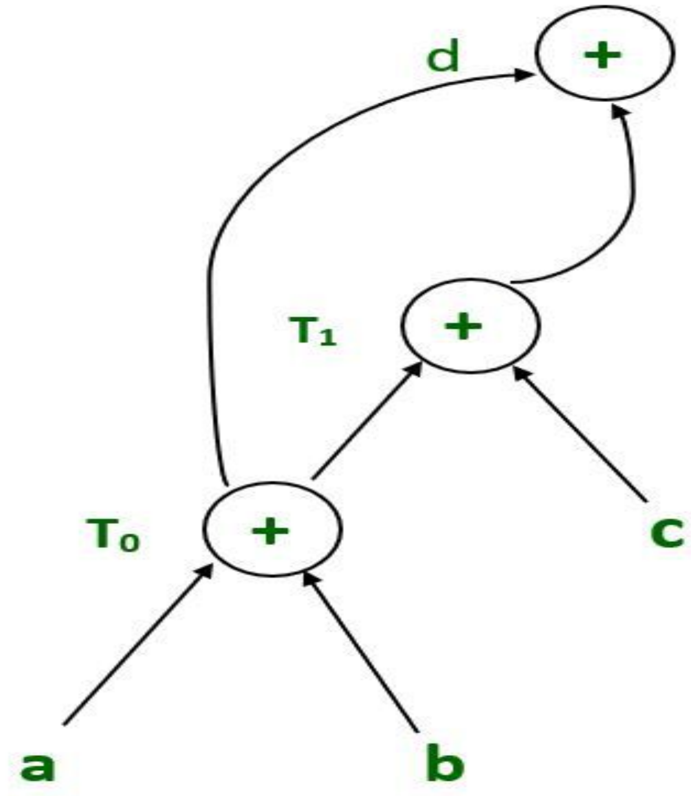
Expression 1 : $T_0 = a + b$



Expression 2: $T_1 = T_0 + c$

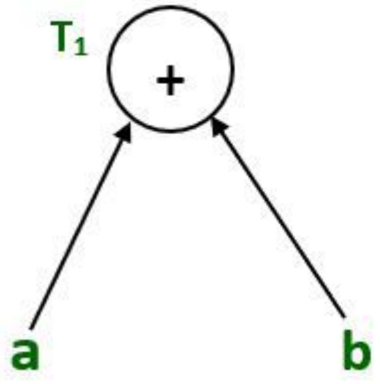


- **Expression 3 : d = T_0 + T_1**

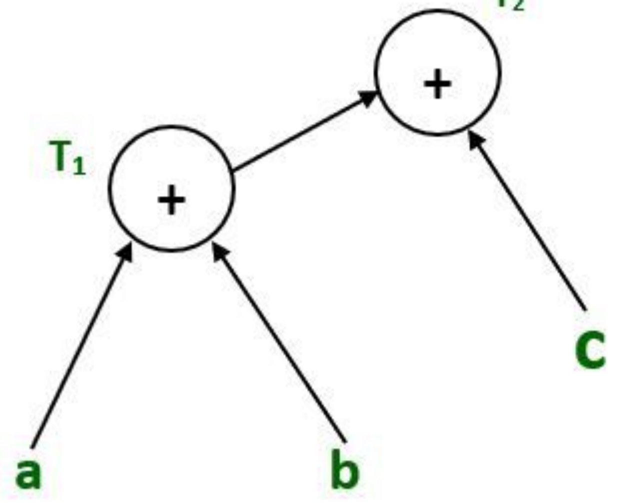


Example :

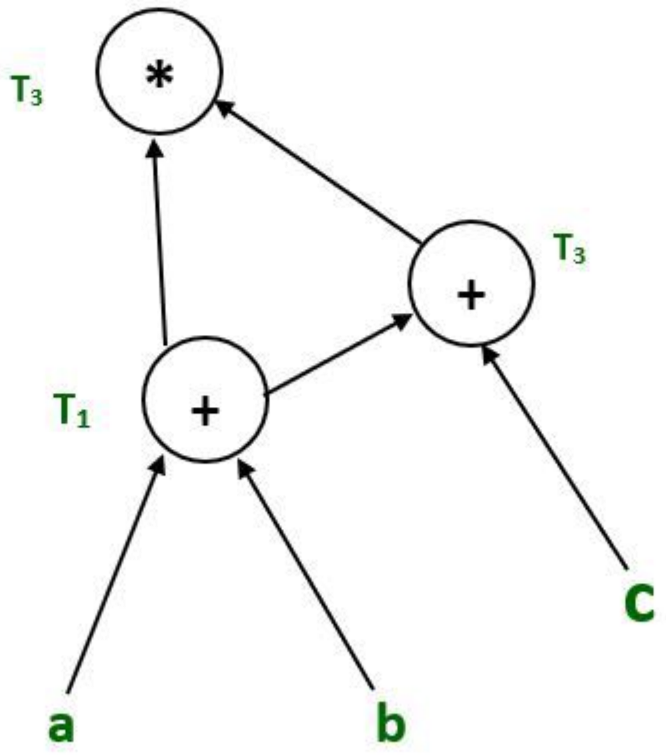
- T1 = a + b
- T2 = T1 + c
- T3 = T1 x T2



$T_1 = a + b$



$T_2 = T_1 + c$



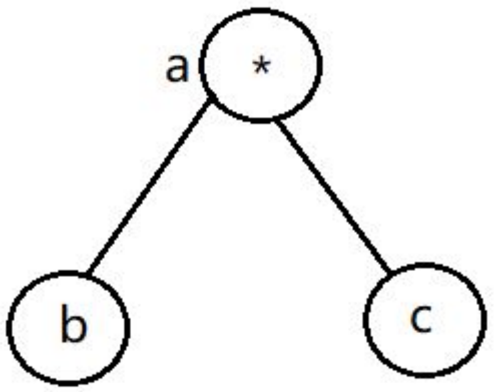
$T_3 = T_1 * T_2$

Algorithm for construction of DAG

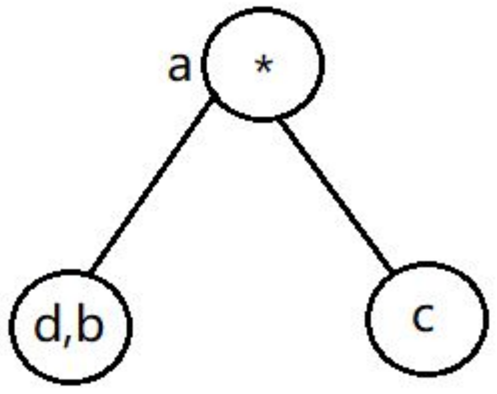
- **Input:** It contains a basic block
- **Output:** It contains the following information:
- Each node contains a label. For leaves, the label is an identifier.
- Each node contains a list of attached identifiers to hold the computed values.
- **Case (i)** x:= y OP z
- **Case (ii)** x:= OP y
- **Case (iii)** x:= y

DAG example:

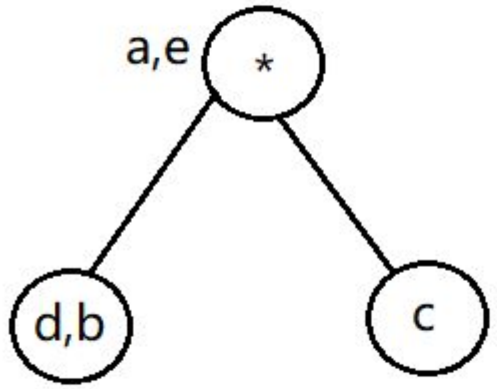
a = b * c
d = b
e = d * c
b = e
f = b + c
g = d + f



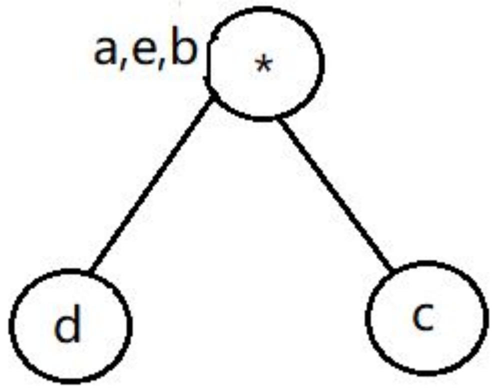
Step 1



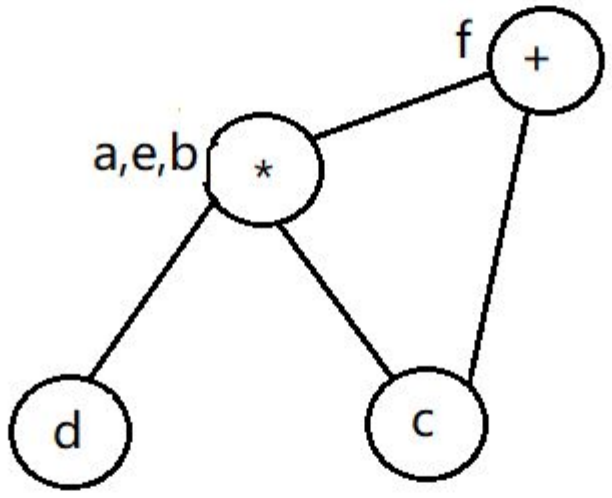
Step 2



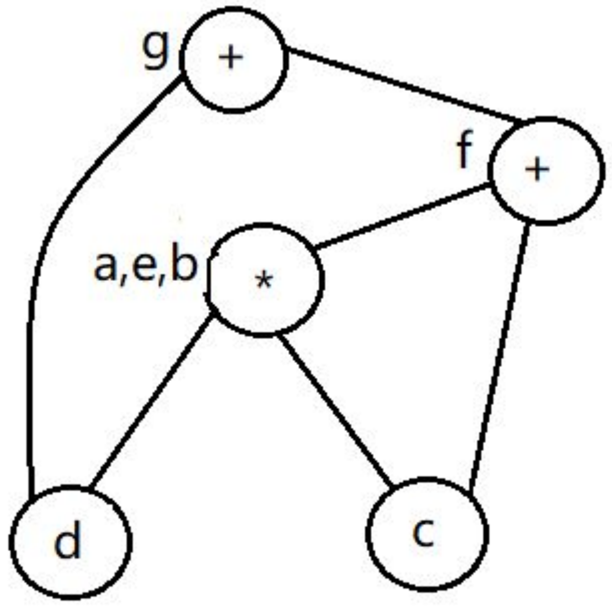
Step 3



Step 4



Step 5



Step 6

Example:

a. S1:= 4 * i

b. S2:= a[S1]

c. S3:= 4 * i

d. S4:= b[S3]

e. S5:= s2 * S4

f. S6:= prod + S5

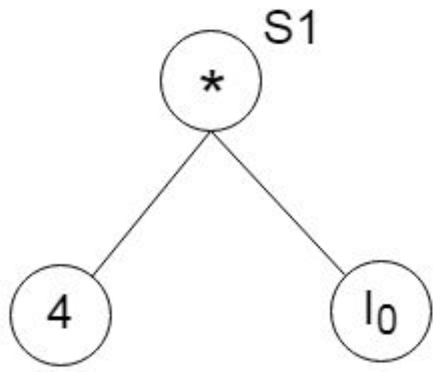
g. Prod:= s6

h. S7:= i+1

i. i := S7

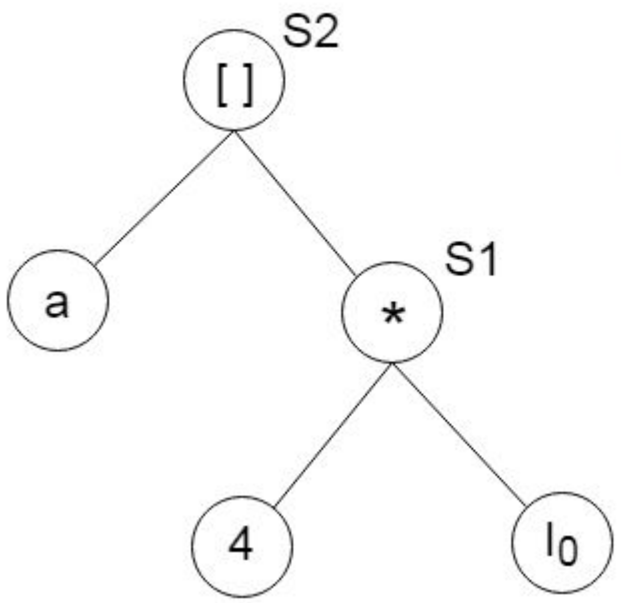
j. if i<= 20 **goto** (1)

(a)



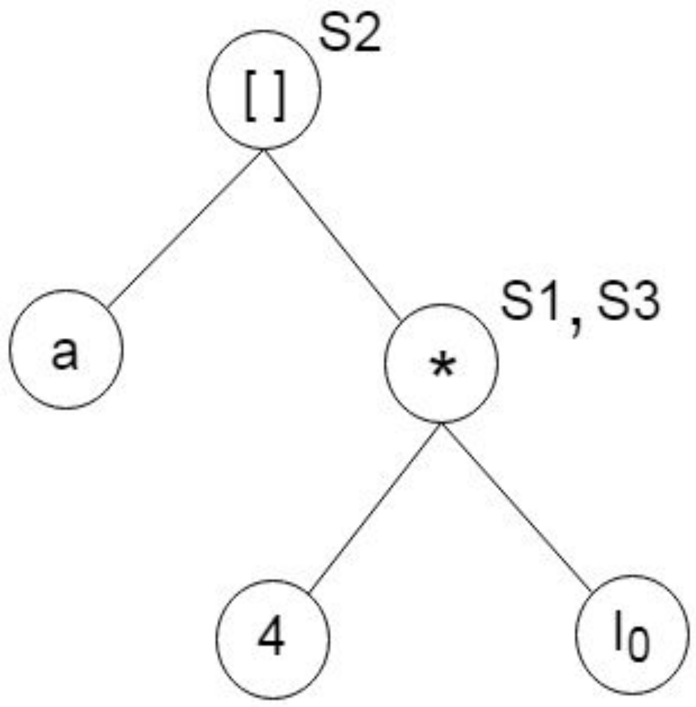
Statement (1)

(b)



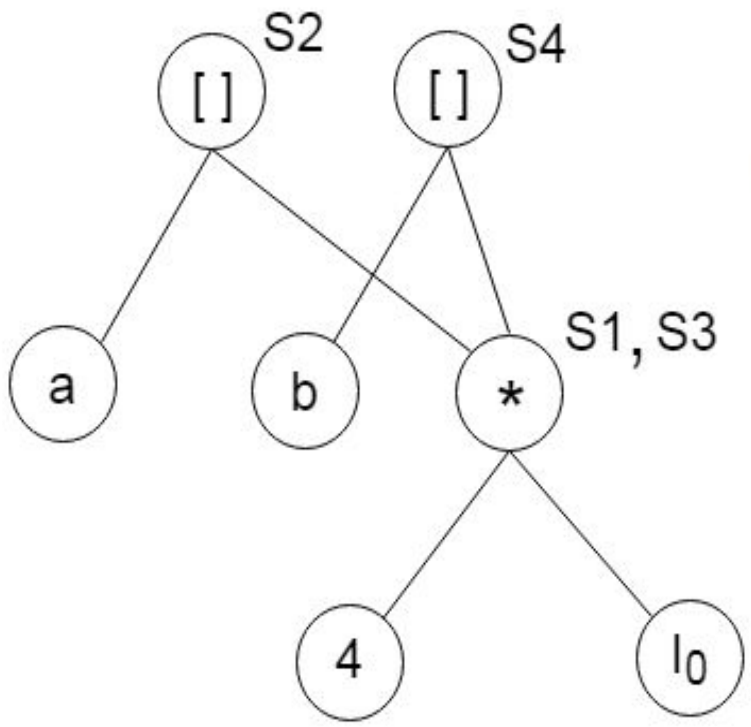
Statement (2)

(c)



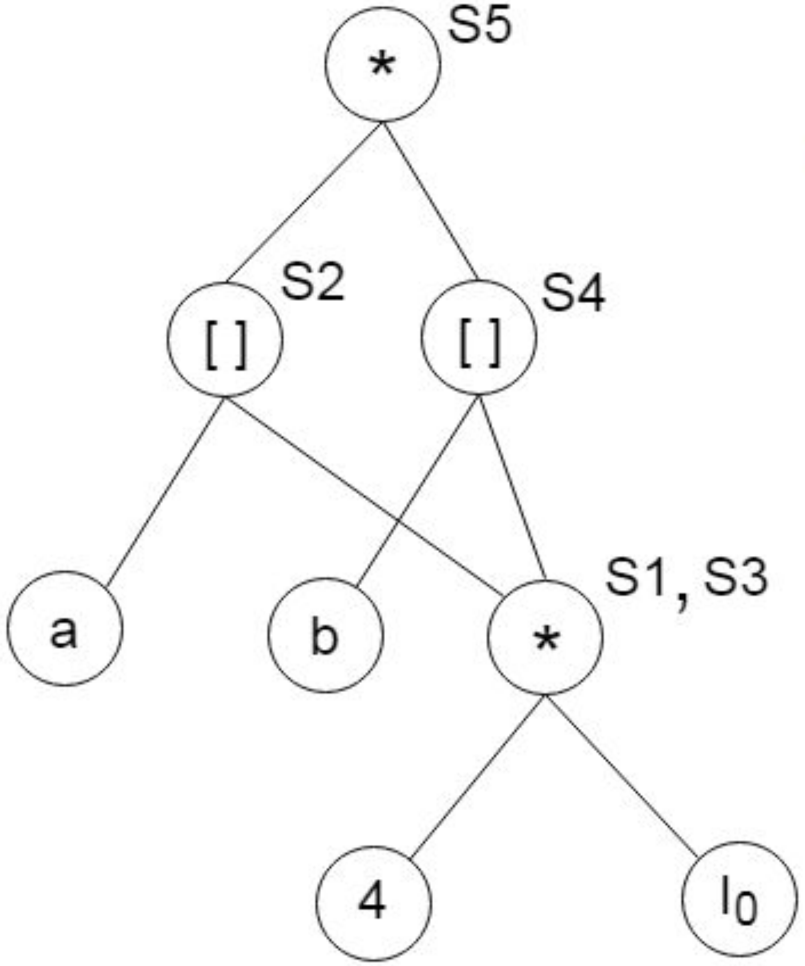
4 * I0 node exist already hence attach identifier S3 to the existing node for statement (3)

(d)



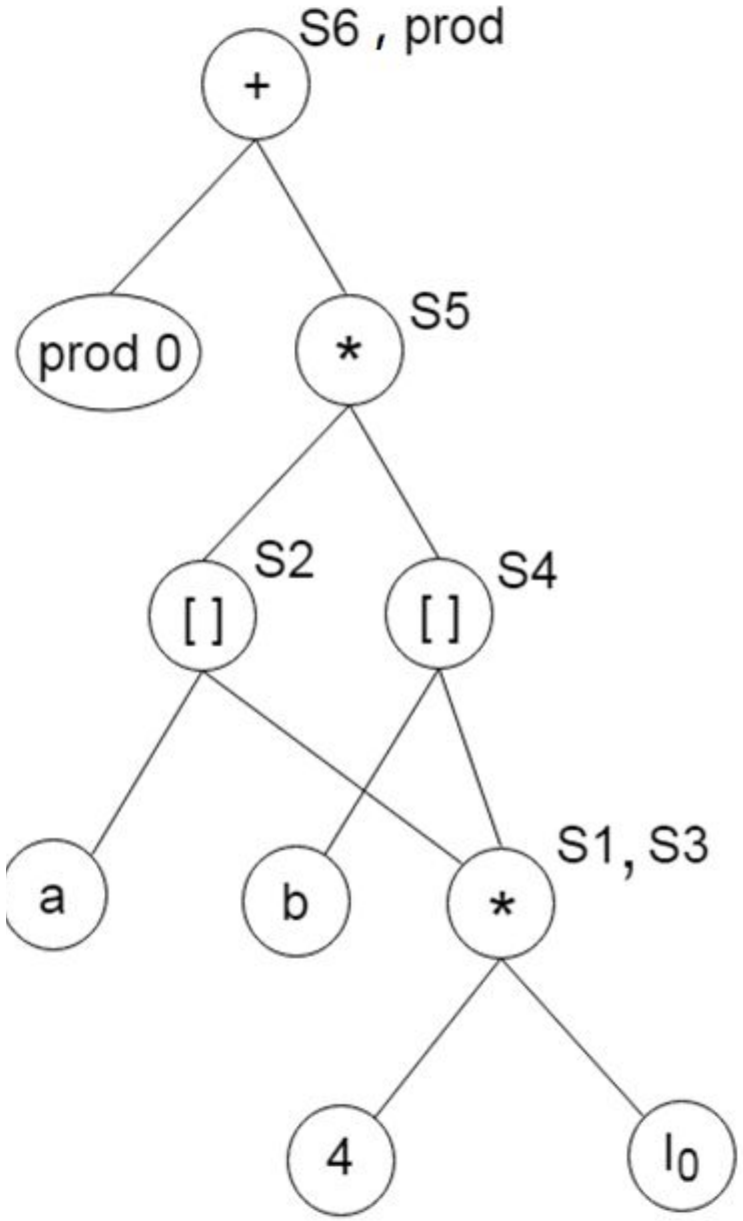
Statement (4)

(e)



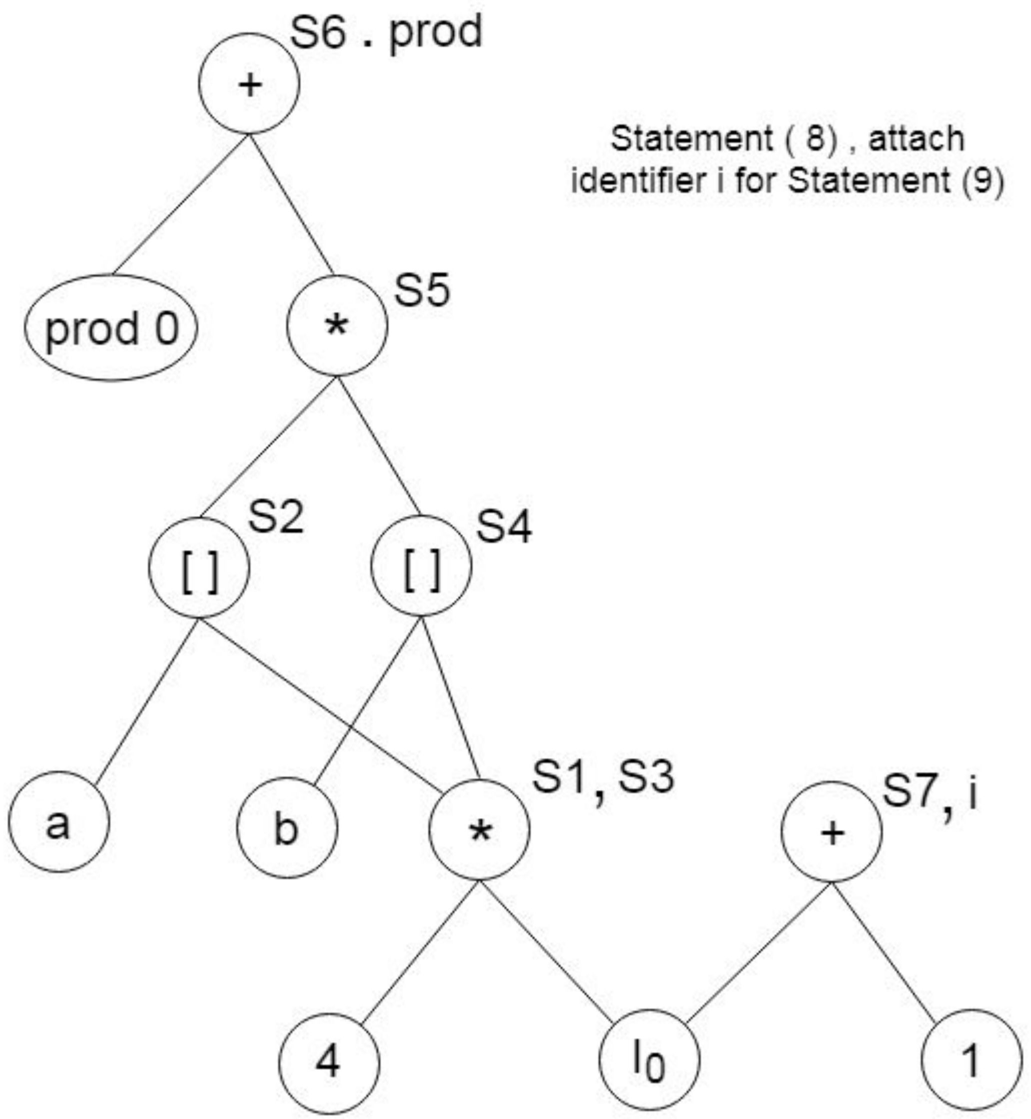
Statement (5)

(f)



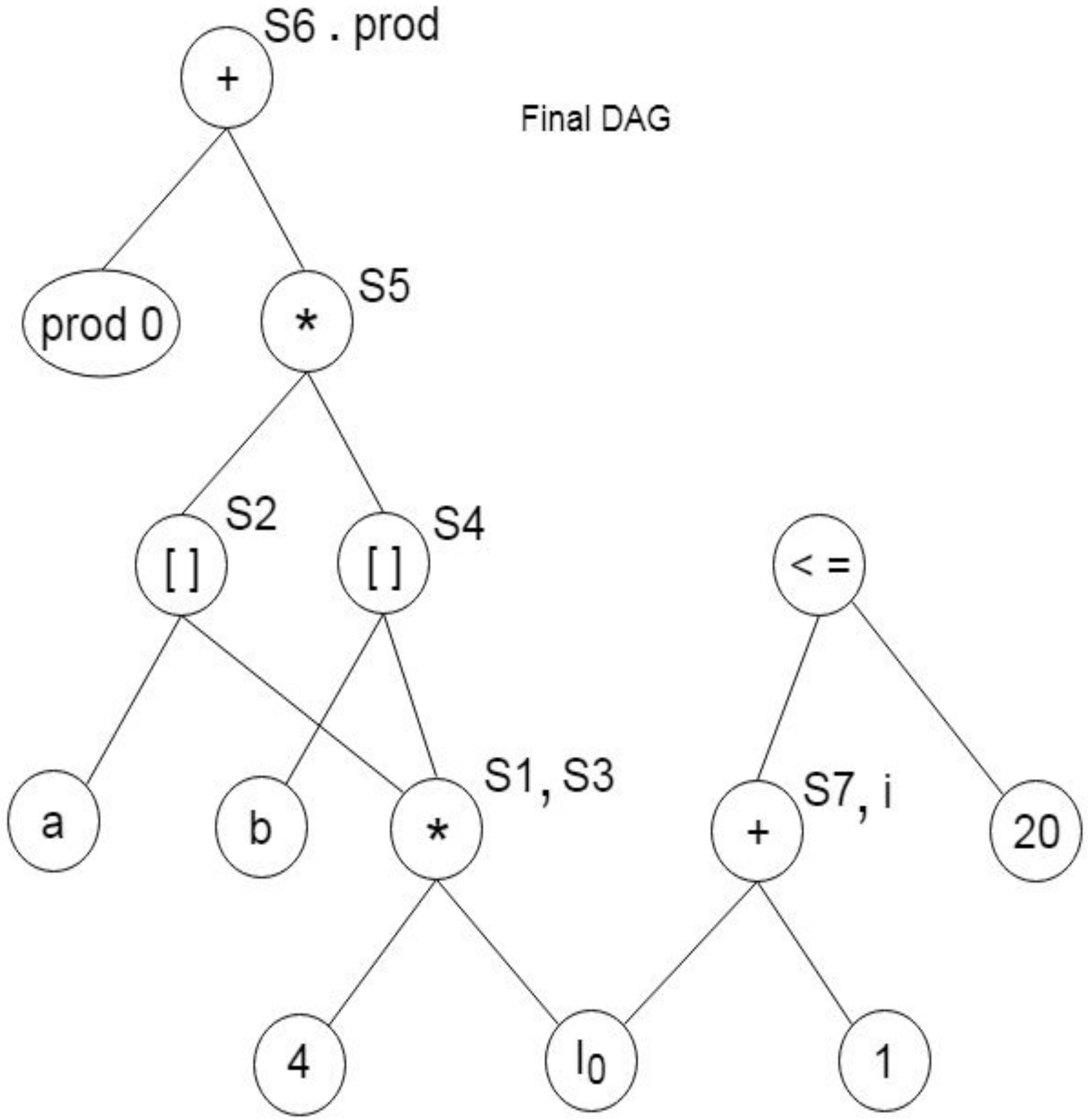
Statement (6), attach identifier prod for Statement (7)

(g)



Statement (8) , attach identifier i for Statement (9)

(h)



Final DAG

Application of Directed Acyclic Graph:

- Directed acyclic graph determines the subexpressions that are commonly used.
- Directed acyclic graph determines the names used within the block as well as the names computed outside the block.
- Determines which statements in the block may have their computed value outside the block.
- Code can be represented by a Directed acyclic graph that describes the inputs and outputs of each of the arithmetic operations performed within the code; this representation allows the compiler to perform common subexpression elimination efficiently.
- Several programming languages describe value systems that are linked together by a directed acyclic graph. When one value changes, its successors are recalculated; each value in the DAG is evaluated as a function of its predecessors.

Syntax directed translation

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

  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

2 types:

1. Syntax directed translation definition
2. Syntax directed translation scheme

Types of attributes – Attributes may be of two types –

- Synthesized
- Inherited

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 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.

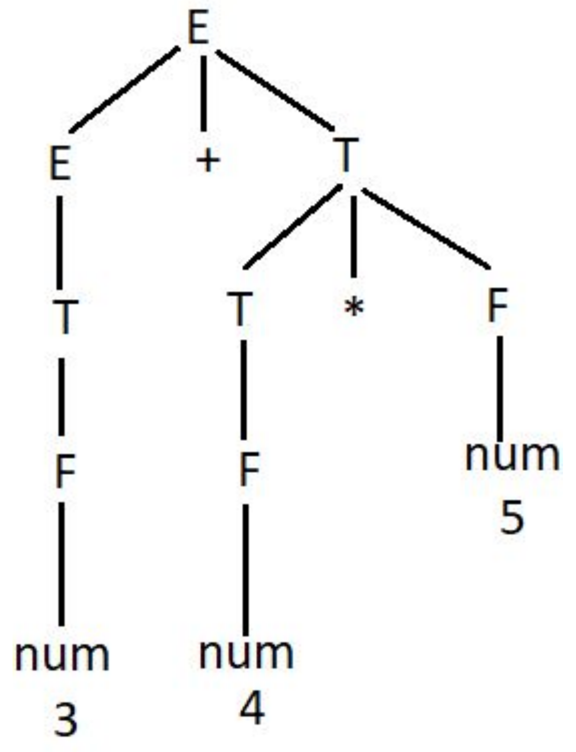
Inherited attributes – An attribute of a nonterminal on the right-hand side of a production is called an inherited attribute. The attribute can take value either from its parent or from its siblings (variables in the LHS or RHS of the production). For example, let’s say A -> BC is a production of a grammar and B’s attribute is dependent on A’s attributes or C’s attributes then it will be inherited attribute.

A -> BC; C = A

Syntax directed translation definition

Production	Semantic Rules
$E \rightarrow E + T$	E.val := E.val + T.val
$E \rightarrow T$	E.val := T.val
$T \rightarrow T * F$	T.val := T.val + F.val
$T \rightarrow F$	T.val := F.val
$F \rightarrow num$	F.val := num.lexval

Example: 3 + 4 * 5 = 23



E.val is one of the attributes of E.

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

Syntax directed translation scheme

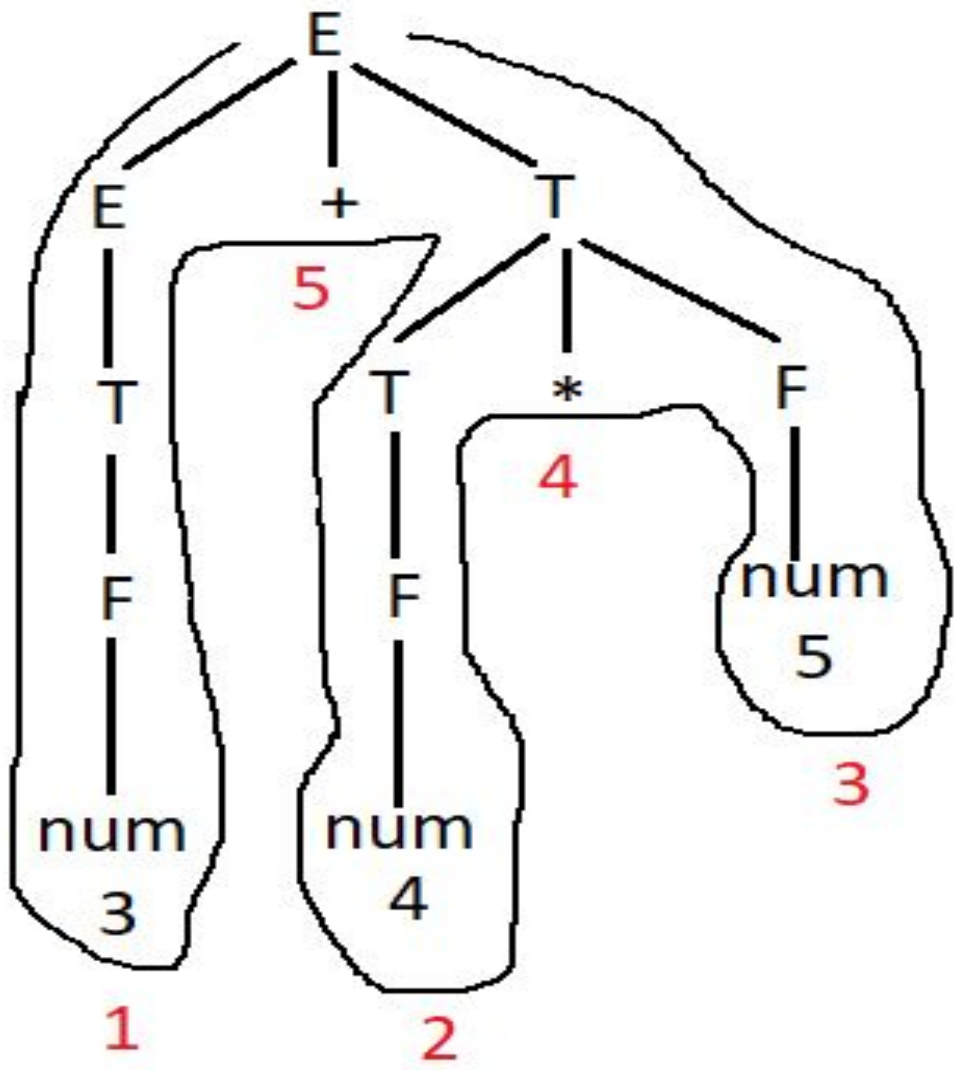
- The Syntax directed translation scheme is a context -free grammar.
- The syntax directed translation scheme is used to evaluate the order of semantic rules.
- In translation scheme, the semantic rules are embedded within the right side of the productions.
- The position at which an action is to be executed is shown by enclosed between braces. It is written within the right side of the production.

Example: Infix to Postfix

Expression	Action
E -> E + T	{print('+')}
E -> T	{ }
T -> T * F	{ print(' * ')}
T -> F	{ }
F -> num	{ print(LexValue)}

Input String: 3 + 4 * 5

Output: 345 * +



Types of SDT:

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.

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.

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.

Backpatching

- 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:

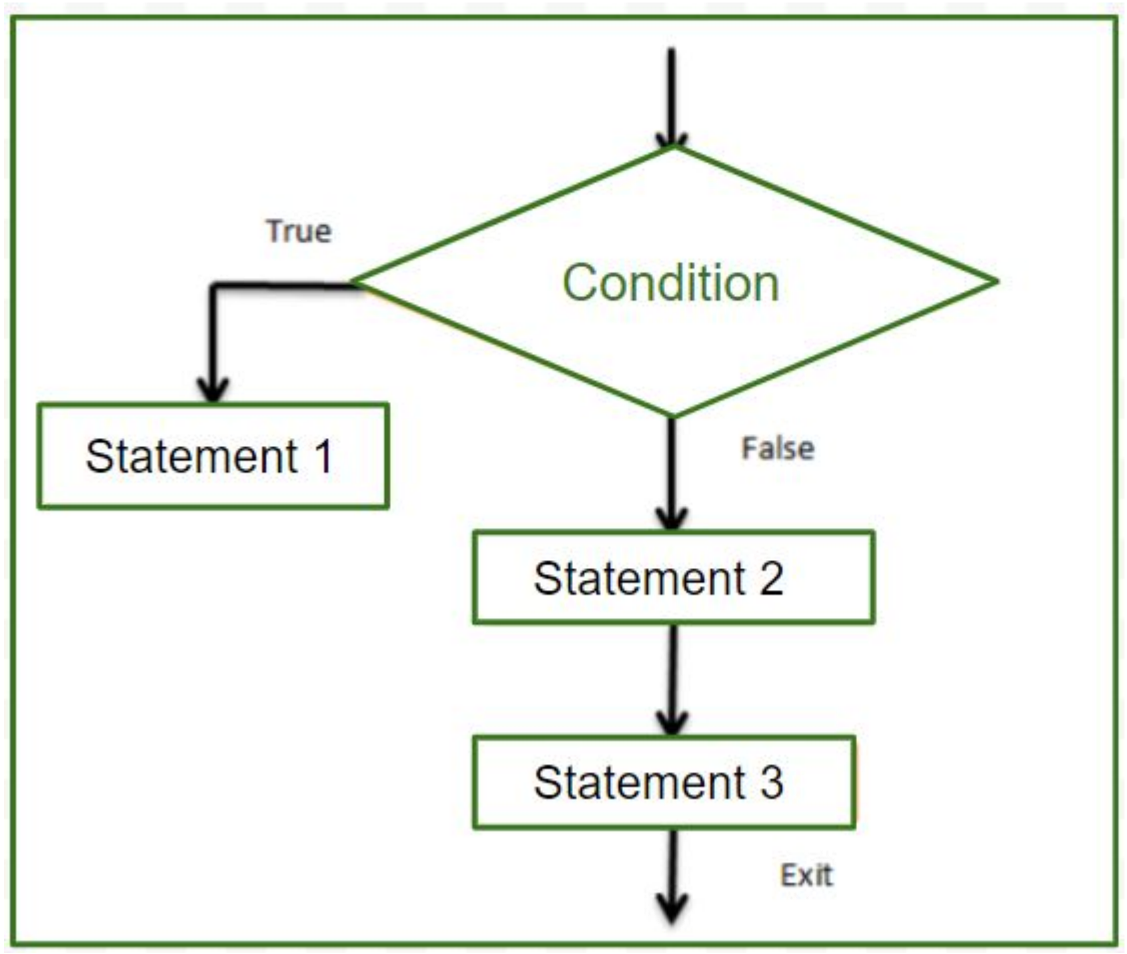
1. Boolean expression: Boolean expressions are statements whose results can be either true or false. A boolean expression which is named for mathematician George Boole is an expression that evaluates to either true or false. Let’s look at some common language examples:

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:

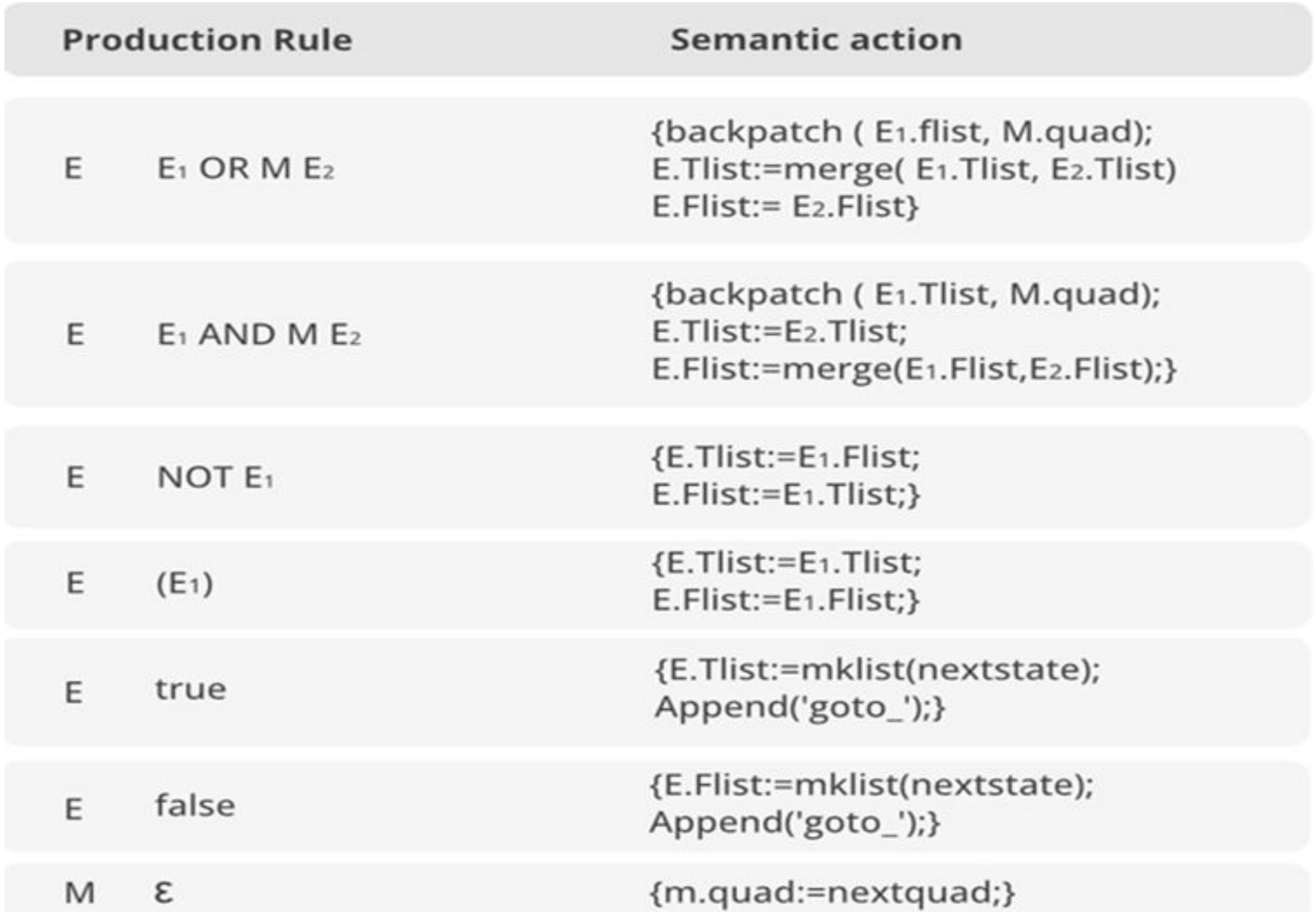


First Pass

```
x < 100 | | y > 200 && x ! = y
100 : if x < 100 then _______
101 : goto ______
102 : if y > 200 then_______
103 : goto ________
104 : if x ! = y then
105 : goto _______
106 : true
107 : false
```

Second Pass

```
x < 100 | | y > 200 && x ! = y
100 : if x < 100 then 106
101 : goto 102
102 : if y > 200 then 104
103 : goto 107
104 : if x ! = y then 106
105 : goto 107
106 : true
107 : false
```



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 true	{E.Tlist:=mklist(nextstate);
	Append('goto_');}

E false	{E.Flist:=mklist(nextstate);
	Append('goto_');}

M ε	{m.quad:=nextquad;}

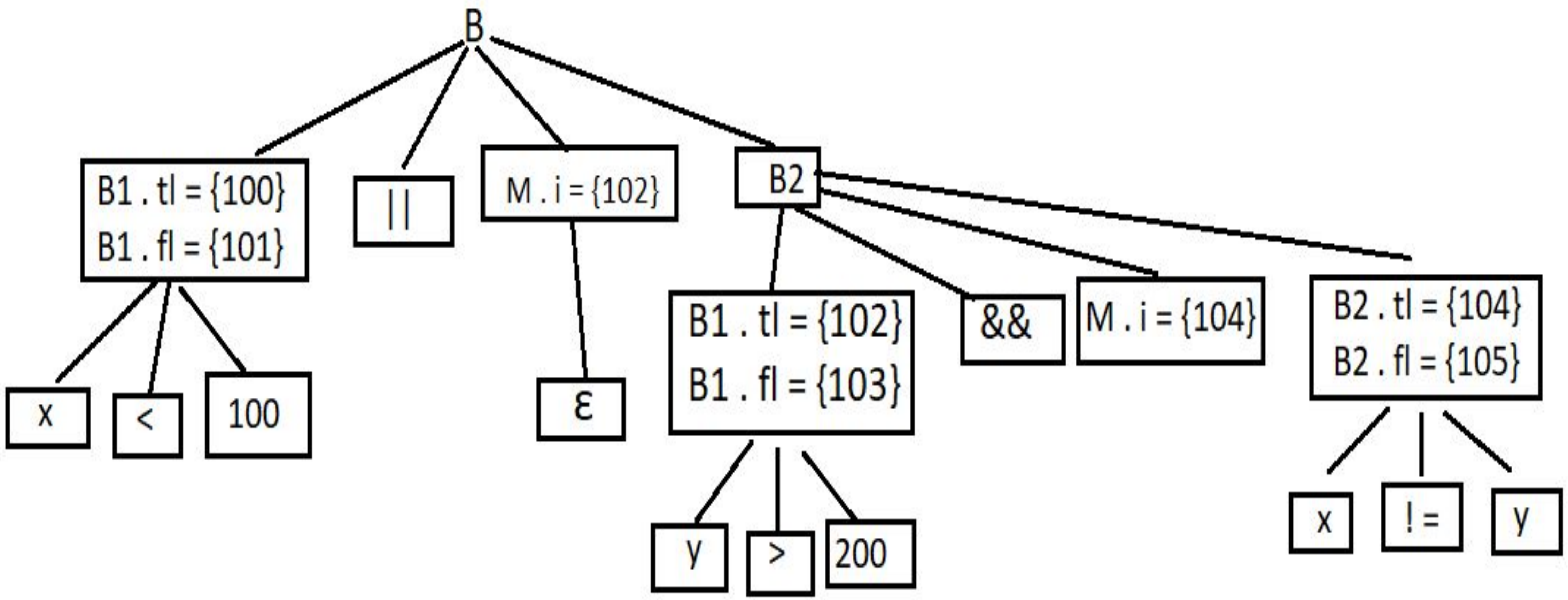
- B. truelist (tl)= {100, 102, 104, 106}
- B. falselist (fl)= {101, 103, 105, 107}
- **Backpatch (p, i):** Inserts i as the target label for each of the instructions on the record pointed to by p.

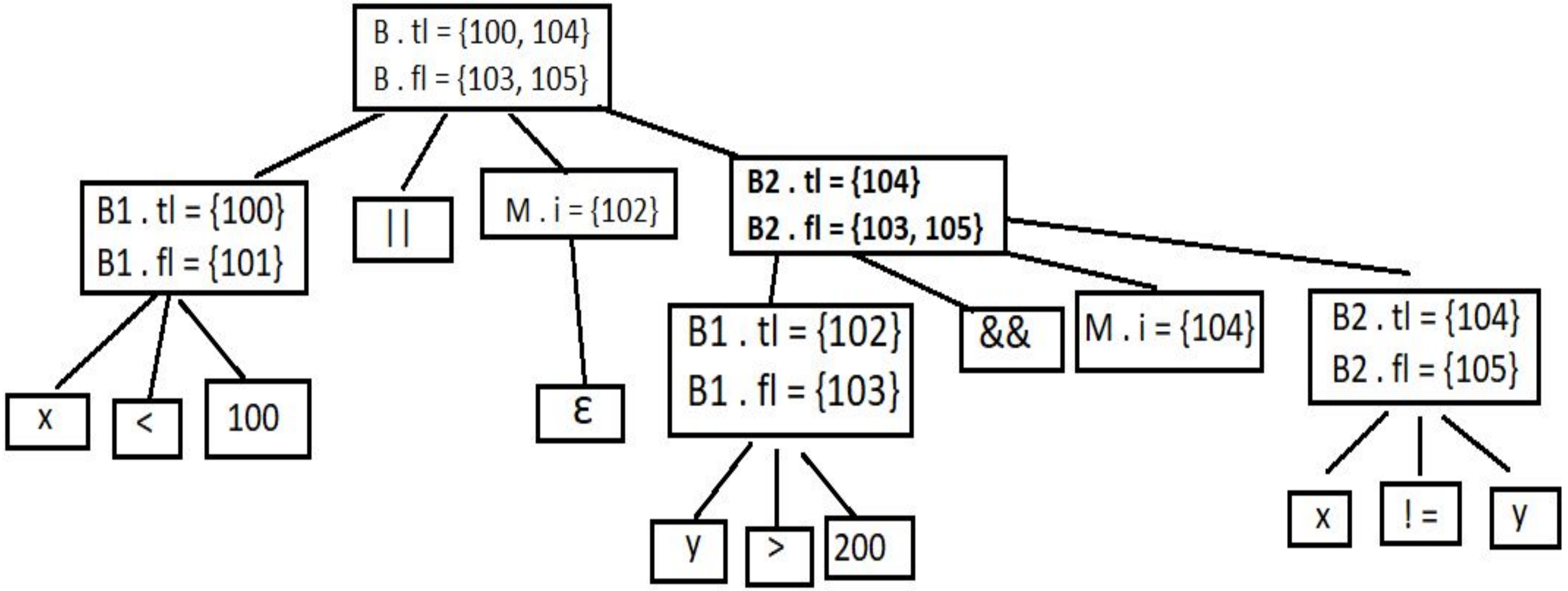
Translation Rules:

```
1. B -> B1 || MB2
{
     Backpatch (B1.fl, M.instr);
     B . tl = merge (B1 . tl, B2 . tl );
     B . fl = B2 . Fl;
}
```

```
2. B -> B1 && MB2
{
     Backpatch (B1 . tl, M . instr);
     B . tl = B2 . Tl;
     B . fl = merge (B1 . fl, B2 . fl);
}
```

```
3. B -> ! B1
{
      B . tl = B1 . fl;
      B . fl = B1 . tl;
}
```


B
B1 . tl = {100}
B1 . fl = {101}
||
M . i = {102}
B2
x
<
100
ε
B1 . tl = {102}
B1 . fl = {103}
&&
M . i = {104}
B2 . tl = {104}
B2 . fl = {105}
y
>
200
x
! =
y


B . tl = {100, 104}
B . fl = {103, 105}
B1 . tl = {100}
B1 . fl = {101}
||
M . i = {102}
B2 . tl = {104}
B2 . fl = {103, 105}
x
<
100
ε
B1 . tl = {102}
B1 . fl = {103}
&&
M . i = {104}
B2 . tl = {104}
B2 . fl = {105}
y
>
200
x
! =
y

Labels and Goto:

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

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

One-pass code generation using backpatching:

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

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

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

Backpatching for Boolean Expressions:

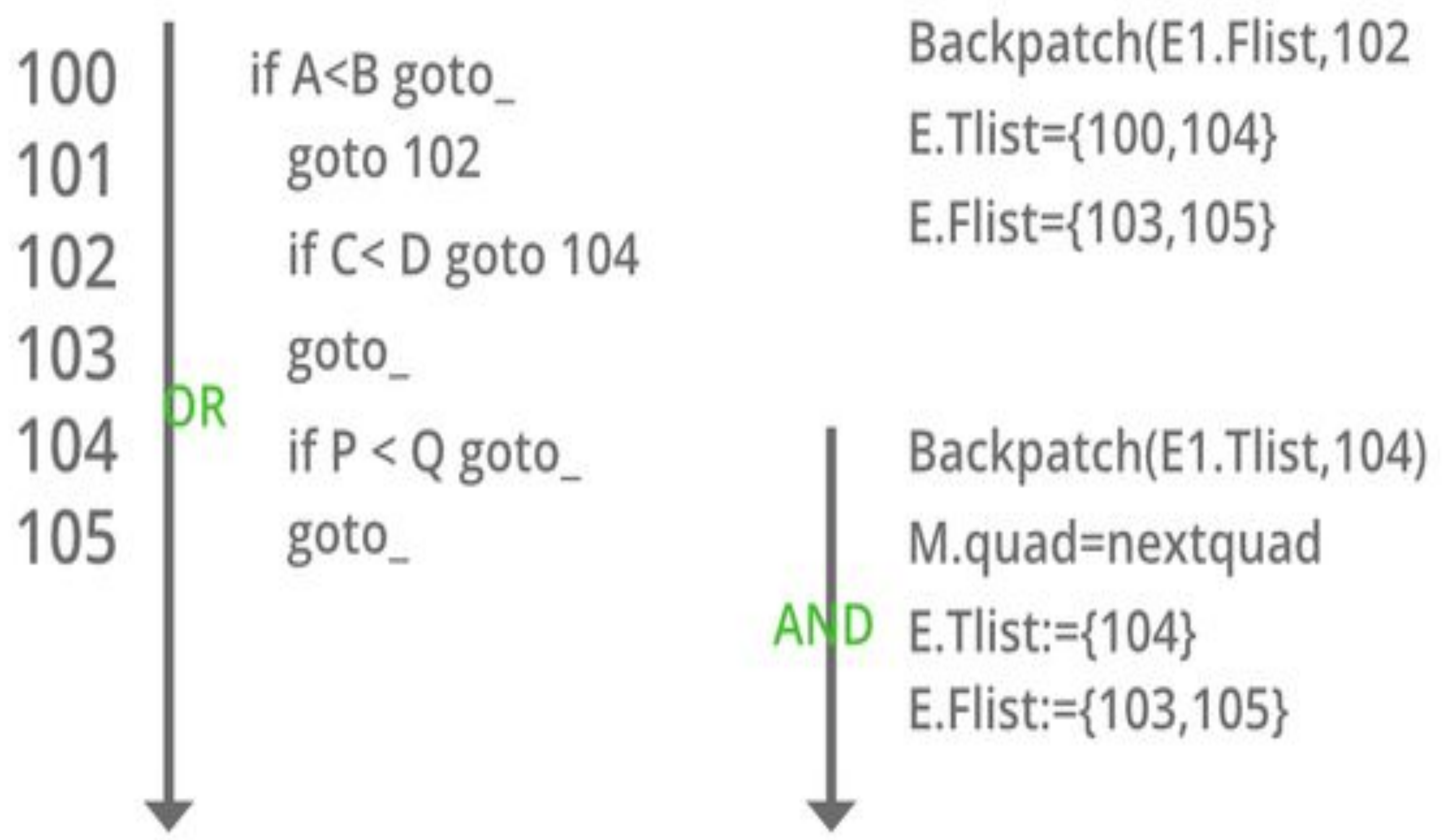
- Using a translation technique, it can create code for Boolean expressions during bottom-up parsing. In grammar, a non-terminal marker M creates a semantic action that picks up the index of the next instruction to be created at the proper time.

For Example, Backpatching using boolean expressions production rules table:

- **Step 1:** Generation of the production table

- **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



```
100   if A<B goto_
101     goto 102
102     if C< D goto 104
103     goto_
104     if P < Q goto_
105     goto_
```

Backpatch(E1.Flist,102

E.Tlist={100,104}

E.Flist={103,105}

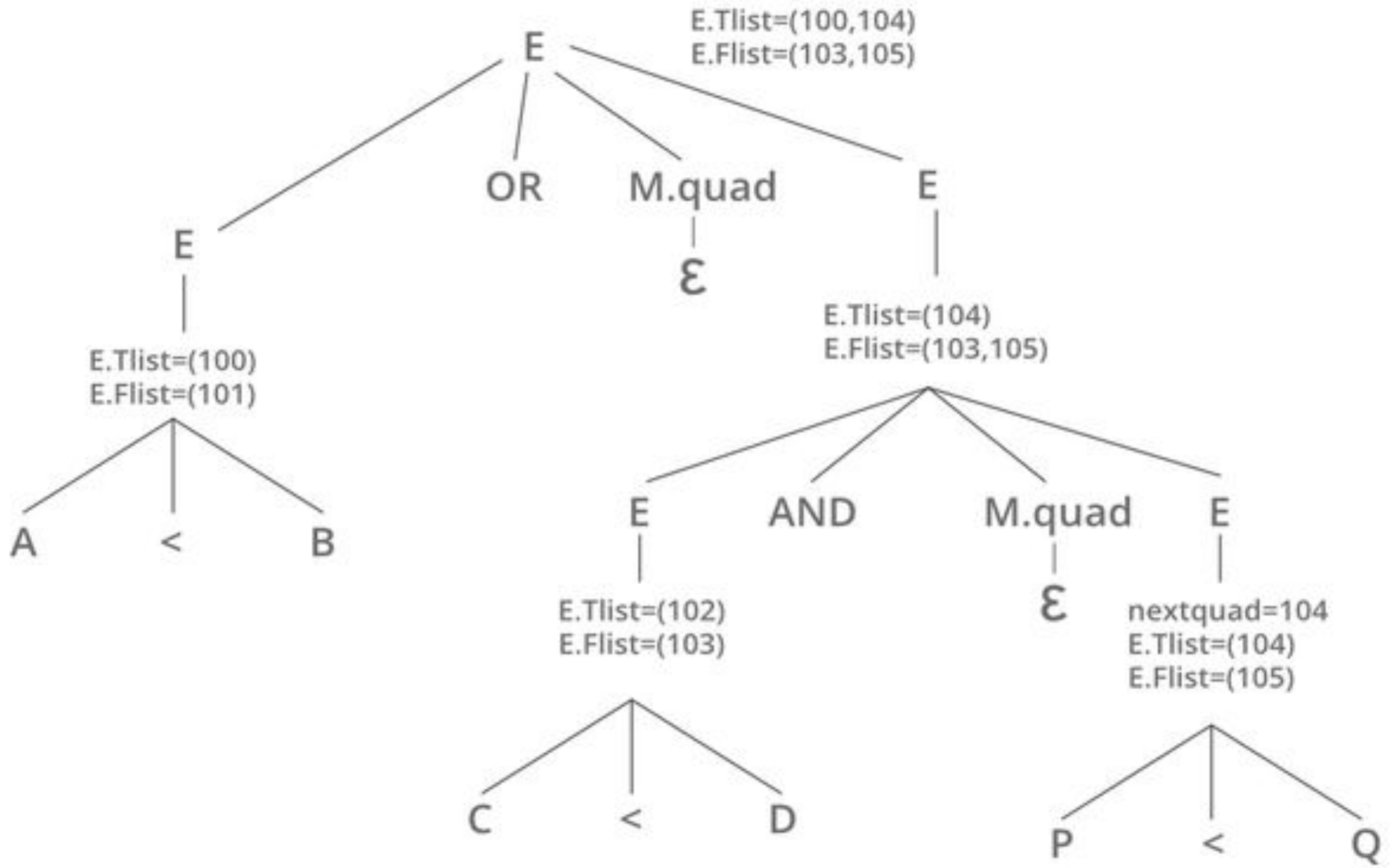
Backpatch(E1.Tlist,104)

M.quad=nextquad

E.Tlist:={104}

E.Flist:={103,105}

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



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

- **Alter the flow of control:** Boolean expressions are conditional expressions that change the flow of control in a statement. The value of such a Boolean statement is implicit in the program's position. For example, if (A) B, the expression A must be true if statement B is reached.

- **Compute logical values:** During bottom-up parsing, it may generate code for Boolean statements via a translation mechanism. A non-terminal marker M in the grammar establishes a semantic action that takes the index of the following instruction to be formed at the appropriate moment.

Applications of Backpatching:

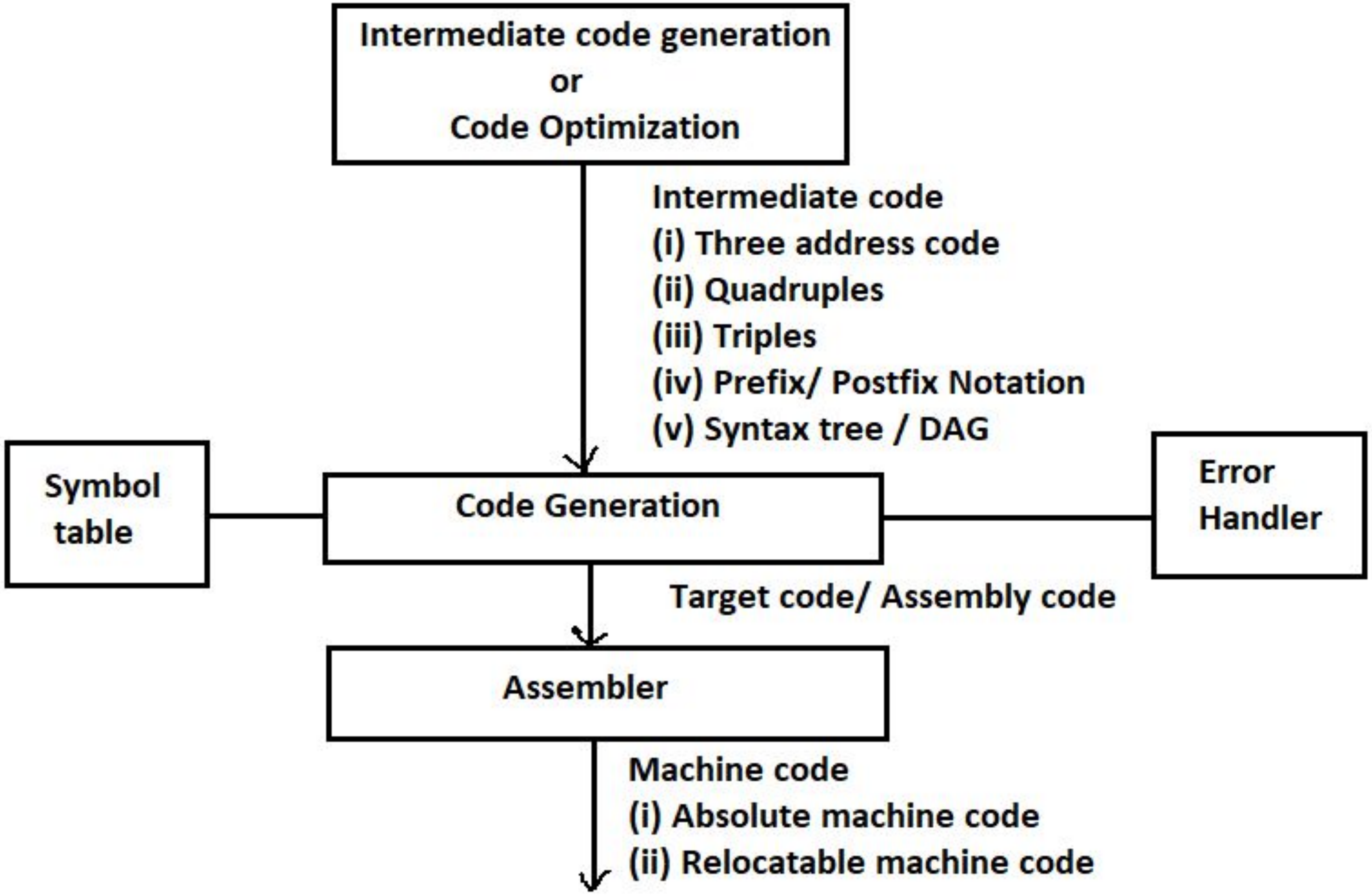
- Backpatching is used to translate flow-of-control statements in one pass itself.
- Backpatching is used for producing quadruples for boolean expressions during bottom-up parsing.
- It is the activity of filling up unspecified information of labels during the code generation process.
- It helps to resolve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

Code Generation

Code generation is the final activity of compiler code generation, is the process of creating assembly/ machine language.

There are some properties of code generation:

- Correctness: purpose of the code should not be altered
- High Quality: no error, no ambiguous statement
- Efficient use of resource of target machine: code generation phase directly interacts with target machine, so efficiently use the components of target machine
- Quick code generation: quickly generate code after the compilation process is over.


Intermediate code generation
or
Code Optimization
Intermediate code
(i) Three address code
(ii) Quadruples
(iii) Triples
(iv) Prefix/ Postfix Notation
(v) Syntax tree / DAG
Symbol table
Code Generation
Error Handler
Target code/ Assembly code
Assembler
Machine code
(i) Absolute machine code
(ii) Relocatable machine code

Issues in code generation

In the code generation phase, various issues can arises:

1. Input to the code generator
2. Target program
3. Memory management
4. Instruction selection
5. Register allocation
6. Evaluation order

1. Input to the code generator

- The input to the code generator contains the intermediate representation of the source program and the information of the symbol table. The source program is produced by the front end.
- Intermediate representation has the several choices:
  a) Postfix notation
  b) Syntax tree
  c) Three address code
- We assume front end produces low-level intermediate representation i.e. values of names in it can directly manipulated by the machine instructions.
- The code generation phase needs complete error-free intermediate code as an input requires.

2. Target program:

The target program is the output of the code generator. The output can be:

a) **Assembly language:** It allows subprogram to be separately compiled.

b) **Relocatable machine language:** It makes the process of code generation easier.

c) **Absolute machine language:** It can be placed in a fixed location in memory and can be executed immediately.

3. Memory management

- During code generation process the symbol table entries have to be mapped to actual p addresses and levels have to be mapped to instruction address.
- Mapping name in the source program to address of data is co-operating done by the front end and code generator.
- Local variables are stack allocation in the activation record while global variables are in static area.

4. Instruction selection:

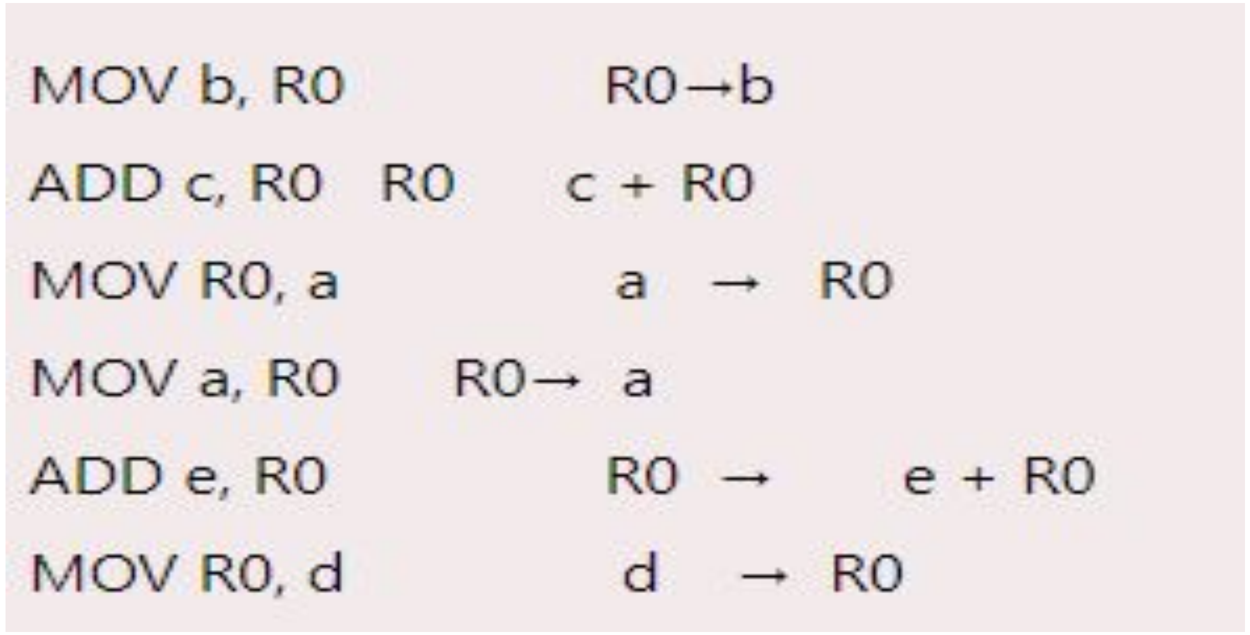
- Nature of instruction set of the target machine should be complete and uniform.
- When you consider the efficiency of target machine then the instruction speed and machine idioms are important factors.
- The quality of the generated code can be determined by its speed and size.

Example: The Three address code is:

a:= b + c

d:= a + e

Inefficient assembly code is:



```
MOV b, R0              R0→b
ADD c, R0   R0      c + R0
MOV R0, a              a  →  R0
MOV a, R0      R0→  a
ADD e, R0              R0  →      e + R0
MOV R0, d              d   →  R0
```

5. Register allocation

- Register can be accessed faster than memory. The instructions involving operands in register are shorter and faster than those involving in memory operand.
- The following sub problems arise when we use registers:
- **Register allocation:** In register allocation, we select the set of variables that will reside in register.
- **Register assignment:** In Register assignment, we pick the register that contains variable.
- Certain machine requires even-odd pairs of registers for some operands and result.

For example: Consider the following division instruction of the form:

- D x, y ; **x** is the dividend even register in even/odd register pair

  y is the divisor

  Even register is used to hold the reminder.

- **Old register** is used to hold the quotient.

6. Evaluation order

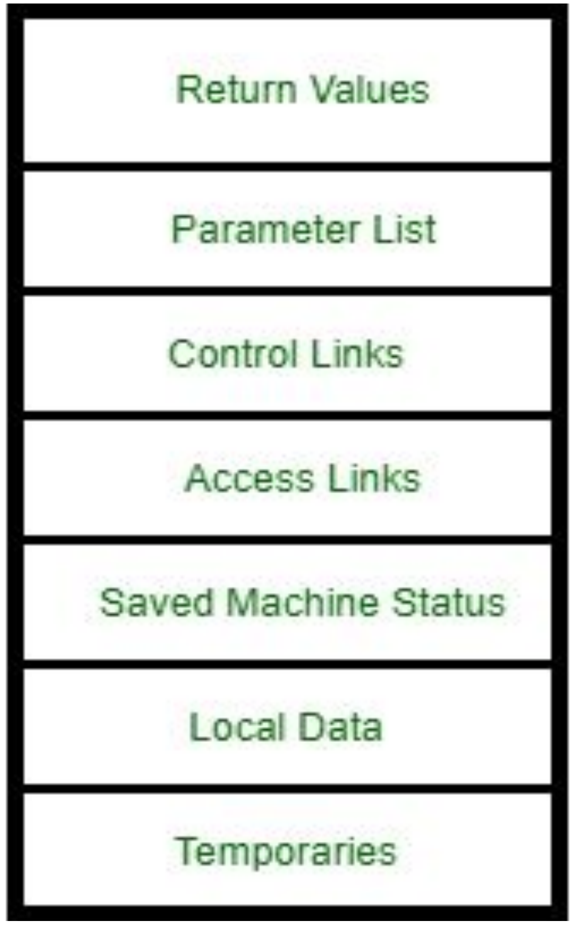
- The efficiency of the target code can be affected by the order in which the computations are performed. Some computation orders need fewer registers to hold results of intermediate than others.

Activation Records

- An activation record is a contiguous block of storage that manages information required by a single execution of a procedure.
- When you enter a procedure, you allocate an activation record, and when you exit that procedure, you de-allocate it.
- Basically, it stores the status of the current activation function. So, whenever a function call occurs, then a new activation record is created and it will be pushed onto the top of the stack.
- It will be in function till the execution of that function. So, once the procedure is completed and it is returned to the calling function, this activation function will be popped out of the stack.
- If a procedure is called, an activation record is pushed into the stack, and it is popped when the control returns to the calling function.

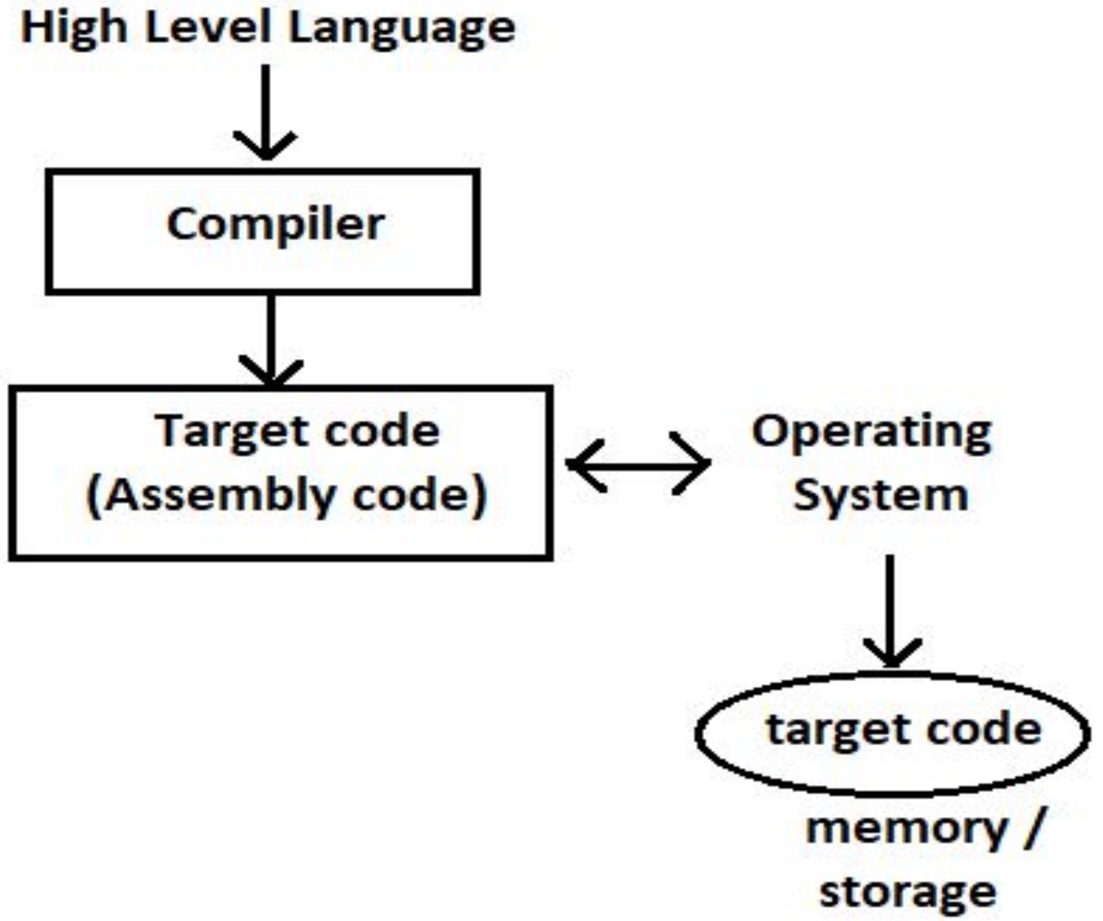
- Activation Record includes some fields which are – Return values, parameter list, control links, access links, saved machine status, local data, and temporaries.

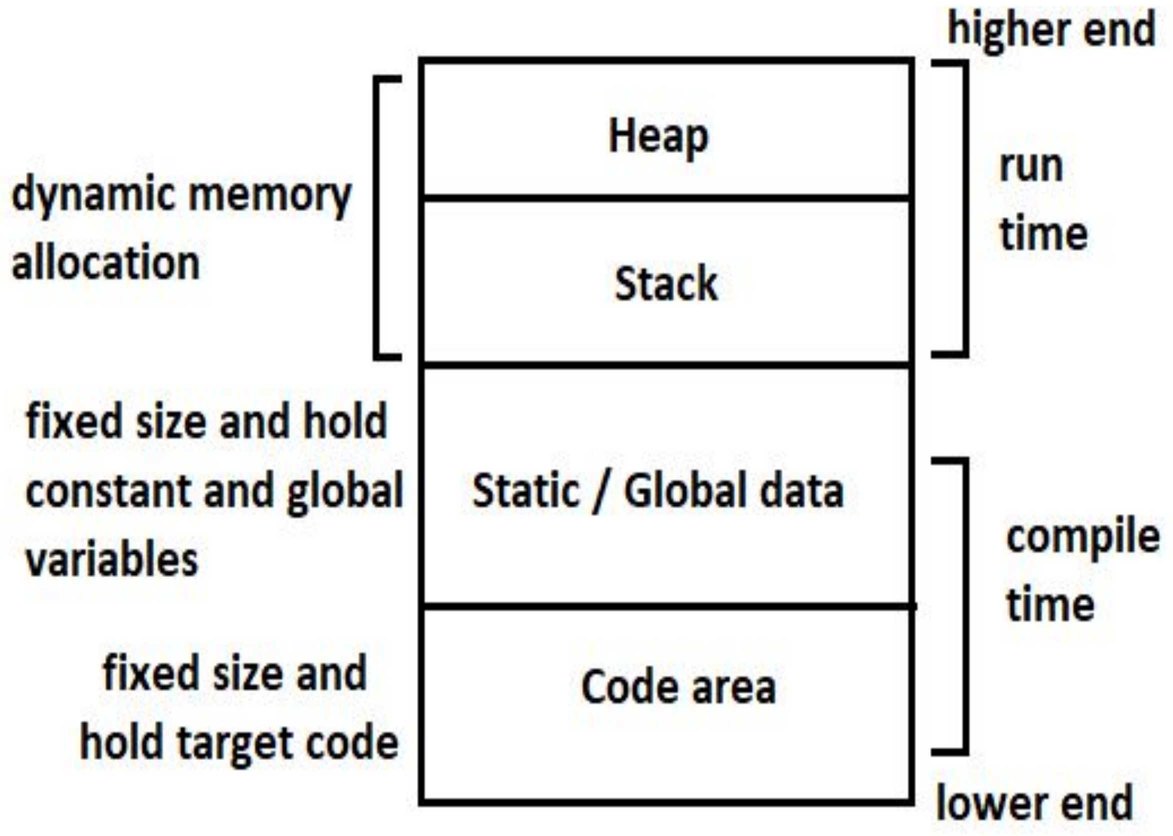
- **Temporaries:** The temporary values, such as those arising in the evaluation of expressions, are stored in the field for temporaries.
- **Local data:** The field for local data holds data that is local to an execution of a procedure.
- **Saved Machine States:** The field for Saved Machine Status holds information about the state of the machine just before the procedure is called. This information includes the value of the program counter and machine registers that have to be restored when control returns from the procedure.



- **Access Link/ Static Link :** It refers to information stored in other activation records that is non-local. The access link is a static link and the main purpose of the access link is to access the data which is not present in the local scope of the activation record. It is a static link.
- **Control Links/ Dynamic Links :** In this case, it refers to an activation record of the caller. They are generally used for links and saved status. It is a dynamic link in nature. When a function calls another function, then the control link points to the activation record of the caller.
  Record A contains a control link pointing to the previous record on the stack. Dynamically executed programs are traced by the chain of control links.
- **Parameter List:** The field for parameters list is used by the calling procedure to supply parameters to the called procedure. We show space for parameters in the activation record, but in practice, parameters are often passed in machine registers for greater efficiency.
- **Return value:** The field for the return value is used by the called procedure to return a value to the calling procedure. Again in practice, this value is often returned in a register for greater efficiency.

Runtime Storage Management




higher end
Heap
Stack
run time
dynamic memory allocation
fixed size and hold constant and global variables
Static / Global data
compile time
fixed size and hold target code
Code area
lower end

Runtime memory

The run-time memory is divided into areas for:

1. **Code area:** It is known as the text part of a program that does not change at runtime. Its memory requirements are known at the compile time.
2. **Static/ global data:** In this allocation scheme, the compilation data is bound to a fixed location in the memory and it does not change when the program executes. As the memory requirement and storage locations are known in advance, runtime support package for memory allocation and de-allocation is not required.

3. Stack: Procedure calls and their activations are managed by means of stack memory allocation. It works in last-in-first-out (LIFO) method and this allocation strategy is very useful for recursive procedure calls.

4. Heap: Variables local to a procedure are allocated and de-allocated only at runtime. Heap allocation is used to dynamically allocate memory to the variables and claim it back when the variables are no more required.

Both stack and heap memory can grow and shrink dynamically and unexpectedly. Therefore, they cannot be provided with a fixed amount of memory in the system.

- The information which required during an execution of a procedure is kept in a block of storage called an activation record. The activation record includes storage for names local to the procedure.

- We can describe address in the target code using the following ways:

1. Static allocation
2. Stack allocation

- In static allocation, the position of an activation record is fixed in memory at compile time.
- In the stack allocation, for each execution of a procedure a new activation record is pushed onto the stack. When the activation ends then the record is popped.

Static allocation

1. Implementation of call statement: The following code is needed to implement static allocation:

- MOV #here + 20, callee.static_area   /*it saves return address*/</p>
- GOTO callee.code_area   /* It transfers control to the target code for the calle d procedure*/

Where,

- **callee.static_area** shows the address of the activation record.
- **callee.code_area** shows the address of the first instruction for called procedure.
- **#here + 20** literal are used to return address of the instruction following GOTO.

2. Implementation of return statement: The following code is needed to implement return from procedure callee:

```
GOTO * callee.static_area
```

It is used to transfer the control to the address that is saved at the beginning of the activation record.

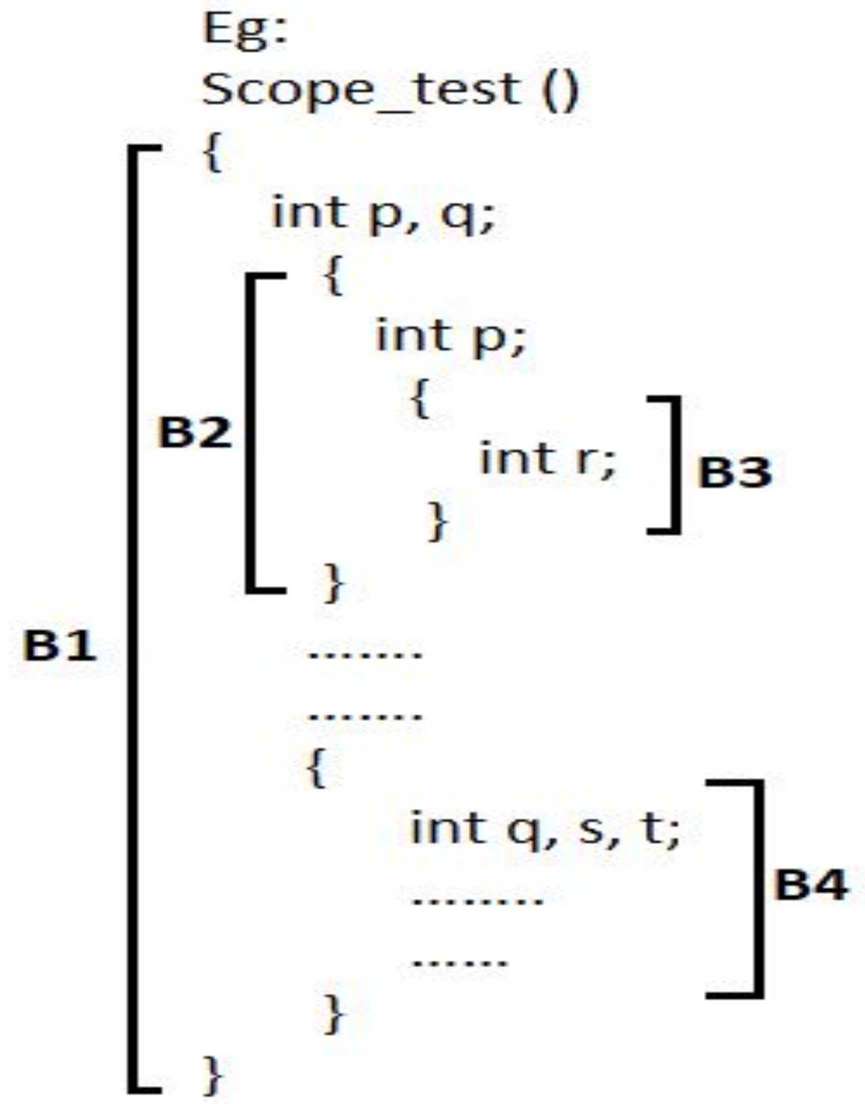
3. Implementation of action statement:

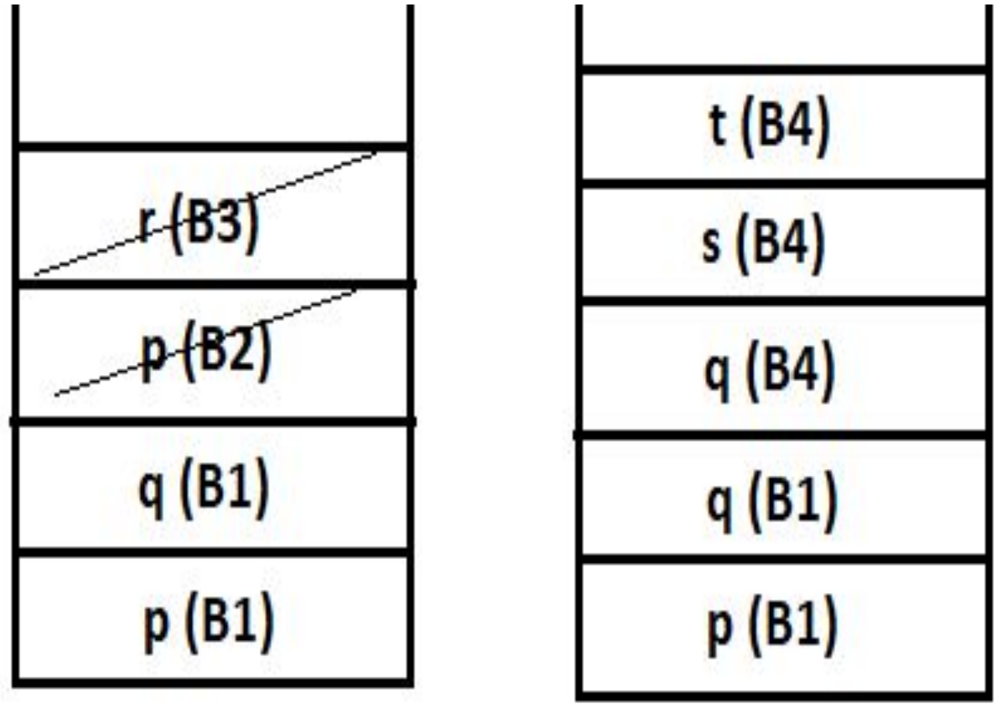
- The ACTION instruction is used to implement action statement.

4. Implementation of halt statement:

- The HALT statement is the final instruction that is used to return the control to the operating system

Static Scope Rule (lexical scope):


```
Eg:
Scope_test ()
{
   int p, q;
     {
       int p;
         {
           int r;
         }
     }
     .......
     .......
     {
        int q, s, t;
        ........
        ......
     }
}
```



Stack allocation

- Using the relative address, static allocation can become stack allocation for storage in activation records.
- In stack allocation, register is used to store the position of activation record so words in activation records can be accessed as offsets from the value in this register.
- The following code is needed to implement stack allocation:

1. Initialization of stack:

- MOV #stackstart , SP   /*initializes stack*/
- HALT   /*terminate execution*/

2. Implementation of Call statement:

- ADD #caller.recordsize, SP/* increment stack pointer */
- MOV #here + 16, *SP /*Save return address */
- GOTO callee.code_area

Where,

- **caller.recordsize** is the size of the activation record
- **#here + 16** is the address of the instruction following the **GOTO**

3. Implementation of Return statement:

- GOTO *0 (SP) /*return to the caller */
- SUB #caller.recordsize, SP /*decrement SP and restore to previous value */

Stack Allocation

Example:

```
main()
{
one()
}
one()
{
two()
}
two()
{
…………
}
```

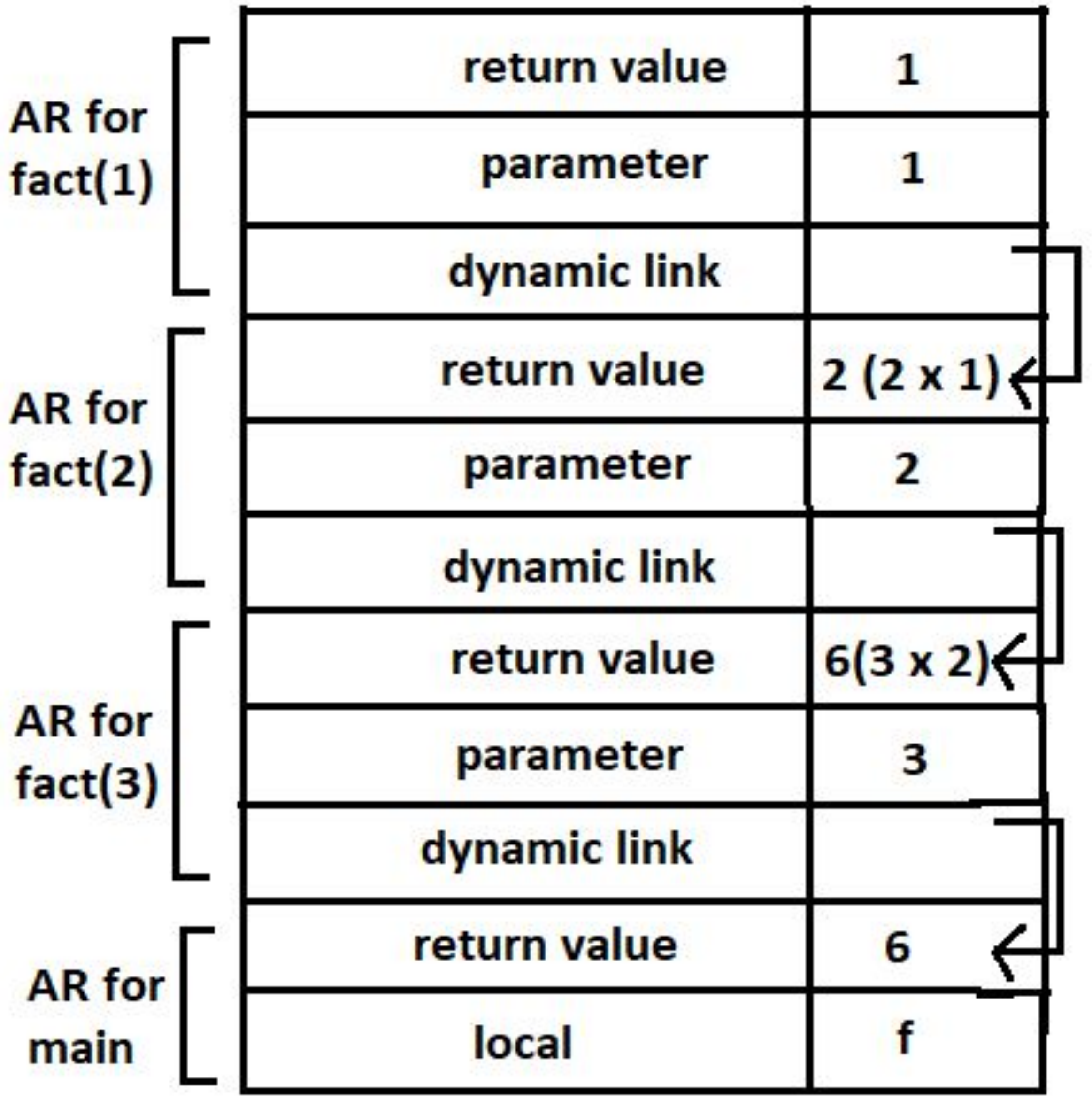
3 Activation records will be generated-

AR – main()
AR - one()
AR – two()

Example 2:

```
main()
{
int f;
f =fact(3);
}
int fact(int n)
{
if (n == 1)
   return 1;
else
   return (n * fact (n - 1))
}
```

AR – main()
AR - fact(3)
AR – fact(2)
AR – fact(1)



Code Generation Algorithm

Code Generator: Code generator is used to produce the target code for three-address statements. It uses registers to store the operands of the three address statement.

Example: Consider the three address statement x:= y + z. It can have the following sequence of codes:

```
MOV x, R0
    ADD y, R0
```

Register and Address Descriptors:

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

A code-generation algorithm: The algorithm takes a sequence of three-address statements as input. For each three address statement of the form "a:= b op c" perform the various actions. These are as follows:

- Invoke a function getreg to find out the location L where the result of computation b op c should be stored.
- Consult the address description for y to determine y'. If the value of y currently in memory and register both then prefer the register y' . If the value of y is not already in L then generate the instruction **MOV y' , L** to place a copy of y in L.
- Generate the instruction **OP z' , L** where z' is used to show the current location of z. if z is in both then prefer a register to a memory location. Update the address descriptor of x to indicate that x is in location L. If x is in L then update its descriptor and remove x from all other descriptor.
- If the current value of y or z have no next uses or not live on exit from the block or in register then alter the register descriptor to indicate that after execution of x : = y op z those register will no longer contain y or z.

Generating Code for Assignment Statements:

- The assignment statement d:= (a-b) + (a-c) + (a-c) can be translated into the following sequence of three address code:

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

Code sequence for the example is as follows:

Statement	Code Generated	Register descriptor Register empty	Address descriptor
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 R1
d:= v + u	ADD R1, R0 MOV R0, d	R0 contains d	d in R0 d in R0 and memory

Cross Compiler

Bootstrapping:

Bootstrapping is a process to create new compiler. It is also used to create cross compiler.

Cross compiler:

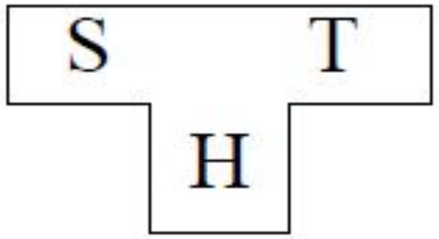
- A cross compiler is a compiler capable of creating executable code for a platform other than the one on which the compiler is running.
- Compiler that runs on one machine and produces target code for another machine.
- For example, a cross compiler executes on machine X and produces machine code for machine Y.

Where is the cross compiler used?

- In bootstrapping, a cross-compiler is used for transitioning to a new platform. When developing software for a new platform, a cross-compiler is used to compile necessary tools such as the operating system and a native compiler.
- For microcontrollers, we use cross compiler because it doesn’t support an operating system.
- It is useful for embedded computers which are with limited computing resources.
- To compile for a platform where it is not practical to do the compiling, a cross-compiler is used.
- When direct compilation on the target platform is not infeasible, so we can use the cross compiler.
- It helps to keep the target environment separate from the built environment.

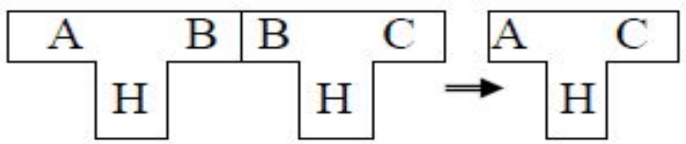
T diagram

- The conversion from source language (S) to target language (T) using compiler which I written in language (H) is presented as



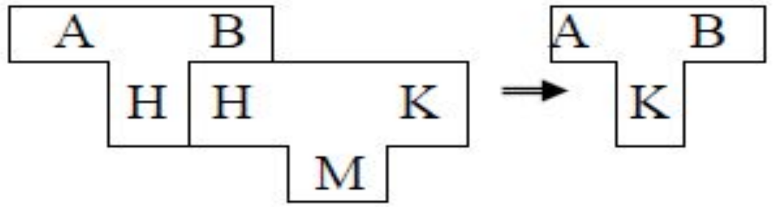
- T-Diagram can be combined in two basic ways.

The First T-diagram Combination



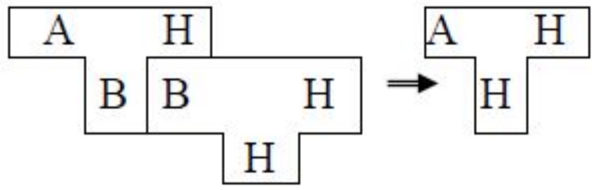
- Two compilers run on the same machine H
  - First from A to B
  - Second from B to C
  - Result from A to C on H

The Second T-diagram Combination



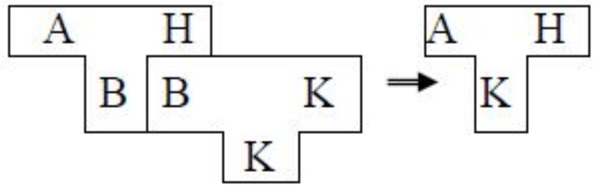
- Translate implementation language of a compiler from H to K
- Use another compiler from H to K

The First Scenario



- Translate a compiler from A to H written in B
  - Use an existing compiler for language B on machine H

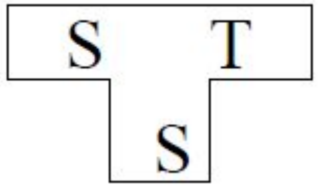
The Second Scenario



- Use an existing compiler for language B on different machine K
  - Result in a cross compiler

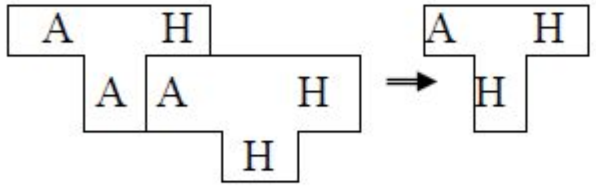
Process of Bootstrapping

- Write a compiler in the same language



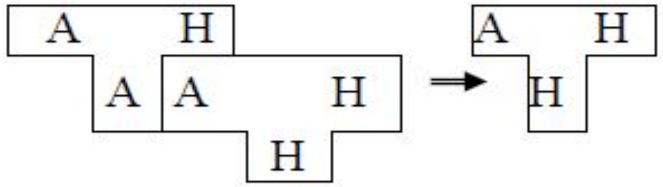
- No compiler for source language yet
- Porting to a new host machine

The First step in bootstrap



- "quick and dirty" compiler written in machine language H
- Compiler written in its own language A
- Result in running but inefficient compiler

The Second step in bootstrap



- Running but inefficient compiler
- Compiler written in its own language A
- Result in final version of the compiler