Syntax Directed Definition (SDD)

Syntax Directed Definition is a kind of abstract specification, a combination of CFG + semantic rules

It is generalization of context free grammar in which each grammar production **X –> a** is associated with it a set of production rules of the form $s = f(b_1, b_2, \ldots\ldots b_k)$ where s is the attribute obtained from function f.

- Attributes are associated with grammar symbols.
- Semantic rules are associated with productions.
- If "X" is a symbol and "a" is one of its attribute then Xa denotes value at node X

s

Example:-

Productions	Semantic Rules
E→E+T	E.Val = E.val + T.val
E→T	E.Val = T.val

Syntax Directed Definition (SDD)

SDD is used to add High Level information to the grammar by adding semantic rules.

SDD gives information like

- How symbols get values
- Simplifying evaluation
- Deriving attributes
- Which production to be evaluated first.

Types of Attributes:-

1. **Synthesized Attribute**, if a node takes value from its children

    Ex:- A→BCD // A is parent & BCD are children and s is attribute

    A.s = B.s // Here parent A is taking value from its child B

2. **Inherited Attribute**, if a node takes value from its parent or siblings

    Ex:- A→BCD // A is parent & BCD are children and s is attribute

    C.s = A.s // Here C is taking value from its parent A

    C.s = B.s // Here C is taking value from its sibling B

Syntax Directed Definition (SDD)

Types of Syntax Directed Definition :

1. S-Attributed SDD or S-Attributed Definition Or S-Attributed Grammar:-

- A SDD that uses only synthesized attributes

  Ex:- A→BCD

  A.s = B.s
- The semantic actions are placed at right end of production that's why called POSTFIX SDD
- Attributes are evaluated with Bottom-Up parsing.

2. L-Attributed SDD or L-Attributed Definition Or L-Attributed Grammar:-

- A SDD that uses both synthesized & inherited attributes but each inherited attribute is restricted to inherit from parent or left siblings only

  Ex:- A→BCD

  A.s =B.s

  C.s = A.s

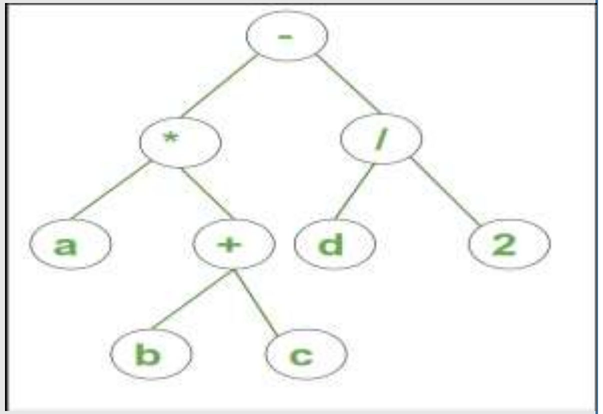
  C.s = B.s

  C.s = D.s // WRONG because its inherited form right sibling.
- The semantic actions are placed at any end of production
- Attributes are evaluated with depth first , left to right order.

Syntax Tree

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

Example:- **a * (b + c) – d /2**



<u>Constructing a Syntax Tree</u>

3 functions are used to construct a syntax tree

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

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

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

Syntax Tree-example

Example:- Construct syntax tree for the given expression X*Y-5+2

STEP1:-- convert the expression from infix to postfix XY*5-Z+

STEP2:-- construct symbol-operation table

Symbol	Operation
X	P1=mkleaf(id , ptr to entry X)
Y	P2= =mkleaf(id , ptr to entry Y)
*	P3=mknode(*, p1,p2)
5	P4=mkleaf (num,5)
-	P5=mknode(-,p3,p4)
Z	P6=mkleaf(id , ptr to entry Z)
+	P7=mknode(+,p5,p6)

Syntax Tree-example

STEP3:-- write the grammar , for the expression X*Y-5+Z ,here operations performed are *, -, +

E→ E1 * T

E→ E1 - T

E→ E1 + T

E→ T

T→ id

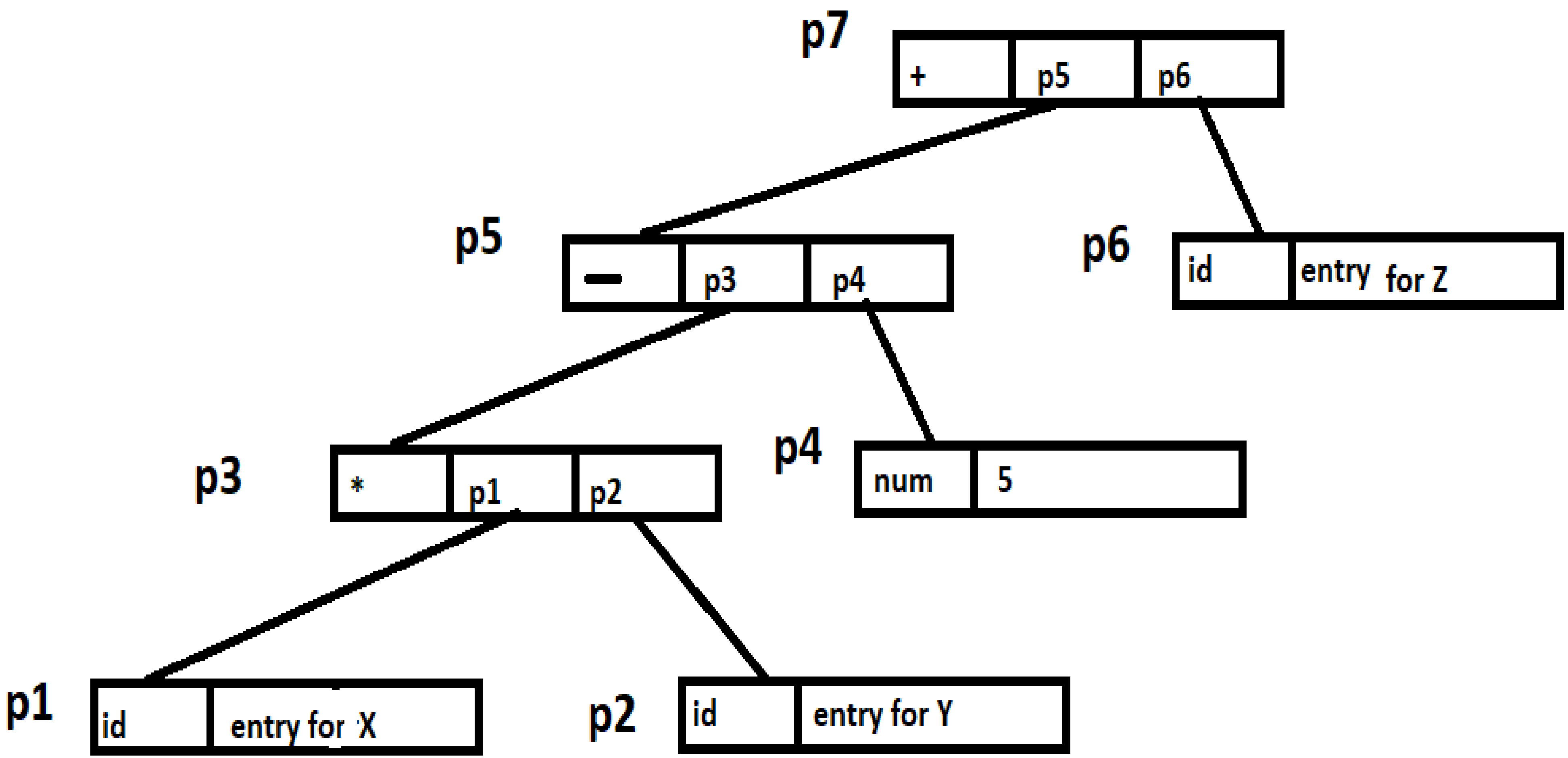
T→ num

STEP3:-- write the SDD for the grammar = CFG + semantic rules (refer operation for a symbol form symbol-operation table)

Production	Semantic Operation
E→ E1 * T	E.node = mknode(*, E1.node, T.node)
E→ E1 - T	E.node = mknode(-, E1.node, T.node)
E→ E1 + T	E.node = mknode(+, E1.node, T.node)
E→ T	E.node = T.node
T →id	T.node =mkleaf(id, id.ptr_entry)
T→ num	T.node =mkleaf(num, num.value)

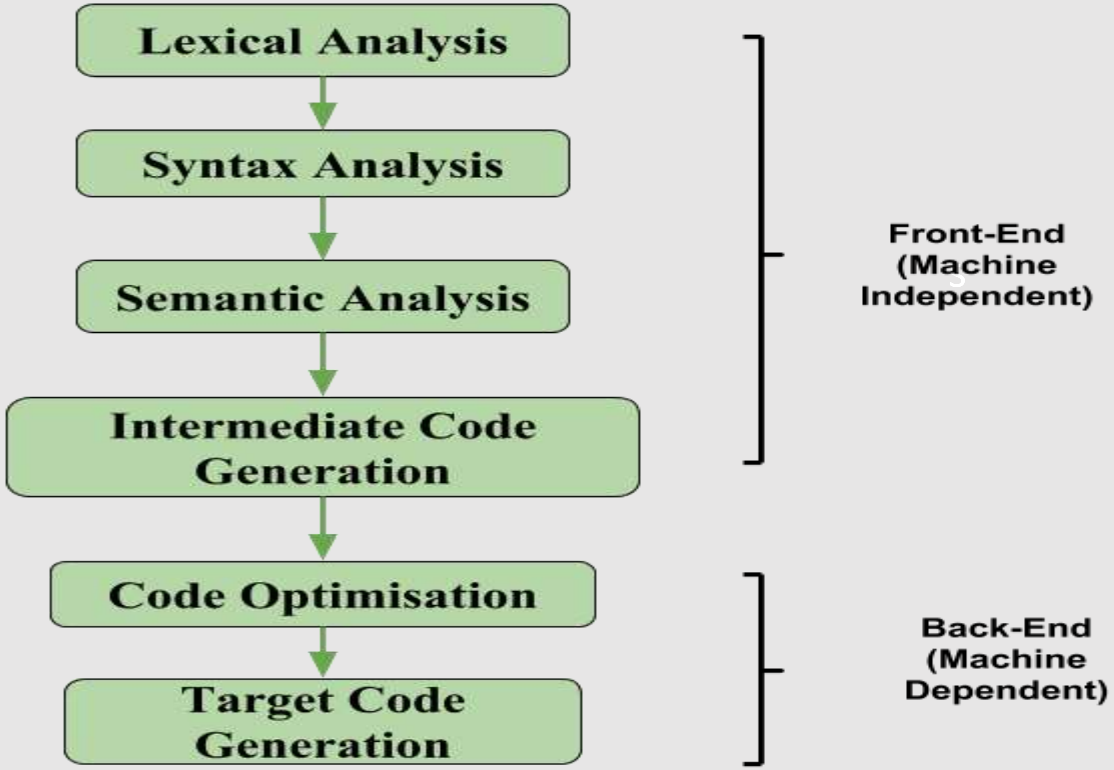
Syntax Tree-example

STEP5:-- construct syntax tree, follow the order of operations as in symbol-operation table



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:



NOTE:-

Intermediate code can be either I

i) language-specific (e.g., Bytecode for Java) or

ii) language independent (three-address code).

Intermediate code Representations

1. Postfix Notation: Also known as reverse Polish notation or suffix notation.

Ex:-For expression (a + b) * c postfix is : ab + c *

For expression(a – b) * (c + d) + (a – b) postfix is :   ab – cd + *ab -+

2. 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.

Three address statement is of form x = y op z, where x, y, and z will have address (memory location).

Ex:- The three address code for the expression a + b * c + d :

T 1 = b * c

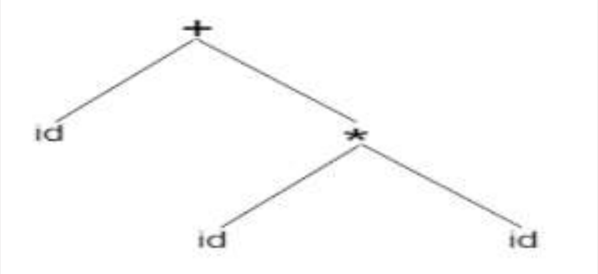
T 2 = a + T 1

T 3 = T 2 + d

T 1 , T 2 , T 3 are temporary variables.

3. Syntax Tree: A syntax tree is a tree in which each leaf node represents an operand, while each inside node represents an operator.

Example: id + id * id



Intermediate code Representations

Advantages of Intermediate Code Generation:

1. **Easier to implement:** Intermediate code generation can simplify the code generation process by reducing the complexity of the input code, making it easier to implement.
2. **Facilitates code optimization:** Intermediate code generation can enable the use of various code optimization techniques, leading to improved performance and efficiency of the generated code.
3. **Platform independence:** Intermediate code is platform-independent, meaning that it can be translated into machine code or bytecode for any platform.
4. **Code reuse:** Intermediate code can be reused in the future to generate code for other platforms or languages.
5. **Easier debugging:** Intermediate code can be easier to debug than machine code or bytecode, as it is closer to the original source code.

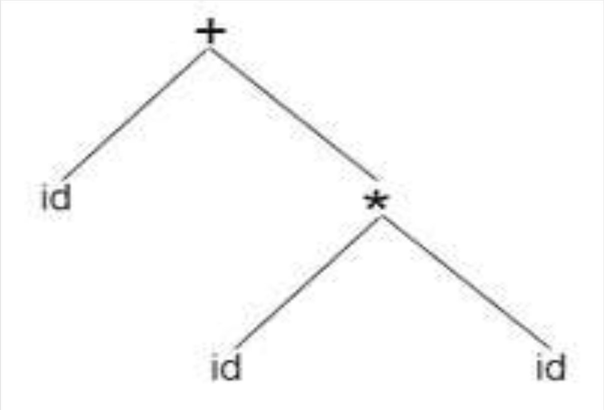
Abstract Syntax Tree(AST)

Abstract Syntax Tree (AST) is a kind of tree representation of the abstract syntactic structure of source code written in a specific programming language.

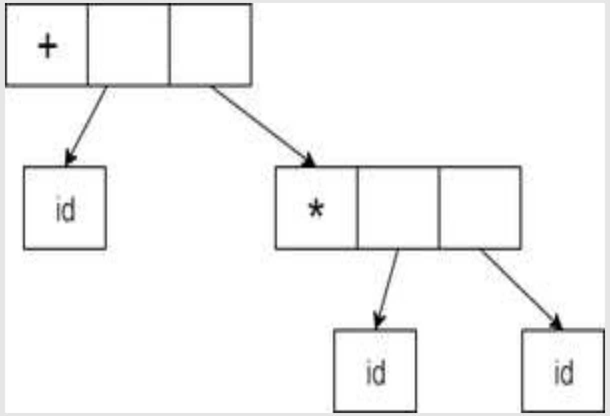
- An AST is essentially a simplified version of a parse tree.
- Each node of the tree denotes a construct occurring in the source code.
- AST is highly specific to programming languages
- Abstract Syntax Tree (AST) is used because some constructs cannot be represented in context-free grammar, such as implicit typing.

Ex:- ***d + id * id** would have the following syntax tree & AST :*

Syntax Tree



Abstract Syntax Tree



Three-Address Code

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.
Three address statement is of form x = y op z, where x, y, and z will have address (memory location).

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.

NOTE:- Three-address code is an intermediate code. It is used by the optimizing compilers.

There are 3 ways to represent a Three-Address Code in compiler design:

i) Quadruples
ii) Triples
iii) Indirect Triples

Three-Address Code

i) Quadruples

The quadruples have four fields to implement the three address code. The field of quadruples contains the name of the operator, the first source operand, the second source operand and the result respectively.

Ex:- For the expression a := -b * c + d

The 3 –address code will be

t1 := -b

t2 := c + d

t3 := t1 * t2

a := t3

Quadruple representation is

	Operator	Source 1	Source 2	Destination
(0)	uminus	b	-	t_1
(1)	+	c	d	t_2
(2)	*	t_1	t_2	t_3
(3)	:=	t_3	-	a

Three-Address Code

ii) **Triples**

The triples have three fields to implement the three address code. The field of triples contains the name of the operator, the first source operand and the second source operand.

Ex:- For the expression a := -b * c + d

The 3 –address code will be

t1 := -b

t2 := c + d

t3 := t1 * t2

a := t3

Triple representation

	Operator	Source 1	Source 2
(0)	uminus	b	-
(1)	+	c	d
(2)	*	(0)	(1)
(3)	:=	(2)	-

Three-Address Code

iii) Indirect Triples

Similar to triple representation , but this representation makes use of pointer to the listing of all references to computations which is made separately and stored.

Ex:- For the expression a := -b * c + d

The 3 –address code will be

t1 := -b

t2 := c + d

t3 := t1 * t2

a := t3

Indirect Triple representation

	Operator	Source 1	Source 2
(0)	uminus	b	-
(1)	+	c	d
(2)	*	(0)	(1)
(3)	:=	(2)	-

A1	(0)
A2	(1)
A3	(2)
A4	(3)

SDD vs SDT

A **syntax-directed definition** (SDD) associates a *semantic rule* with each grammar production, the rule states how attributes are calculated

- conceptually, (and for SDTs too) each node may have multiple attributes
  - perhaps a struct/record/dictionary is used to group many attributes
- attributes may be concerned e.g. with data type, numeric value, symbol identification, code fragment, memory address, machine register choice

A **syntax-directed translation scheme** (SDT) uses *semantic actions* embedded anywhere in the bodies (right hand sides) of productions

- actions may perform arbitrary computations, such as appending output
- they are carried out in left-to-right order during parsing.

Syntax Directed Translation into 3 address Code

Syntax-directed translation rules can be defined to generate the three address code while parsing the input. It may be required to generate temporary names for interior nodes which are assigned to non-terminal E on the left side of the production $E \rightarrow E_1$ op E_2. we associate two attributes place and code associated with each non-terminal.

E.place, the name that will hold the value of E.

E.code, the sequence of three-address statements evaluating E.

Syntax Directed Translation into 3 address Code

To generate intermediate code for SDD, first searching is applied to get the information of the identifier from the symbol table. After searching, the three address code is generated for the program statement. Function lookup will search the symbol table for the lexeme and store it in id.place.

Function **newtemp** is defined to return a new temporary variable when invoked and

gen function generates the three address statement in one of the above standard forms depending on the arguments passed to it.

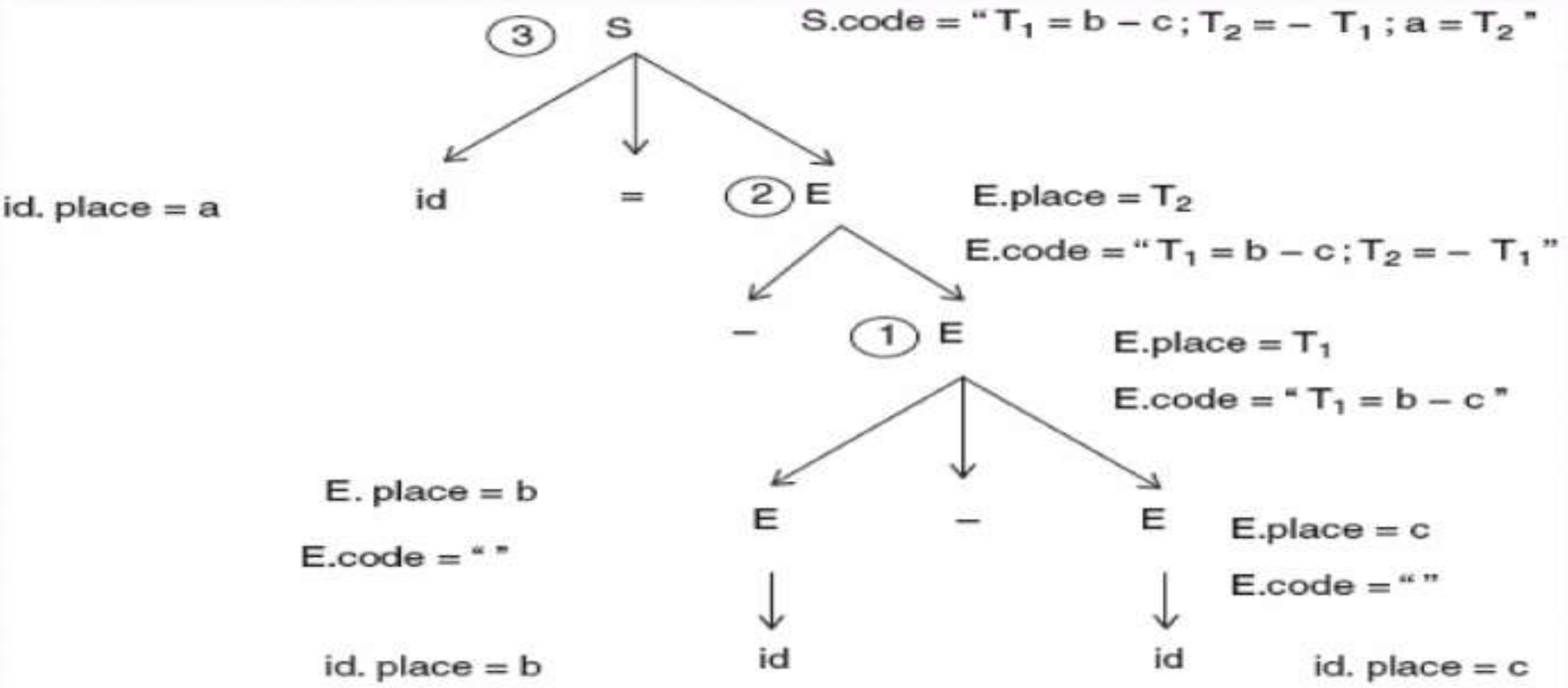
Syntax Directed Translation into 3 address Code

Three-address code for expressions

Production	Semantic rule
$S \rightarrow id = E$	{id.place = lookup(id.name); if id.place ≠ null then S.code = E.code | | gen(id.place ":=" E.place) else S.code = type_error}
$E \rightarrow - E_1$	{E.place = newtemp(); E.code = E_1.code | | gen(E.place ":=" "–" E_1.place)}
$E \rightarrow E_1 + E_2$	{E.place = newtemp(); E.code = E_1.code | | E_2.code | | gen(E.place ":=" E_1.place "+" E_2.place)}
$E \rightarrow E_1 - E_2$	{E.place = newtemp(); E.code = E_1.code | | E_2.code | | gen(E.place ":=" E_1.place "–" E_2.place)}
$E \rightarrow E_1 * E_2$	{E.place = newtemp(); E.code = E_1.code | | E_2.code | | gen(E.place ":=" E_1.place "*" E_2.place)}
$E \rightarrow E_1 / E_2$	{E.place = newtemp(); E.code = E_1.code | | E_2.code | | gen(E.place ":=" E_1.place "/" E_2.place)}
$E \rightarrow id$	{E.place = lookup(id.name), E.code = " "}

Syntax Directed Translation into 3 address Code: Example

Example :- Syntax tree for **a** = –(**b** – **c**)



Translation of Boolean Expression

Boolean Expression have 2 primary purpose.

1. Used to computing logical values.
2. Used to computing conditional expression using if then else or while-do.

$$E \rightarrow E_1 \text{ or } E_2$$
$$E \rightarrow E_1 \text{ and } E_2$$
$$E \rightarrow \text{not } E_1$$
$$E \rightarrow id_1 \text{ relop } id_2$$
$$E \rightarrow (E_1)$$
$$E \rightarrow \text{true}$$
$$E \rightarrow \text{false}$$

Boolean Expression: Grammar & Action

Production	Semantic Rule
$E \rightarrow E_1$ or E_2	{E.place = newtemp(); gen(E.place "=" E_1.place "or" E_2.place)}
$E \rightarrow E_1$ and E_2	{E.place = newtemp(); gen(E.place "=" E_1.place "and" E_2.place)}
$E \rightarrow$ not E_1	{E.place = newtemp(); gen(E.place "=" "not" E_1.place)}
$E \rightarrow (E_1)$	{E.place = E_1.place}
$E \rightarrow id_1$ relop id_2	{E.place = newtemp(); gen("if" id_1.place relop.op id_2.place "goto" nextstat + 3) gen(E.place "=" "0") gen("goto" nextstat + 2) gen(E.place "=" "1")}
$E \rightarrow$ true	{E.place = newtemp(); gen(E.place "=" "1")}
$E \rightarrow$ false	{E.place = newtemp(); gen(E.place "=" "0")}

Boolean Expression: Example

Example1 :-
Given Boolean Expression
if x < y then 1 else 0

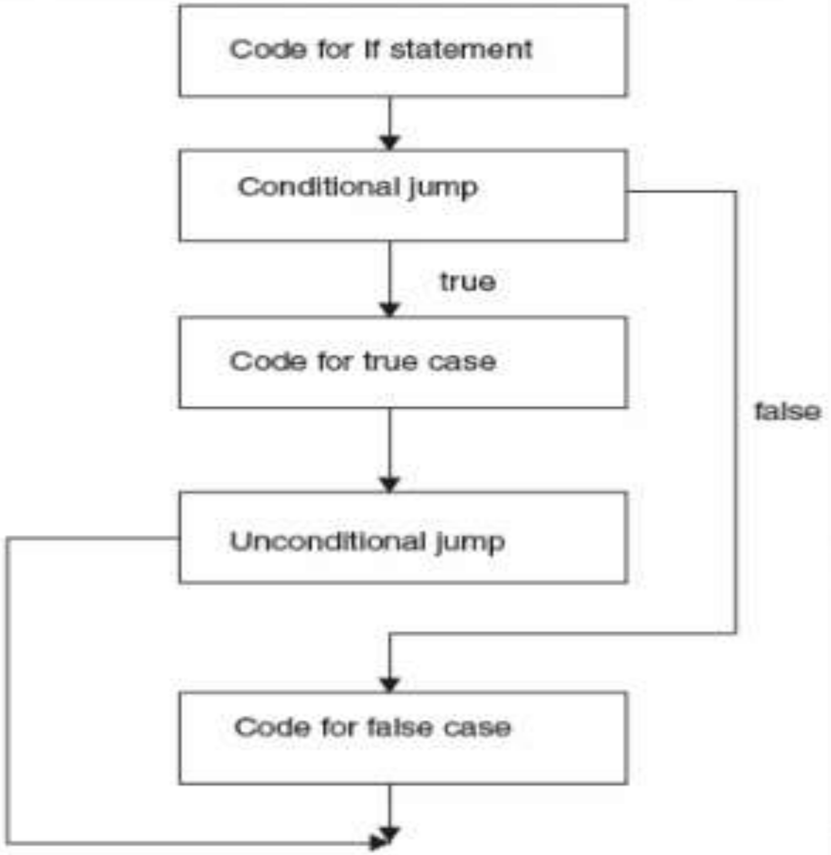
The address code for the above expression is

1. if x < y go to 3
2. $t_1 = 0$
3. go to 4
4. $t_1 = 1$

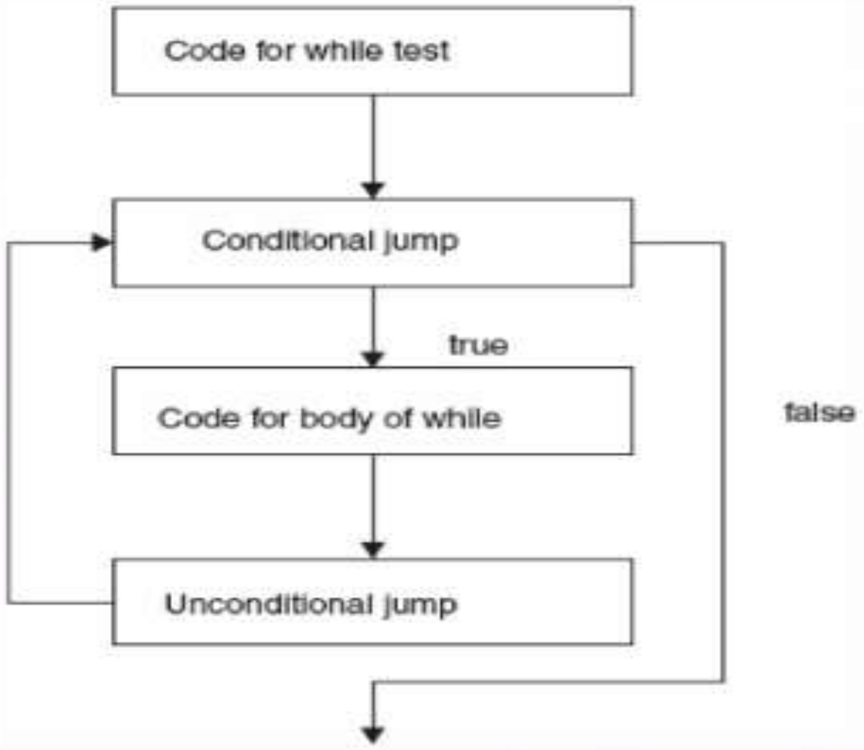
Production	Semantic Rule
$E \rightarrow E_1$ or E_2	{E.place = newtemp(); gen(E.place "=" E_1.place "or" E_2.place)}
$E \rightarrow E_1$ and E_2	{E.place = newtemp(); gen(E.place "=" E_1.place "and" E_2.place)}
$E \rightarrow$ not E_1	{E.place = newtemp(); gen(E.place "=" "not" E_1.place)}
$E \rightarrow (E_1)$	{E.place = E_1.place}
$E \rightarrow id_1$ relop id_2	{E.place = newtemp(); gen("if" id_1.place relop.op id_2.place "goto" nextstat + 3) gen(E.place "=" "0") gen("goto" nextstat + 2) gen(E.place "=" "1")}
$E \rightarrow$ true	{E.place = newtemp(); gen(E.place "=" "1")}
$E \rightarrow$ false	{E.place = newtemp(); gen(E.place "=" "0")}

Flow of Control Statements

Flow of control statements can be shown pictorially as



Control low for if-then-else statement



Control low for while statement

Translation of Control Statements

Production	Semantic Rule
$S \rightarrow$ if E then S_1	{E.true = newlabel(); E.false = S.next; S_1.next = S.next; S.code = E.code | | gen(E.true,":") | | S_1.code}
$S \rightarrow$ if E then S_1 else S_2	{E.true = newlabel(); E.false = newlabel(); S_1next = S.next; S_2.next = S.next; S.code = E.code | | gen(E.true,":") | | S_1.code | | gen(" GOTO ", S.next) | | gen(E.false, " :") | | S_2.code}
$S \rightarrow$ while E do S_1	{S.begin = newlabel(); E.true = newlabel(); E.false = S.next; S_1.next = S.next; S.code = gen(S.begin":") | | E.code | | gen(E.true,":") | | S_1.code | | gen("GOTO",S.begin)}

Flow of Control Statements

Semantic rules for three-address code for if- then statement

Production	Semantic Rules	Inference
S à if E then S1	E.true:= newlabelE.false := S.next S1.next := S.next S.code := E.code || gen (E.true':') || S1.code	We first generate a new address location E.true. If the expression E is false then control should go to the statement following the body of the if-then. Hence, we assign E.false as S.next. If the expression E is true the body of S, the statements following if-then block need to be evaluated and this is ensured by setting S1.next and S.next as same. Finally, the code corresponding to S is evaluated as E.code followed by the generation of E.true label followed by S1.code.

Flow of Control Statements

Semantic rules for three-address code for while loop

Production	Semantic Rules	Inference
S à while E do S1	S.begin := newlabelE.true := newlabel E.false := S.next S1.next := S.begin S.code := gen (S.begin ':') || E.code || gen (E.true':') || S1.code || gen ('goto' S.begin)	Two new labels S.begin which points to the expression's first line and E.true which points to the beginning of the statement S1 is generated. Expression's false should skip the body of the while and should go to the statement following the statement S and hence E.false is assigned S.next. The next of the statement S1 is assigned as S.begin as the exit from the while block if from

Flow of Control Statements: Example

Example : Give three address code for the following:

```
while a<b
do
if c < d then
    x = y + z
else
    x = y - z
done
```

Solution:-

```
L1. if a<b then GOTO L2
GOTO LNEXT
L2. if c<d then GOTO L3
GOTO L4
L3. tl = y + z
x = tl
GOTO L1
L4. tl= y - z
x = tl
GOTO L1
LNEXT.
```