Syntax Directed Definition and Translation

Dr. Venkatesh
Computer Science and Engineering, UVCE

- The third phase of the compiler is called semantic analysis. The main goal of semantic analysis is to check for semantic correctness of program and enable proper execution.
- Semantic analysis phase acts as an interface between syntax phase and code generation phase. It accepts the parse tree from the syntax analysis phase and adds the semantic information such type, value, address, pointer to table etc, to the parse tree and performs certain checks based on this information.
- It checks for operand mismatch, type mismatch, no. of arguments to function, no. operand to operator, data type for array index, L-value, R-value for expression.

Actions performed by semantic analysis phase

- Type checking : Number and type of arguments in function cal and in function definition must be same. Else it results in semantic error.
- Object binding : Associates the variables with respective function definitions.
- Automatic type conversion of integers in mixed mode of operations.
- Helps in intermediate code generation.
- Display appropriate error messages.

The semantics of a language can be described using two notations

1. Syntax Directed Definition (SDD)
2. Syntax Directed Translation (SDT)

Syntax Directed Definition (SDD)

It is a Context Free Grammar with *attributes* and *semantic rules*.
Attributes are associated with grammar symbols.
Semantic rules are associated with productions. These rules are used to compute the attribute values.

Example:

Production : $E \rightarrow E + T$

Semantic rule: $\underbrace{E.val = E.val + T.val} \longrightarrow$ attributes

<u>**Note :**</u>

- Semantic rule is associated with production.
- Attribute name *val* is associated with each non-terminal used in the rule.

Attribute

An attribute is a property of a programming language construct. Attributes are always associated with grammar symbols. Example: Type, Value, address, pointer to symbol table
If X is a grammar symbol and a is an attribute, then X.a denotes the value of the attribute a at a particular node X in the parse tree.

Examples of attributes

1. The data types associated with variables such as int, float, char etc.
2. The value of an expression
3. The location of a variable in memory
4. The object code of a function
5. The number of significant digits in a number etc

Semantic rule

The rule that describes how to compute the attribute values associated with a grammar symbol is called semantic rule.

Example :
Consider the production E → E + T
The attribute value of E which is on LHS of the production denoted by E.val can be calculated by adding the attribute values of variables E and T on RHs of the production.
E.val = E.val + T.val

Types of attributes

1. Synthesized attribute
2. Inherited attribute

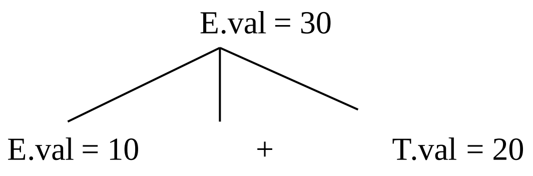
Synthesized attribute

The attribute value of a non-terminal A in the parse tree is defined in-term attribute values of its children or itself is called as synthesized attribute.
Hence the attribute values of synthesized attributes are passed from children to the parent node in bottom-up manner.

Example:

Production :	E → E + T
Semantic rule:	E.val = E.val + T.val

Parse tree with attribute values:



Inherited attribute

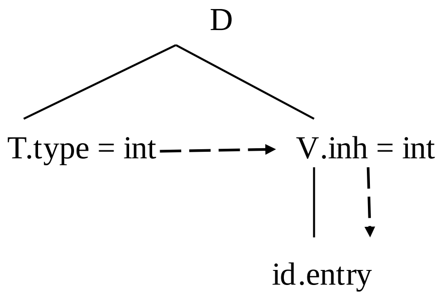
The attribute value of a non-terminal A derived from the attribute values of its siblings or from its parent or itself is called as inherited attribute.
Hence the attribute values of inherited attributes are passed from siblings or from parent to children in top-down manner.

Example:

Production : D → T V
Where D – declaration
T – type such as int
V – variable such as sum

Semantic rule: V.inh = T.type

Parse tree with attribute values:



The type **int** obtained from the Lexical Analyzer is already stored in **T.type** whose value is transferred to its sibling V. ie, V.inh = T.type
Since attribute value of V is obtained from its sibling, it is inherited attribute and its attribute is denoted by **inh**.
Similarly, the value **int** stored in **V.inh** is transferred to its child **id.entry** and hence **entry** is inherited attribute of **id** and attribute value is denoted by **id.entry**.

Annotated Parse Tree

- A parse tree showing the attribute values of each node is called annotated parse tree.
- The terminals in the annotated parse tree can have only synthesized attribute values and they are obtained directly from Lexical Analyzer. So, there are no semantic rules in SDD to get lexical values into terminals of the annotated parse tree. *Terminals can never have inherited attributes.*
- The other nodes in the annotated parse tree may have synthesized or inherited attributes.

Questions

1. Write SDD for the following grammar

$S \rightarrow En$ where n represents end of file marker

$E \rightarrow E+T \mid T$

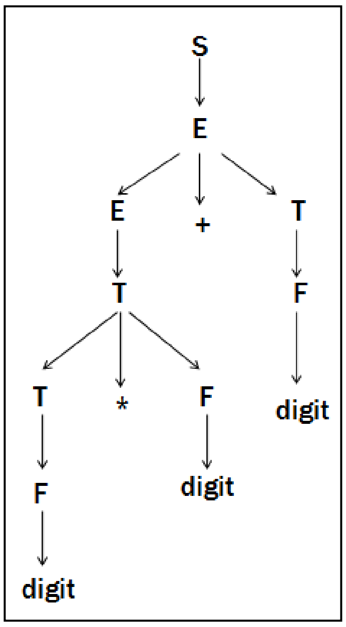
$T \rightarrow T*F \mid F$

$F \rightarrow (E) \mid digit$

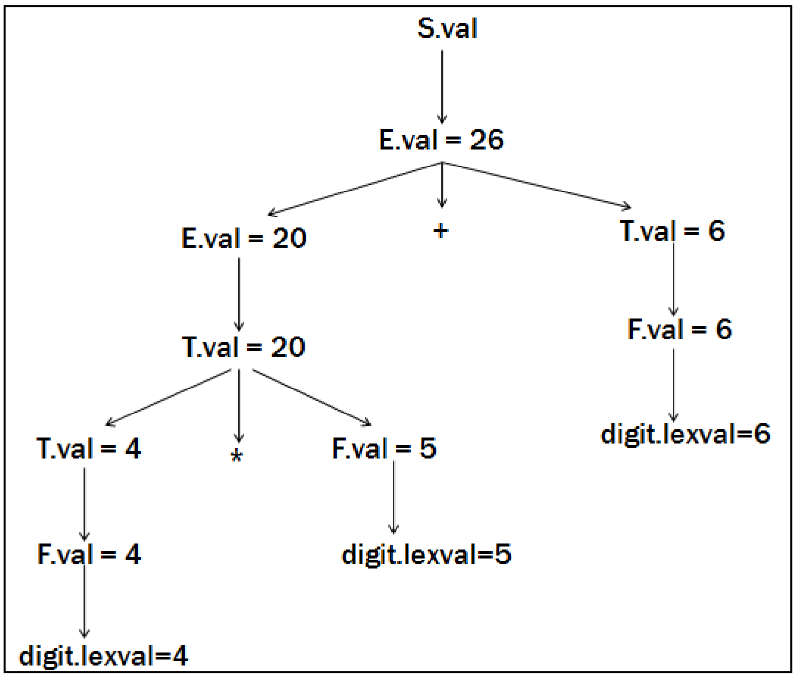
Solution :

Let us assume an input string **4 * 5 + 6** for computing synthesized attributes. The annotated parse tree for the input string is

Parse Tree



Annotated Parse Tree



Productions	**Semantic rules**
$S \rightarrow En$	S.val = E.val
$E \rightarrow E + T$	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
$F \rightarrow (E)$	F.val = E.val
$F \rightarrow digit$	F.val = digit.lexval

2. Write the grammar and SDD for a simple desk calculator and show annotated parse tree for the expression (3+4)*(5+6).

Solution :

A simple desk calculator performs operations such as addition, subtraction, multiplication and division with or without ().

Grammar :

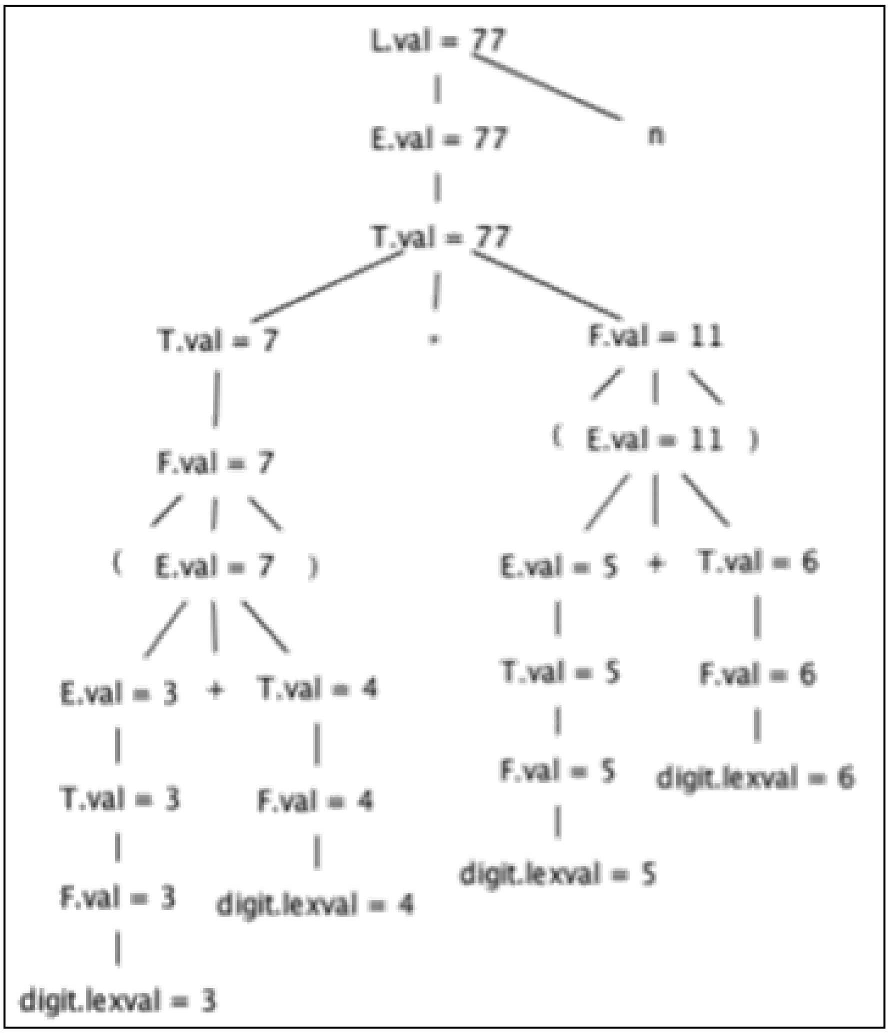
$S \rightarrow En$ where n represents end of file marker

$E \rightarrow E + T \;|\; E - T \;|\; T$

$T \rightarrow T * F \;|\; T / F \;|\; F$

$F \rightarrow (E) \;|\; digit$

Annotated parse tree for the expression (3+4)*(5+6) consisting of attribute values for each non-terminal is given below.



Productions	**Semantic rules**
$S \to En$	S.val = E.val
$E \to E + T$	E.val = E.val + T.val
$E \to E - T$	E.val = E.val - T.val
$E \to T$	E.val = T.val
$T \to T * F$	T.val = T.val * F.val
$T \to T / F$	T.val = T.val / F.val
$T \to F$	T.val = F.val
$F \to (E)$	F.val = E.val
$F \to digit$	F.val = digit.lexval

3. Consider the following grammar

$S \to EN$

$E \to E + T \ | \ E - T \ | \ T$

$T \to T * F \ | \ T / F \ | \ F$

$F \to (E) \ | \ digit$

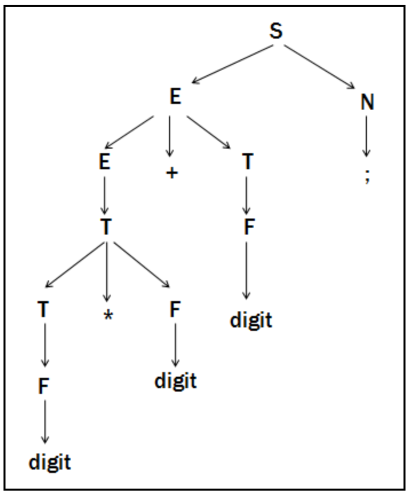
$N \to ;$

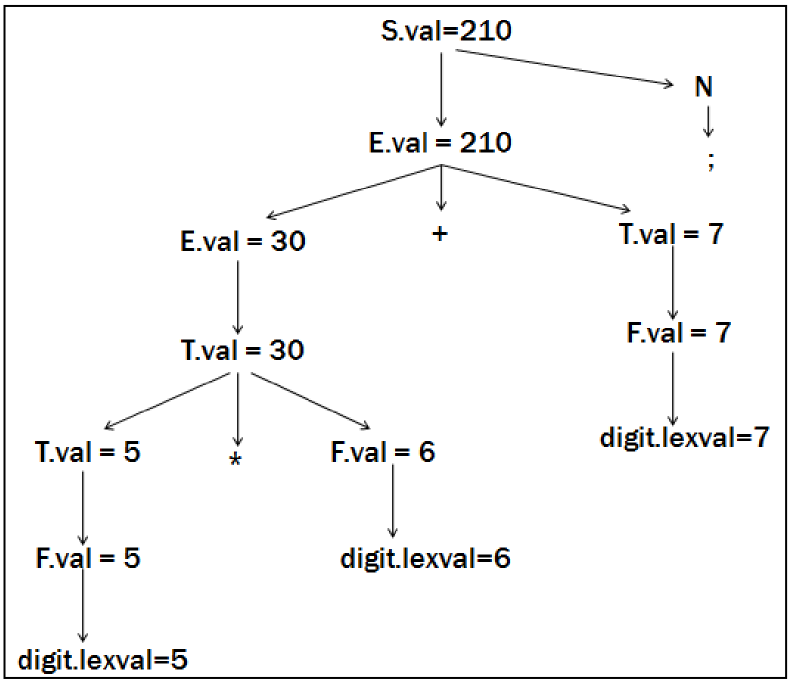
i) **Obtain SDD for the grammar**
ii) **Construct annotated parse tree for the input string 5*6+7**

Solution :

Productions	**Semantic rules**
$S \to EN$	S.val = E.val
$E \to E + T$	E.val = E.val + T.val
$E \to E - T$	E.val = E.val - T.val
$E \to T$	E.val = T.val
$T \to T * F$	T.val = T.val * F.val
$T \to T / F$	T.val = T.val / F.val
$T \to F$	T.val = F.val
$F \to (E)$	F.val = E.val
$F \to digit$	F.val = digit.lexval
$N \to ;$	;

Parse tree **Annotated Parse tree**

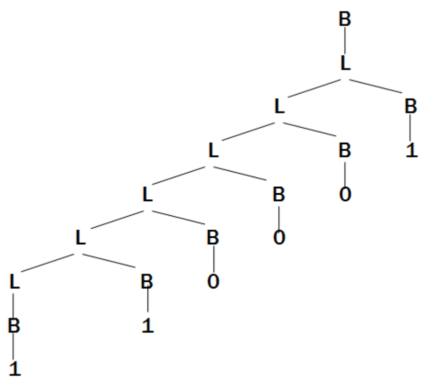


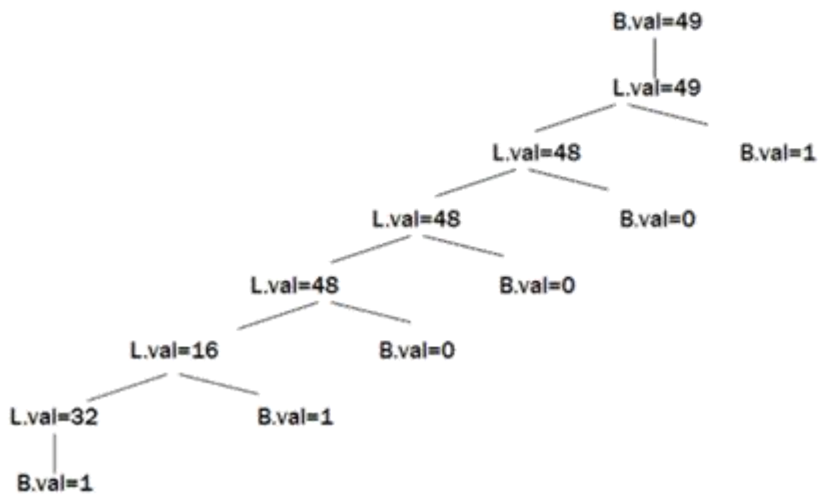


4. The SDD to translate binary integer number into decimal is shown below. Construct the parse tree and annotated parse tree for the string 110001.

Production	**Semantic rule**
B → L	B.val = L.val
L → L,B	L.val = 2*L1.val + B.val
L → B	L.val = B.val
B → 0	B.val = 0
B → 1	B.val = 1

Parse Tree **Annotated Parse Tree**





Syntax Directed Definition (SDD)

In Syntax Directed Translation, along with the grammar, we associate some informal notations and these notations are called as semantic rules.

Syntax Directed Translation = Grammar + Semantic rules

SDTs are used

- To build syntax trees for programming constructs
- To translate infix expressions into postfix notation
- To evaluate expressions

Types of SDT

1. S-attributed SDT
2. L-attributed SDT

S-attributed SDT

1) It uses only synthesized attributes.

   Example :

   $A \rightarrow BCD$ { A.s = f (B.s, C.s, D.s} ie, A can value from the children B or C or D.

2) Semantic actions are placed at right end of the children production. Hence it is also called as Postfix SDT.

   Example :

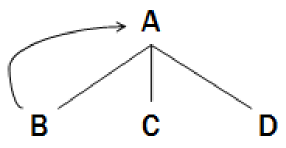
   $A \rightarrow BCD$ {……}

3) Attributes are evaluated during bottom up parsing.

   Example :

   $A \rightarrow BCD$

   We can find the value of A only after evaluating B, C, D values.



L-attributed SDT

1) It uses both synthesized and inherited attributes. Inherited attributes can inherit values from ***either parent or left siblings only.***

   Example :

   A $\rightarrow$ BCD { C.inh=A.inh, C.inh=B.inh, D.inh=B.inh, D.inh=A.inh}

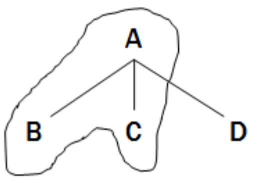
   But C.inh=D.inh is invalid as it takes value from right sibling.

2) Semantic actions are placed anywhere on RHS of the production.

   Example :

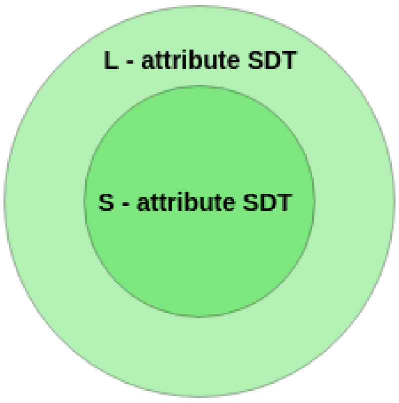
   A $\rightarrow$ {…} BC | B {…} C | BC {…}

3) Attributes are evaluated by traversing the parse tree - *depth first, left to right.*



Note

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



Problems

1. Check whether the following grammar is S-attributed or L-attributes grammar.

A $\rightarrow$ LM { L.inh = f(A.inh);
M.inh = f(L.syn);
A.syn = f(M.syn);
}

Solution :

Semantic rule	Synthesized	Inherited	S-attribute	L-attribute
L.inh = f(A.inh)	No	Yes	No	Yes
M.inh = f(L.syn)	No	Yes	No	Yes
A.syn = f(M.syn)	Yes	No	Yes	Yes

So, this grammar is L-attributed.

2. Check whether the following grammar is S-attributed or L-attributes grammar.

A $\rightarrow$ QR { R.inh = f(A.inh);
Q.inh = f(R.inh);
A.syn = f(Q.syn);
}

Solution :

Semantic rule	Synthesized	Inherited	S-attribute	L-attribute
R.inh = f(A.inh)	No	Yes	No	Yes
Q.inh = f(R.inh)	No	Yes	No	No
A.syn = f(Q.syn)	Yes	No	Yes	Yes

So, this grammar is neither S-attributed nor L-attributed.

3. Write the SDD for a simple type declaration and write the annotated parse tree for the declaration " float id1, id2, id3".

Solution :

The grammar for the simple type declaration is

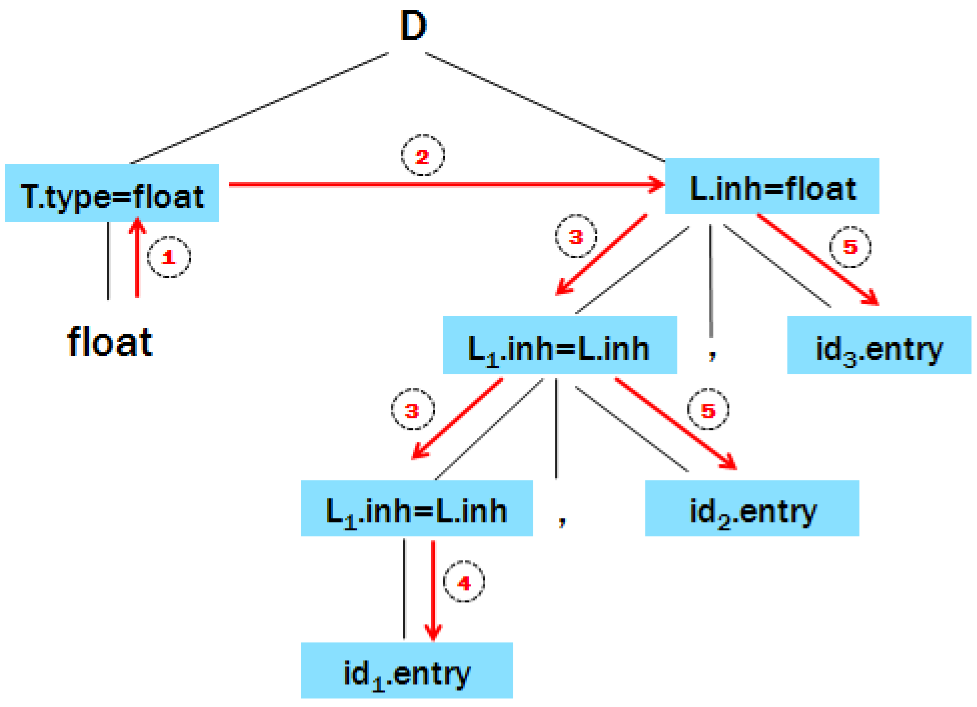
D $\rightarrow$ T L

T $\rightarrow$ int | float

L $\rightarrow$ L$_1$, id | id

Where
D : Declaration
T : Data type(int /float)
L : List of identifiers or identifier

Input string : float id$_1$, id$_2$, id$_3$



1) The declaration D consists of basic data type T followed by list of L identifiers. T can be either **int** or **float.** Thus, the tokens corresponding to **int** or **float** such as *integer* or *float* contained from Lexical analyzer are copied into attribute value of T. The corresponding productions and semantic rules are

Production	**Semantic rule**
$T \rightarrow$ int	T.type=integer
$T \rightarrow$ float	T.type=float

2) The attribute value T.type available in the left subtree should be transferred to the right subtree L. Since attribute value is transferred from left sibling to right sibling, its attribute must be inherited attribute and is denoted by L.inh and can be obtained by the following production.

Production	**Semantic rule**
$D \rightarrow T\ L$	L.inh=T.type

3) The type L.inh must be transferred to identifier id and hence it has to be copied into L1.inh which is the left most child in RHS of the production $L \rightarrow L_1$, id. This can be obtained by the following production.

Production	**Semantic rule**
$L \rightarrow L_1$, id	L_1.inh=L.inh

4) The attribute value of L.inh in turn must be entered as the type for identifier **id** using the production $L \rightarrow$ id. This can be done as follows.

Production	**Semantic rule**
$L \rightarrow$ id	Addtype (id.entry, L.inh)

Addtype()

- id.entry is a lexical value that points to the symbol table.
- L.inh is the type being assigned to every identifier in the list
- The function installs L.inh as the type of corresponding identifier.

5) The attribute value of L.inh in turn must be entered as the type for identifier id which is the right most child in RHS of the production $L \rightarrow L_1$, id. This can be done as follows.

Production	**Semantic rule**
$L \rightarrow L_1$, id	Addtype (id.entry, L.inh)

SDD for the grammar

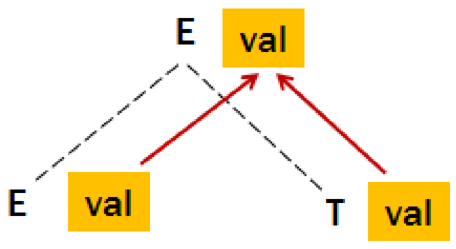
Production	**Semantic rule**
$D \rightarrow T\ L$	L.inh=T.type
$T \rightarrow$ int	T.type=integer
$T \rightarrow$ float	T.type=float
$L \rightarrow L_1$, id	L1.inh=L.inh Addtype (id.entry, L.inh)
$L \rightarrow$ id	Addtype (id.entry, L.inh)

Dependency Graph

A graph that shows the flow of information which helps in computation of various attribute values in a particular parse tree is called dependency graph.
An edge from one attribute instance to another attribute instance indicates that the attribute value of the first is needed to compute the attribute value of the second.
While Annotated parse tree shows the values of attributes, Dependency graph shows how these values are computed.

Example :

Production	**Semantic rules**
E → E + T	E.val = E.val + T.val



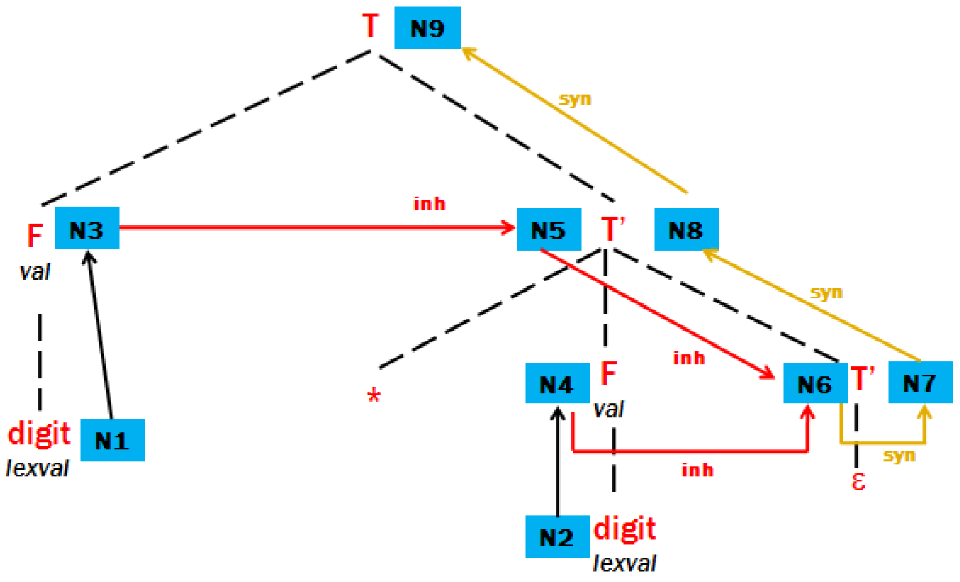
In the above figure, the dotted lines along the nodes connected to them represent the parse tree.
The shaded nodes represented as val with solid arrows originating from one node and ends in another node is the dependency graph.

Example :

T → F T'

T' → * F T' | ε

F → digit



1) Nodes N1 and N2 represent the attribute *lexval* associated with **digit**.
2) Nodes N3 and N4 represent the attribute *val* associated with **F.**
3) Edge from N1 to N3 and edge from N2 to N4 indicate that in the semantic rule, the attribute value F.val is obtained using digit.lexval

4) Nodes N5 and N6 represent the inherited attribute T'.inh associated with each of non-terminal T'.
5) The edge from N3 to N5 indicate that indicate that T'.inh is obtained from its sibling F.val and hence T' is inherited attribute name inh.
6) The edge from N5 to N6 and edge from N4 to N6 indicate that the two attribute values T;.inh and F.val are multiplied to get attribute value at N6.
7) The edge from N6 to N7 indicate that there is an ε production and the attribute value is obtained from itself and hence its attribute value T'.syn is obtained from T'.inh
8) N7 is obtained from itself, N8 is obtained from N7, N9 is obtained from N8. All these are synchronized attributes.
9) T.val at N9 is obtained from its child at N8.

Question :

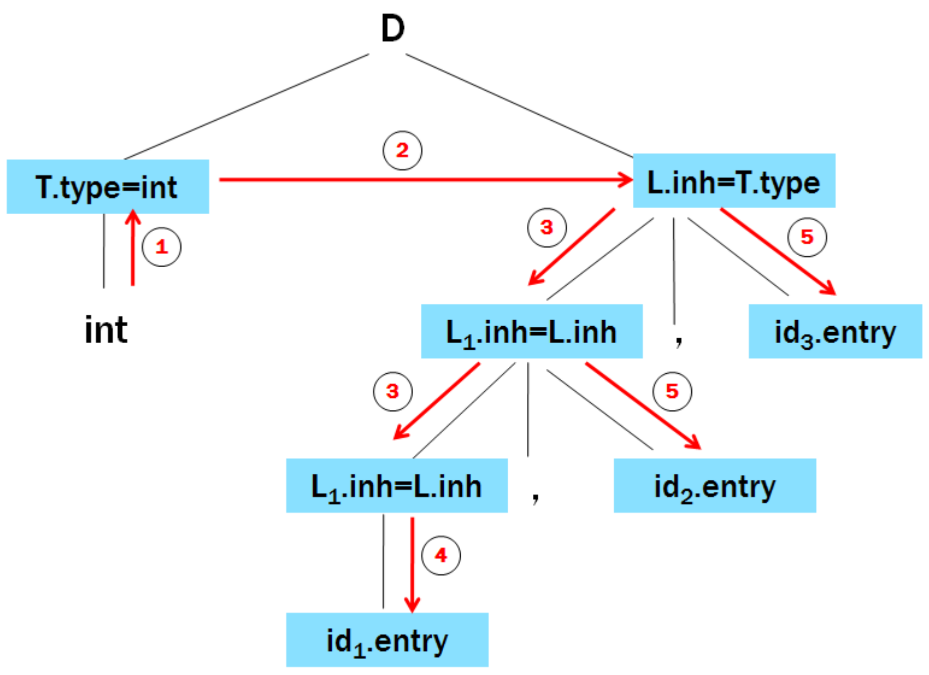
1. Give the SDD to process a sample variable declared in C and dependency graph for the input "int a, b, c"

Solution :

D → T L
T → int
T → float
L → L_1, id
L → id

Production	**Semantic rule**
D → T L	L.inh=T.type
T → int	T.type=integer
T → float	T.type=float
L → L_1, id	L1.inh=L.inh Addtype (id.entry, L.inh)
L → id	Addtype (id.entry, L.inh)

Dependency graph



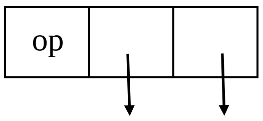
Construction of Syntax Tree

The syntax tree is an abstract representation of the language constructs. The syntax trees are used to write the translation routines using SDD. Constructing syntax tree for an expression means translation of expression into postfix form.

Functions

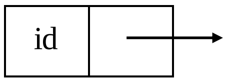
1. mknode(op, left, right)
2. mkleaf(id, entry)
3. mkleaf(num, val)

1. mknode(op, left, right)



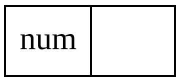
This function creates a node with a filed operator having *op* as label and two pointers *left* and *right.*

2. mkleaf(id, entry)



This function creates a node for an identifier with label *id* and a pointer to symbol table is given by *entry*.

3. mkleaf(num, val)



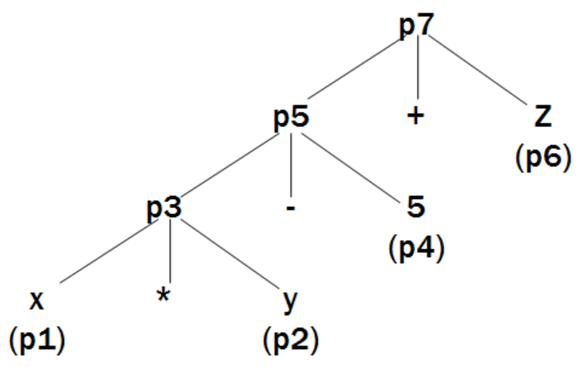
This function creates a node for number with label *num* and *val* is for value of that number.

Questions

1. Construct syntax tree for the expression x*y-5+z

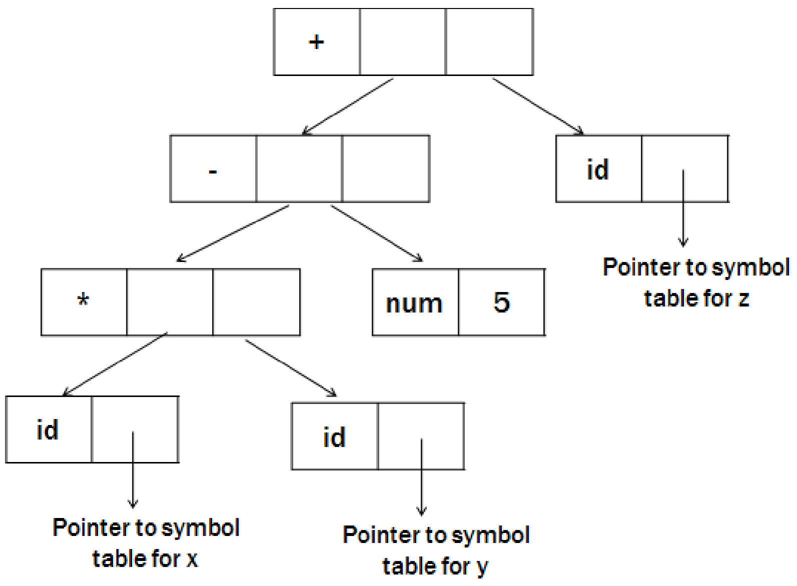
Solution :

Convert infix expression to postfix expression : xy*5-z+



Symbol	Operation
x	p1=mkleaf(id, ptr for x)
y	p2=mkleaf(id, ptr for y)
*	p3=mknode(*, p1, p2)
5	p4=mkleaf(num, 5)
-	p5=mknode(-, p3, p4)
z	p6=mkleaf(id, ptr for z)
+	p7=mknode(+, p5, p6)

Syntax tree

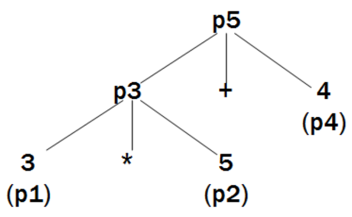


+
-
id
Pointer to symbol table for z
*
num
5
id
id
Pointer to symbol table for x
Pointer to symbol table for y

2. Construct syntax tree for the expression 3*5+4

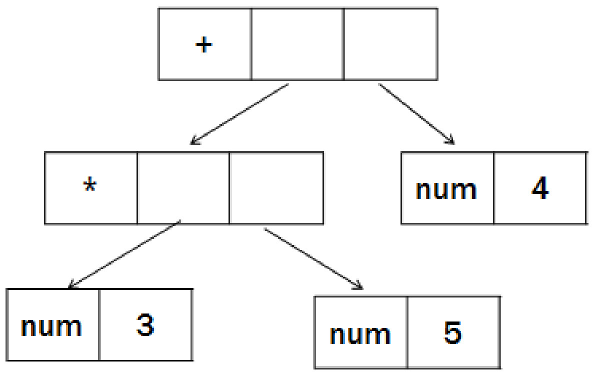
Solution :

Convert infix expression to postfix expression : 35*4+



Symbol	Operation
3	p1=mkleaf(num, 3)
5	p2=mkleaf(num, 5)
*	p3=mknode(*, p1, p2)
4	p4=mkleaf(num, 4)
+	p5=mknode(+, p3, p4)

Syntax tree



3. Assuming suitable SDD, construct syntax tree for the expression a-4+e

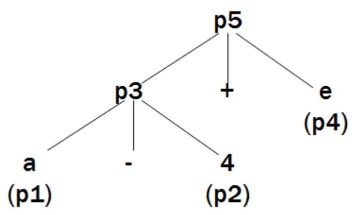
Solution :

Production	**Semantic Rule**
S → En	S.val = E.val

E → E+T | E-T | T	E.val = E.val + T.val
	E.val = E.val – T.val
	E.val = T.val

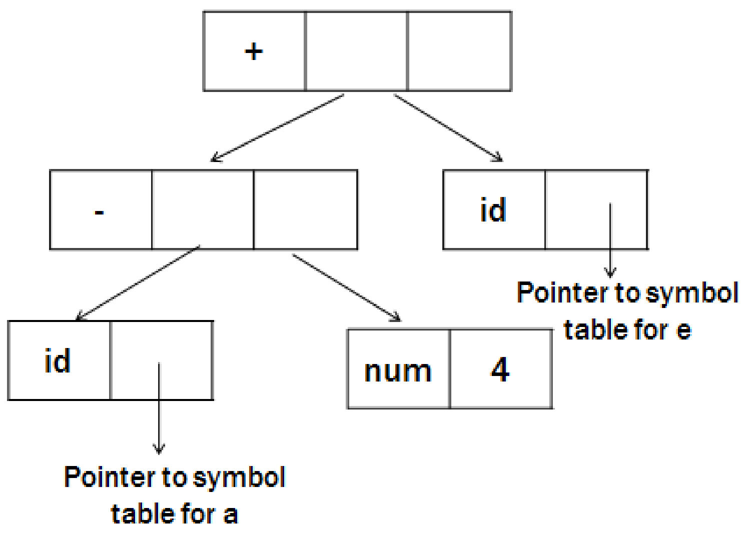
T → F	T.val = F.val
F → digit	F.val = digit.lexval
F → id	F.val = addType(id, entry)

Convert infix expression to postfix expression : **a4-e+**



Symbol	**Operation**
a	p1=mkleaf(id, ptr for a)
4	p2=mkleaf(num, 4)
-	p3=mknode(-, p1, p2)
e	P4=mkleaf(id, ptr for e)
+	p5=mknode(+, p3, p4)

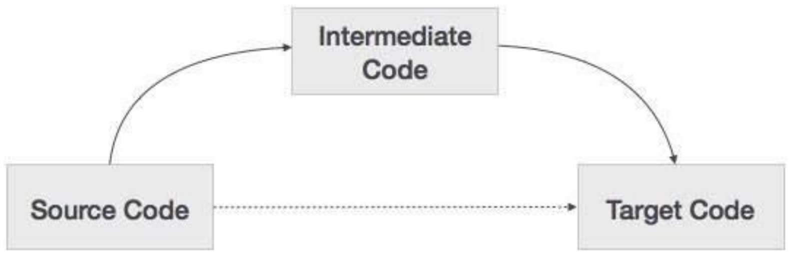
Syntax tree



Intermediate Code Generation

In the analysis-synthesis model of a computer, the front end of a compiler translates a source program into an independent intermediate code and then back end of the compiler uses this intermediate code to generate the target code.
The benefits using machine independent intermediate code are
It is easy to change the source or the target language by adapting only the front-end or back-end.
It makes optimization easier.
The intermediate representation can be directly interpreted.

Intermediate representations



The intermediate representation is chosen in such a way that

- It should be easy to translate the source language to the intermediate representation
- It should be easy to translate the intermediate representation to the machine code.
- The intermediate representation should be suitable for optimization

The following are commonly used intermediate code representation

1. Linear representation - Postfix notation (POSIX)
2. Graphical representation
3. Three-address code
4. Static single Assignment form (SSA)

1. Linear representation - Postfix notation (POSIX)

Infix expression : a+b

Postfix expression : ab+

No ( ) are needed in postfix notation because the position and no. of arguments of the operators permit only one way to decode a postfix expression. In postfix notation, the operator follows the operand.

Example :

Infix representation :	(a-b)*(c+d)+(a-b)
Postfix representation :	ab-cd+*ab-+

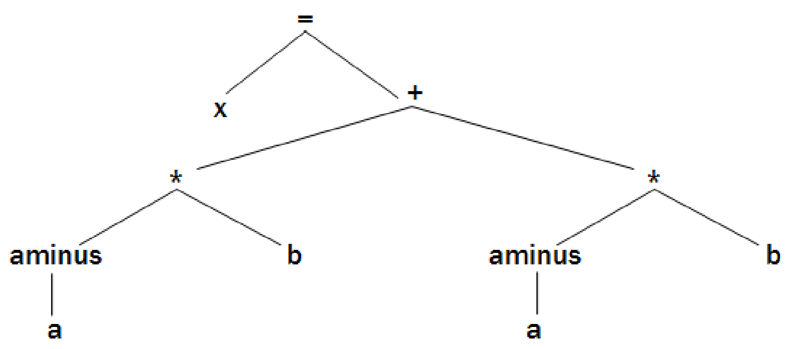
2. Graphical representation

a) **Syntax Tree**

Syntax tree is a condensed form of a parse tree. The operator and keyword nodes of the parse tree are moved to their parents and a chain of single productions is replaced by single link in syntax tree. The internal nodes are operators and child nodes are operands. To form syntax tree, write ( ) in the expression. This way it is easy to recognize which operand should come first.

Example :

x = -a * b + -a * b

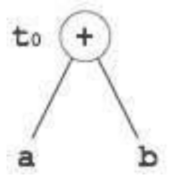


b) **DAG (Directed Acyclic Graph)**

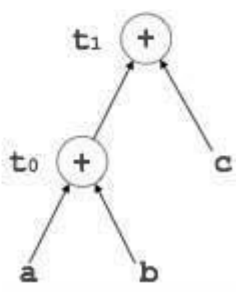
DAG is a directed graph that contains no cycles. A DAG is used to represent common sub expressions. It has leaves corresponding to operands and interior nodes corresponding to operators. A node may have more than one parent.

Example 1 :

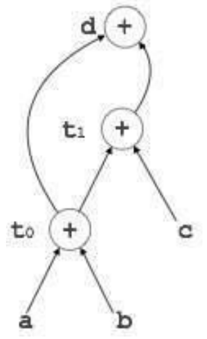
$t_0 = a + b$
$t_1 = t_0 + c$
$d = t_0 + t_1$



[$t_0 = a + b$]



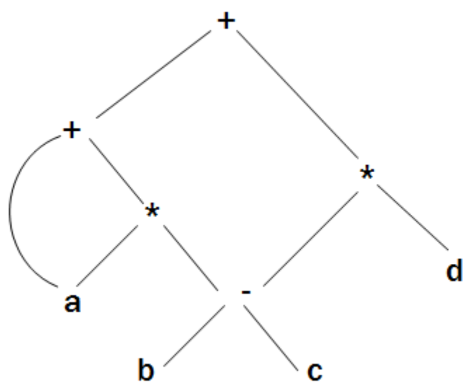
[$t_1 = t_0 + c$]



[$d = t_0 + t_1$]

Example 2 :

a + a * (b – c) + (b – c) * d



Construction of DAG

Step 1:

For each 3-address instruction of the form ***x* := *y op z*** do the following activities

a) Find a node labeled y. If none exists, create it.
b) Find a node labeled z. If none exists, create it.
c) Find a node labeled op with y as the left child and z as the right child. If none found, then create one and call this node N. If node op exists, whose name is N with y as left child and z as right child, then add x to the list of identifiers attached to N.

Step 2:

For each 3-addres instruction of the form ***x* := *y***, do the following activities.

a) Find a node labeled y. If node does not exist, create it and name it N. If exists, name it N.
b) Add x to the list of identifiers attached to N.

Step 3:

For each 3-address instruction of the form ***x* := *-y***, do the following activities.

a) Find a node labeled y. If node does not exist, create it.
b) Find a node labeled – with y as the child. If none exists, create it. Call this node N.
c) Add x to the list of identifiers attached to N.

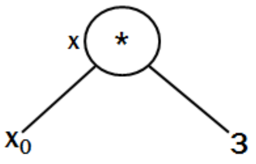
Questions

1. Give the DAG representation for the following basic block.

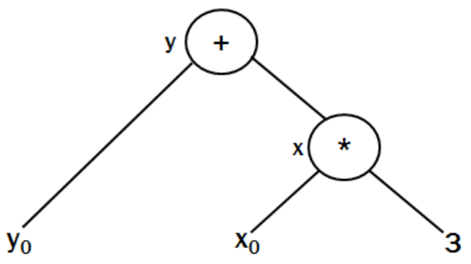
x=x*3
y=y+x
x=y-z
y=x

Solution :

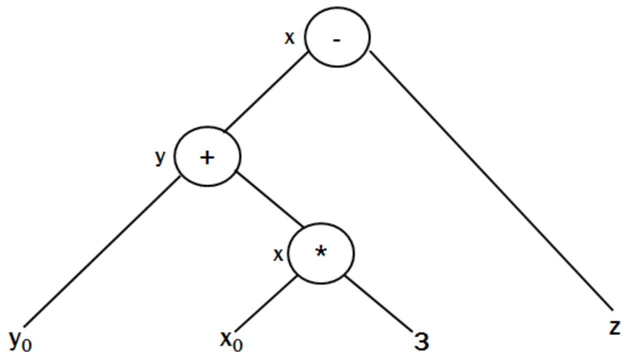
1. x = x*3



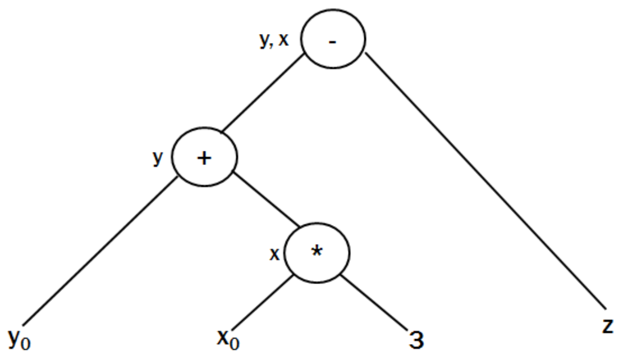
2. y = v+x



3. x = y-z



3. y = x



2. Obtain the DAG representation for the expression (a+b) * (a+b+c)

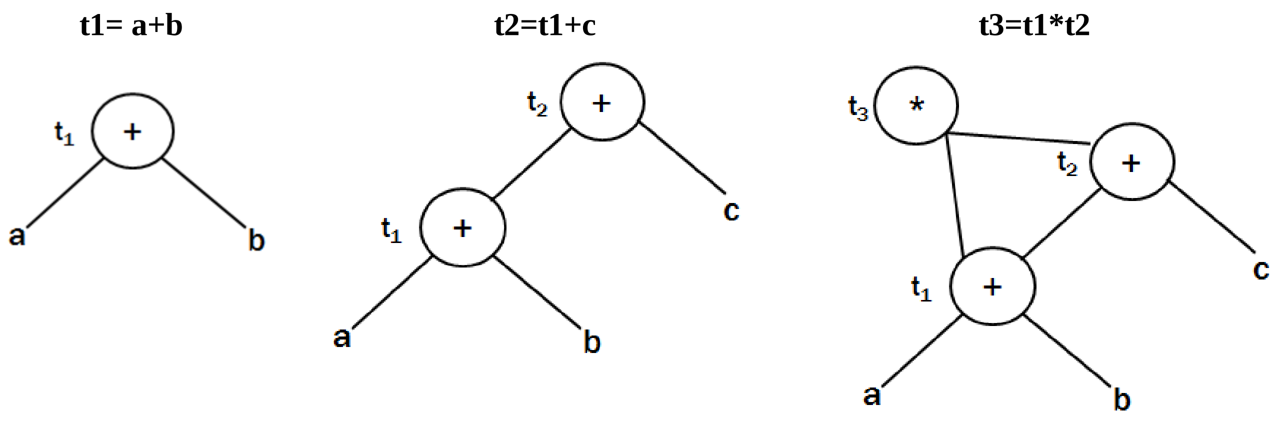
Solution :

Covert to 3-address instruction.

t1= a+b

t2=t1+c

t3=t1*t2



3. Obtain the DAG representation for the expression a + a * (b - c) + (b – c) * d

Solution :

Note : There is no parenthesis for a+a. * has the highest precedence among +, *, -

Hence the expression order will be a + (a * (b - c)) + ((b – c) * d)

Covert to 3-address instruction.

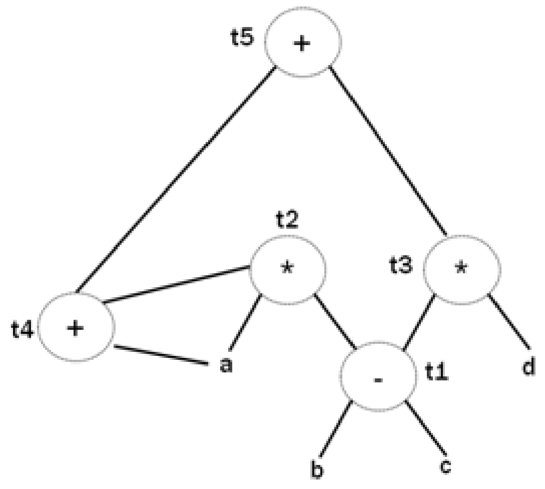
t1 = b – c

t2= a * t1

t3 = t1 * d

t4= a + t2

t5 = t4 + t3



4. Obtain the DAG representation for the 3-address representation

a=b*c

d=b

e=d*c

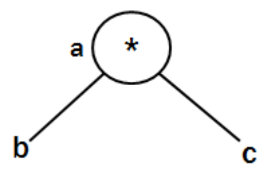
b=e

f=b+c

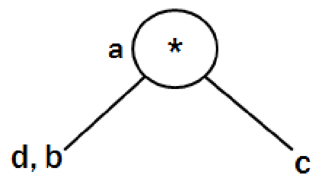
g=f+d

Solution :

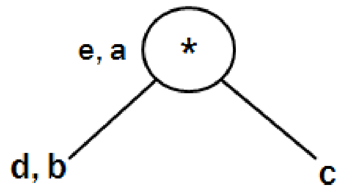
a=b*c



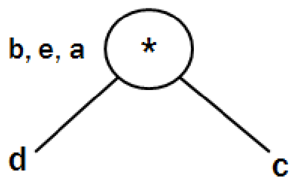
d =b



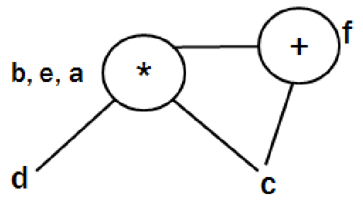
e=d*c



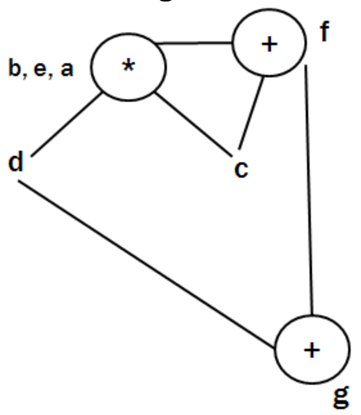
b=e



f=b+c



g=f+d



5. Obtain the DAG representation for the expression (((a+a) + (a+a)) + ((a+a) + (a+a)))

Solution :

Covert to 3-address instruction.

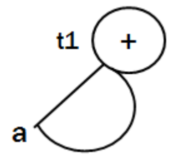
t1=a+a

t2=t1+t1

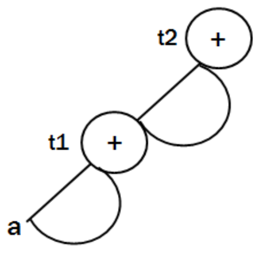
t3=t2

t4=t3+t3

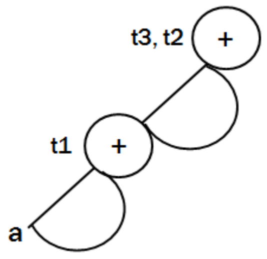
t1=a+a



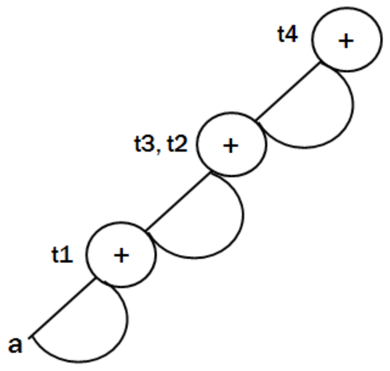
t2=t1+t1



t3=t2



t4=t3+t3



6. Obtain the DAG representation for the expression (a + b) * (a + b) + c + d

Solution :

Covert to 3-address instruction.

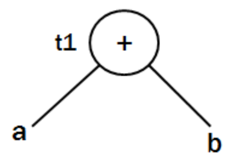
t1=a+b

t2=t1*t1

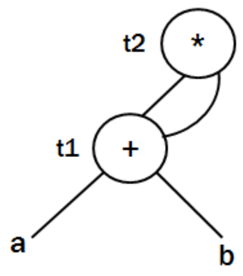
t3=t2+c

t4=t3+d

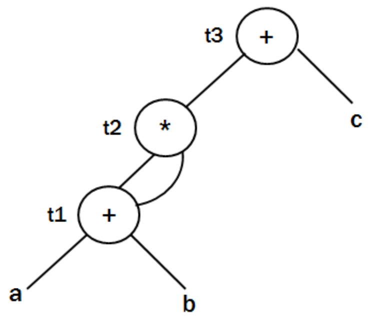
t1=a+b



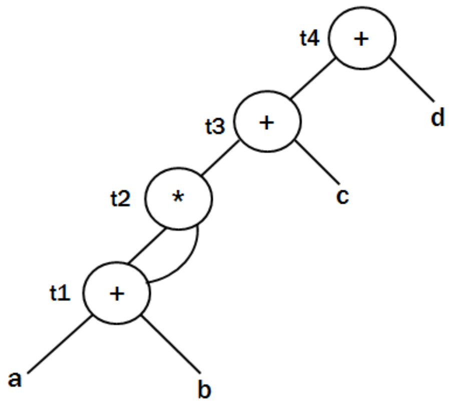
t2=t1*t1



t3=t2+c



t4=t3+d



6. Obtain the DAG representation for the expression (a + a) * (b - c) + (b – c) * d

Solution :

Covert to 3-address instruction.

t1=a+a

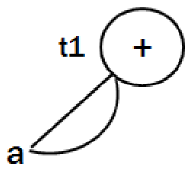
t2=b-c

t3=t1*t2

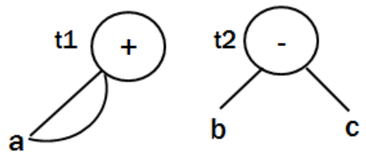
t4=t2*d

t5=t3+t4

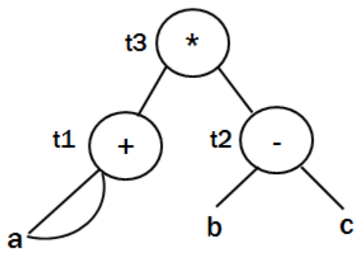
t1=a+a



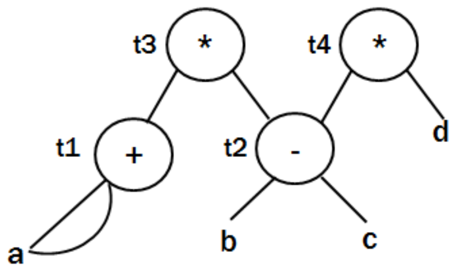
t2=b-c



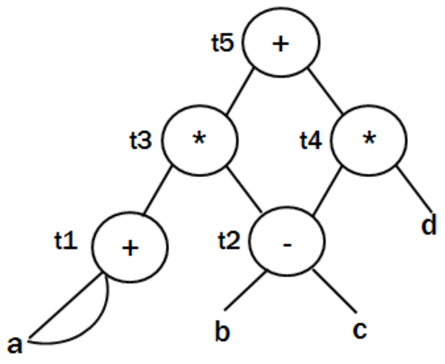
t3=t1*t2



t4=t2*d



t5=t3+t4



7. Obtain the DAG representation for the expression ((x+y) – (x+y) * (x-y)) + ((x+y) * (x-y))

Solution :

Covert to 3-address instruction.

t1=x+y

t2=x-y

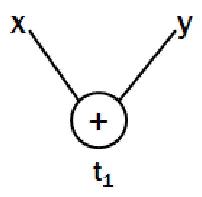
t3=t1*t2

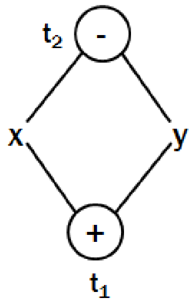
t4=t1-t3

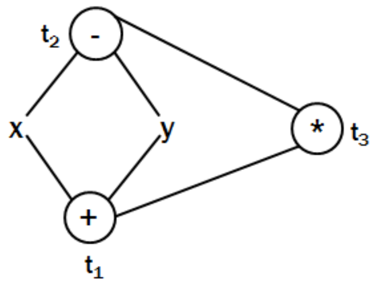
t5=t3

t6=t4+t5

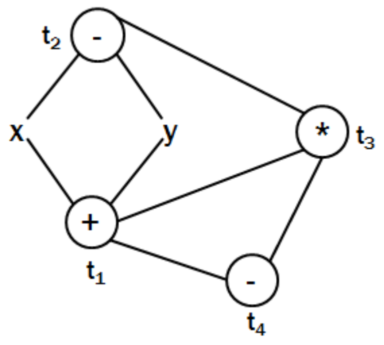
t1=x+y **t2=x-y** **t3=t1*t2**



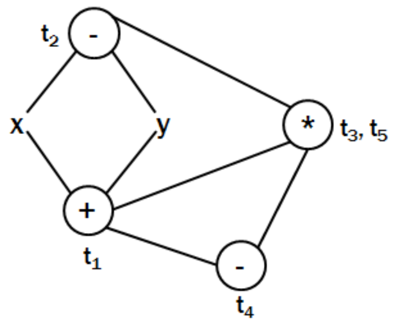




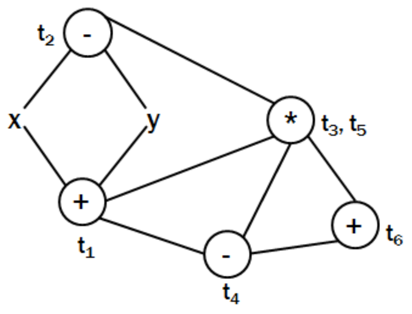
t4=t1-t3



t5=t3



t6=t4+t5



3. Three address code

In the three address code form, at the most, three addresses are used to represent any statement. The general form of three address code representation is a:= b op c where a, b, c are operands and op is an operator.

Example :

Consider the expression x = a + b + c. Its three-address code will be

$t_0 = a + b$

$t_1 = t_0 + c$

$x = t_1$

where t_0 and t_1 are temporary names generated by the compiler.

The various types of three address statements are

1	Assignment statement	x := y op z	op is arithmetic or logical operator to perform binary operation.
2	Assignment instruction	x := op y	Performs unary operation, op here is an unary operator.
3	Copy statement	x := y	The value of y is assigned to x.
4	Unconditional jump	goto L	The control goes to the statement labelled by L.
5	Conditional jump	If x relop y goto L	The relop indicates relational operators such as <, >, <=, >=. If x relop y is true, then it executes goto L statement.

6	Procedure calls	pqr (x)
		{
		…..
		return y;
		}
			Here x is used as the parameter to procedure pqr.
			The return statement indicates to return the value of y.

7	Indexed assignment	x := y[i]
		x[i] := y
			The value of array y at ith index is assigned to x.
			The value of identifier y is assigned to index i of array x.

8	Address and pointer assignment	x := &y
		x := *y
		*x := y
			The value of x will be the address or location of y
			y is a pointer whose value is assigned to x.
			r-value of object pointed by x is set by l-value of y.

Implementation of three address statements

Three address code is an abstract form of intermediate code that can be implemented as records with fields for operator and operands. The three representations are

a) Quadruple representation
b) Triple representation
c) Indirect triple representation

a) Quadruple representation

In quadruple representation, each instruction is divided into four fields - op, arg1, arg2 and result.

- The op field is used to represent the internal code for the operator.
- The arg1 and arg2 represent two operands
- The result is used to store the result of the expression.

Example :

a:= -b * c + d

Three address code	Location	op	arg1	arg2	result
t1 = -b	(0)	uminus	b	-	t1
t2 = c + d	(1)	+	c	d	t2
t3 = t1 * t2	(2)	*	t1	t2	t3
a = t3	(3)	:=	t3	-	a

b) Triple representation

- In this representation, the use of temporary variables is avoided.
- Instead, references to instructions are made.
- The triple is a record field containing three fields op, arg1, arg2.

Example :

a := -b * c + d

Three address code	Location	op	arg1	arg2
t1 = -b	(0)	uminus	b	-
t2 = c + d	(1)	+	c	d
t3 = t1 * t2	(2)	*	(0)	(1)
a = t3	(3)	:=	(2)	-

c) Indirect Triple representation

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.

Example :

a = b * – c + b * – c

Three address code	Location	op	arg1	arg2
t1=-c	(1)	uminus	c	-
t2=b*t1	(2)	*	b	(1)
t3=-c	(3)	uminus	c	
t4=b*t3	(4)	*	b	(3)
t5=t2+t4	(5)	+	(2)	(4)
a=t5	(6)	:=	(5)	-

Address	Location
30	(1)
31	(2)
32	(3)
33	(4)
34	(5)
35	(6)

4. Static Single Assignment form (SSA)

SSA is an intermediate representation that facilitates certain code optimizations.

Example :

Note that the subscripts distinguish each definition of variables p and q in SSA representation.

3 address code	SSA form
p = a + b	$p_1 = a + b$
q = p – c	$q_1 = p_1 - c$
p = q * d	$p_2 = q_1 * d$
p = e – p	$p_3 = e - p_2$
q = p + q	$q_2 = p_3 + q_1$

Code Generation

- Code generation is the final phase in the compiler design.
- The code optimizer accepts intermediate code representation which is generated from the front end of the compiler and produces another intermediate code representation which is optimized.
- Code generator takes intermediate representation produced by code optimizer along with supplementary information in symbol table of the source program and produces as output an equivalent target program.

- Code generator has 3 main tasks.
    - Instruction selection : Choose an appropriate target machine instructions to implement intermediate representation statements.
    - Register allocation and assignment : Decide what values to keep in which registers.
    - Instruction ordering : Decide in what order the schedule the execution of instructions.

Issues in the design of a code generator

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 quadruples, triples, indirect triples, Postfix notation, syntax trees, DAG's, etc. 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, relocatable machine language, assembly language. 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 are subdivided into two subproblems:
   1. During **Register allocation** – we select only those set of variables that will reside in the registers at each point in the program.
   2. During a subsequent **Register assignment** phase, the specific register is picked to access the variable.

   As the number of variables increases, the optimal assignment of registers to variables becomes difficult. Mathematically, this problem becomes NP-complete. Certain machine requires register pairs consist of an even and next odd-numbered register. Example : M a, b

   These types of multiplicative instruction involve register pairs where the multiplicand is an even register and b, the multiplier is the odd register of the even/odd register pair.

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. However, picking the best order in the general case is a difficult NP-complete problem.

Approaches to code generation issues

Code generator must always generate the correct code. It is essential because of the number of special cases that a code generator might face. Some of the design goals of code generator are:

- Correct
- Easily maintainable
- Testable
- Efficient

The target language

It is a 3-address machine language with the following format

op target, source1, source2

The target machine is byte addressable ie, it can access 8 bit of information from the specified address.

It has n no. of registers denoted by R_0, R_1, R_2, ….. R_{n-1}.

The various types of instructions that are supported by the target machine are

1) Load instructions
2) Store instructions
3) Computational instructions
4) Unconditional instructions
5) Conditional jumps

1. Load instructions

They are used to copy the data into the destination operand which must be a register.

Syntax :

LD destination, address

The second operand can either be a register or a memory location.

Example :

LD R1, R2

LD R1, A

2. Store instructions

It is the opposite of load instruction. It is used to copy the data into memory location specified in the destination operand.

Syntax :

ST destination, register

Destination must be a memory location

Example :

ST A, R1

3. Arithmetic instruction

The arithmetic operations are performed using these instructions.

Syntax :

OP destination, source1, source2

Example :

ADD R0, R1, R2

SUB R0, R0, R1

MUL R2, R0, R1

4. Unconditional jump instructions

The branch instruction without any condition is called unconditional jump instruction.

Syntax :

BR label or *JMP label*

Where BR stands for **BR**anch instruction.

4. Conditional Jump instructions

Based on the value stored in a register ie, whether it is +ve or –ve or zero, if branching takes place, then the instructions are called conditional branch instructions.

Syntax :

B[condition] register, label

B – Branch

Condition – LT for less than GT for greater than

Example :

BL R0, T1

BLTZ R1, T2

(Branch on less than zero)

The addressing modes that are supported by generalized target machine are

a) Direct addressing

Address of the data to be accessed is directly present in the instruction ie, If a location is identified by a variable name x, the value stored in a memory location can be accessed directly using x.

Example : *LD R1, R2*

Load the content of register R2 to R1.

b) Indexed addressing

The data can be accessed from a memory location using an index.

Example : *LD R1 A[R2]*

c) Indexed addressing where a memory location is integer

Same as the previous one except that a memory location is represented as an integer.

Example : *LD R1, 100[R2]*

Load the content of memory address (100 + content of register R2) to register R1.

d) Indirect addressing

The contents of data can be accessed by de-referencing using * operator.

Example :

*LD R1, *(R2)*

LOAD R1, @100

Load the content of memory address stored at memory address 100 to the register R1.

e) Immediate addressing

The data to be manipulated is directly present in the instruction and proceeded by #.

Example : *LD R1, #100*

Questions

1. Generate the code for 3-address statement x = y – z

```
LD R1, y            // R1=y
LD R2, z            // R2=z
SUB R1, R1, R2      // R1=R1-R2
ST x, R1            // x=R1
```

2. Generate the code for 3-address statement x=*p

```
LD R1, p            // R1=p
LD R2, 0(R1)        // R2=*p
ST x, R2            // x=R2
```

3. Generate the code for 3-address statement *p=x

```
LD R1, x            // R1=x
LD R2, p            // R2=p
ST 0(R2), R1        // *p=x
```

4. Generate the code for 3-address statement : if(x<y) goto L

```
LD R1, x            // R1 = x
LD R2, y            // R2 = yo
SUB R1, R1, R2      // R1 = R1 - R2
BLTZ R1, M          // if R1<0 jump M
```

5. Generate the code for 3-address statement b=a[i]

```
LD R1, i            // R1=i
MUL R1, R1, 8       // R1=R1*8, array elements are 8 byte values
LD R2, a(R1)        // R2=a(R1)
ST b, R2            // b=[a(R2)]
```

6. Generate the code for 3-address statement a[i]=b

```
LD R1, b            // R1=b
LD R2, i            // R2=i
MUL R2, R2, 8       // R2=R2*8
ST a(R2), R1        // a[i]=b
```

Program and instruction cost

Cost of the program = compilation cost + run time cost
Cost of each instruction = 1 + cost(addressing mode)
where

- Registering addressing mode where both operands are registers having cost=0
- Addressing modes with variables have cost=1
- Addressing modes with constants have cost =1
- Indirect addressing modes have cost=1

Questions

1. Determine the cost of the following sequence of instructions

```
LD R0, y            // cost = 1 + 1 = 2
LD R1, z            // cost = 1 + 1 = 2
ADD R0, R0, R1      // cost = 1 + 0 = 1
ST x, R0            // cost = 1 + 1 = 2
```

Total cost = 7

2. Determine the cost of the following sequence of instructions

```
LD R0, i            // cost = 1 + 1 = 2
MUL R0, R0, 8       // cost = 1 + 1 = 2
LD R1, a(R0)        // cost = 1 + 1 = 2
ST b, R1            // cost = 1 + 1 = 2
```

Total cost = 8

3. Determine the cost of the following sequence of instructions

```
LD R0, c            // cost = 1 + 1 = 2
LD R1, i            // cost = 1 + 1 = 2
MUL R1, R1, 8       // cost = 1 + 1 = 1
ST a(R0), R0        // cost = 1 + 1 = 2
```

Total cost = 8

4. Determine the cost of the following sequence of instructions

```
LD R0, p            // cost = 1 + 1 = 2
LD R1, 0(R0)        // cost = 1 + 1 = 2
ST x, R1            // cost = 1 + 1 = 2
```

Total cost = 6

Code Optimization Techniques

1. Compile Time Evaluation
2. Common sub-expression elimination
3. Dead Code Elimination
4. Code Movement
5. Strength Reduction

1. Compile Time Evaluation

Two techniques that falls under compile time evaluation are

a) Constant Folding

In this technique,

- As the name suggests, it involves folding the constants.
- The expressions that contain the operands having constant values at compile time are evaluated.
- Those expressions are then replaced with their respective results.

Example:

Circumference of Circle = (22/7) x Diameter

Here,

- This technique evaluates the expression 22/7 at compile time.
- The expression is then replaced with its result 3.14.
- This saves the time at run time.

b) Constant Propagation

In this technique,

- If some variable has been assigned some constant value, then it replaces that variable with its constant value in the further program during compilation.
- The condition is that the value of variable must not get alter in between.

Example:

pi = 3.14, radius = 10, Area of circle = pi x radius x radius

Here,

- This technique substitutes the value of variables 'pi' and 'radius' at compile time.
- It then evaluates the expression 3.14 x 10 x 10.
- The expression is then replaced with its result 314.
- This saves the time at run time.

2. Common Sub-Expression Elimination

In this technique,

- As the name suggests, it involves eliminating the common sub expressions.
- The redundant expressions are eliminated to avoid their re-computation.
- The already computed result is used in the further program when required.

Example :

Code Before Optimization	**Code After Optimization**

S1 = 4 x i
S2 = a[S1]
S3 = 4 x j
S4 = 4 x i // **Redundant Expression**
S5 = n
S6 = b[S4] + S5
	S1 = 4 x i
	S2 = a[S1]
	S3 = 4 x j
	S5 = n
	S6 = b[S1] + S5

3. Code Movement

In this technique,

- As the name suggests, it involves movement of the code.
- The code present inside the loop is moved out if it does not matter whether it is present inside or outside.
- Such a code unnecessarily gets execute again and again with each iteration of the loop.
- This leads to the wastage of time at run time.

Example :

Code Before Optimization	Code After Optimization

for (int j = 0 ; j < n ; j ++)
{
   x = y + z ;
  a[j] = 6 x j;
}
	x = y + z ;
	for (int j = 0 ; j < n ; j ++)
	{
	    a[j] = 6 x j;
	}

4. Dead Code Elimination

In this technique,

- As the name suggests, it involves eliminating the dead code.
- The statements of the code which either never executes or are unreachable or their output is never used are eliminated.

Example :

Code Before Optimization	Code After Optimization

i = 0 ;
if (i == 1)
{
   a = x + 5 ;
}
	i = 0 ;

5. Strength Reduction

In this technique,

- As the name suggests, it involves reducing the strength of expressions.
- This technique replaces the expensive and costly operators with the simple and cheaper ones.

Example :

Code Before Optimization	Code After Optimization
B = A x 2	B = A + A

Here,

- The expression “A x 2” is replaced with the expression “A + A”.
- This is because the cost of multiplication operator is higher than that of addition operator.

Three Address code statement for

Question If A < B and C < D then t = 1 else t = 0

Solution

1) If (A < B) goto (3)

(2) goto (4)

(3) If (C < D) goto (6)

(4) t = 0

(5) goto (7)

(6) t = 1

(7)

Three Address code statement for

If A < B then 1 else 0

Solution

(1) If (A < B) goto (4)

(2) T1 = 0

(3) goto (5)

(4) T1 = 1

(5)

Three Address code statement for
Question
-(a x b) + (c + d) – (a + b + c + d)
Solution

(1) T1 = a x b

(2) T2 = uminus T1

(3) T3 = c + d

(4) T4 = T2 + T3

(5) T5 = a + b

(6) T6 = T3 + T5

(7) T7 = T4 – T6

Three Address code statement for

```
if (a < b + c)
 a = a - c;
c = b * c;
```

Solution

```
t1 = b + c;
t2 = a < t1;
If  t2 Goto L0;
t3 = a - c;
a = t3;

L0:  t4 = b * c;
c =  t4;
```

Relational operators

```
t3 = t2 == t1;
t3 = t2 < t1;
t3 = t2 && t1;
t3 = t2 || t1;
```

if statement
conditional jump

```
L1:
Goto L1;
IfZ t1 Goto L1;
```

```
void main()
{
int a;
a = 2 + a;
Print(a);
}
```

```
main:
 BeginFunc 12;
 _t0 = 2;
 _t1 = _t0 + a;
 a = _t1;
 PushParam a;
 LCall _PrintInt;
 PopParams 4;
 EndFunc;
```

```
void main()
{
int b;
int a;
b = 3;
a = 12;
a = (b + 2)-(a*3)/6;
}
```

```
main:
 BeginFunc 44;
 _t0 = 3;
 b = _t0;
 _t1 = 12;
 a = _t1;
 _t2 = 2;
 _t3 = b + _t2;
 _t4 = 3;
 _t5 = a * _t4;
 _t6 = 6;
 _t7 = _t5 / _t6;
 _t8 = _t3 - _t7;
 a = _t8;
 EndFunc;
```

```
S -> while E do S1

{ S.begin = newlabel;
S.after = newlabel;
S.code = gen(S.begin ':')
 E.code
 gen('if' E.place '=' '0' 'goto' S.after)
 S1.code
 gen('goto' S.begin)
 gen(S.after ':')
```

```
arr[1] = arr[0] * 2;
```

```
_t0 = 1;
 _t1 = 4;
 _t2 = _t1 * _t0;
 _t3 = arr + _t2;
 _t4 = 0;
 _t5 = 4;
 _t6 = _t5 * _t4;
```