5 Intermediate Code Generation

5.1 BASIC OF INTERMEDIATE CODE GENERATION

- Compiler may generate sequential intermediate representations to translate a source code to a target machine code language.

Source program → High level intermediate representation → ..... → Low level intermediate representation → Target code

- Source program is associated with "High level intermediate representation", and the target machine code is with "Low level intermediate representation".
- Syntax tree is high level intermediate representation which shows the natural hierarchical structure of the source program. They are well suited for the tasks like static type checking.
- Register allocations and instruction selection these kinds of machine dependent jobs can be done efficiently using "Low level intermediate representation".

5.2 REPRESENTATION OF INTERMEDIATE CODE

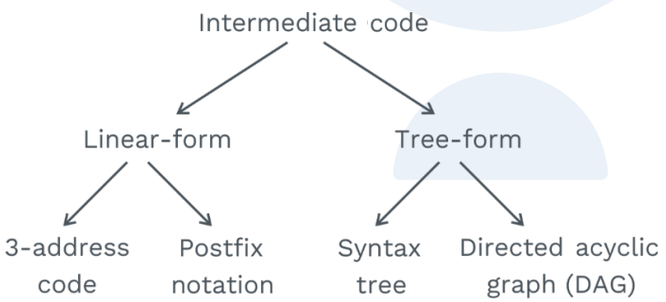


Fig. 5.1 Representation of Intermediate Code

3-Address code:

- Three address code is built from two concepts: address and instructions.
- The implementation of three address codes consists of records as well the address fields
- Three address instructions consist of at most three addresses in a line of code.
- An address can be one of the following:
  1) A name: For convenience, we allow the source program name to appear as in the 3-address code.
  2) A constant: Constants and variables are the mandatory fields with which compiler has to deal with
  3) A compiler-generated temporary variable: Temporary variable is mainly for optimising compiler to store intermediate results each time a temporary variable is needed.

- **Types of 3-address code:**

i)	x = y op z	|| x stores the result of y operator z
ii)	x = op y	|| x stores result of unary operator y
iii)	x = y	|| x stores y
iv)	x = a [i]	|| x stores value of array a at index i
v)	a [i] = x	|| Stores the value of x in an array a at index i
vi)	x = f(a)	|| x stores the value return by function 'f' with parameter 'a'
vii)	if x < y goto z	{conditional goto}
viii)	goto x	{Unconditional goto}
ix)	z = x[i] [j]	|| This is not 3-address code representation because it uses variables z, x, i, j
x)	x = f(a, b)	|| This is not 3-address code representation because it uses 4 variables x, f, a, b
xi)	x = &a	|| address assignment

Types of 3-address code representation:

Basically, there are 3-ways of representing 3-address code

1) Quadruple
2) Triple
3) Indirect triple

Let us consider an expression:

$$a = b * - c + b * - c ;$$

Three address code:

$$t_1 = \text{minus } c$$

$$t_2 = b * t_1$$

$$t_3 = \text{minus } c$$

$$t_4 = b * t_3$$

$$t_5 = t_2 + t_4$$

$$a = t_5$$

1) Quadruples representation of 3-address code:

	Op	arg_1	arg_2	result
0.	minus	c	–	t_1
1.	*	b	t_1	t_2
2.	minus	c	–	t_3
3.	*	b	t_3	t_4
4.	+	t_2	t_4	t_5
5.	=	t_5	–	a

Fig. 5.2 Quadruples Representation of 3-Address Code

For readability, we use actual identifiers like a, b, and c in the field arg_1, arg_2 and result instead of pointers to their symbol table entries.

- Op field contains the internal code for the operator.
- arg_1 and arg_2 contain the operand.
- result field stores the result.

2) Triplet representation of 3-address code:

- A triplet has only tree fields Op, arg_1, arg_2
- Using triplet, we refer to the result of an operation x op y by its position rather than an explicit temporary name.
- Above 3-address code representation in triplet.

	Op	arg_1	arg_2
0.	minus	c	–
1.	*	b	(0)
2.	minus	c	–
3.	*	b	(2)
4.	+	(1)	(3)
5.	=	a	(4)

Fig. 5.3 Triplet Representation of 3-Address Code

Triple representation of a = b * – c + b * – c ;

3) Indirect triple representation of 3-address code:

- Indirect triples do not list the triples, but they list the pointers of triples. For eg: an array of instructions to list pointers to triples in the desired order.
- An optimising compiler is able to reorder the instruction in triples without affecting the triples themselves.

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

	Op	arg_1	arg_2
0.	minus	c	–
1.	*	b	(0)
2.	minus	c	–
3.	*	b	(2)
4.	+	(1)	(3)
5.	=	a	(4)

Fig. 5.4 (Indirect Triple Representation of 3-Address Code)

Indirect-triple representation of a = b * (–c) + b * (–c) ;

- Instruction is an array that contains a pointer to triple in the desired order.

Rack Your Brain

What could be the benefit of quadruples over triple?

SOLVED EXAMPLES

Q.1 **Write 3-address code for the following program**

```
x = 2 ;
for (x = 25 ; x ≤ 100 ; x ++)
{
   x = p + q + r ;
}
```

Sol:

1000: x = 2

1001: x = 25

1002: if x ≤ 100 goto 1004.

1003: goto 1009
1004: $t_1 = p + q$
1005: $t_2 = t_1 + r$
1006: $x = t_2$
1007: $x = x + 1$
1008: goto 1002
1009: _____

Q2 **3-address code representation in quadruple form for the expression**
x = b + b + b + b + b

Sol: 3-address code representation:

$t_1 = b + b$
$t_2 = t_1 + b$
$t_3 = t_2 + b$
$t_4 = t_3 + b$
$x = t_4$

Quadruple form representation of 3-address code:

	Op	arg_1	arg_2	result
(0)	+	b	b	t_1
(1)	+	t_1	b	t_2
(2)	+	t_2	b	t_3
(3)	+	t_3	b	t_4
(4)	=	t_4	–	x

Fig. 5.5 (Quadruple Representation)

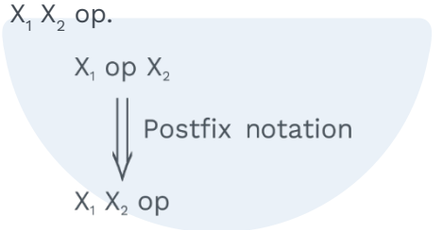
Q3 **3-address code representation for expression**
x = (a + b) * a * b * (c + d) ;

Sol: 3-address code for given expression

$t_1 = a + b$
$t_2 = t_1 * a$
$t_3 = t_2 * b$
$t_4 = c + d$
$t_5 = t_3 * t_4$
$x = t_5$

Postfix notation:

- Let us consider there are two expressions X_1 and X_2 and there is an operand op which are represented as X_1 op X_2 then in postfix notation it will be represented as X_1 X_2 op.



SOLVED EXAMPLES

Q4 **Convert the following expression in postfix notation.**

x = a * b + c * d + d * e

where * has higher precedence, and both are left-associative, and '=' has the least precedence.

Sol: * has higher precedence thus they will be calculated first.

∴ (x) = (a*b) + (c*d) + (d*e)
(x) = (ab*) + (cd*) + (de*)
(x) = ((ab*cd*)+) + (de*)
(x) = (ab*cd*+de*+)
xab*cd* + de* + =

Q5 **Given an expression**
x $ y $ z # x $ b # t ⊕ d
Convert the expression in postfix notation when the precedence order of the operator is $ > # > ⊕ and all the three operators are left-associative.

Sol: $ has highest precedence and ⊕ has the least precedence.

∴ (x y $) $ z # x $ b # t ⊕ d

⇒ (x y $ z $) # (x $ b) # t ⊕ d

⇒ (x y $ z $) # (x b $) # t ⊕ d

⇒ (x y $ z $ x b $ #) # t ⊕ d

⇒ (x y $ z $ x b $ # t #) ⊕ d

⇒ x y $ z $ x b $ # t # d ⊕

Post order notation ⇒ x y $ z $ x b $ # t # d ⊕

Q6 **x = a + uminus b+ uminus d**
All the operators have these usual precedences and associativity then convert given expression in postfix notation uminus represent uniary minus operator.

Sol: As we know, uniary operator has the highest precedence

∴ Order of precedence uminus > + > =

Postfix notation: x = a +(b uminus) + (d uminus)

x = (a b uminus +) + (d uminus)

x = (a b uminus + d uminus +)

x a b uminus + d uminus + =

Syntax tree:

- Syntax tree is the representation in the form of tree in which each internal node represents the operators, and the leaf node represents the operand.
- During the formation of a syntax tree of any code, we should design tree in such a form that lower precedence operator should be on the top level and higher precedence should be on lower levels.

SOLVED EXAMPLES

Q7 **x = a * b + c**
Syntax tree for the above expression when all the operators have their usual associativity and precedence.

Sol: The Precedence order of operators:
* > + > =
Syntax Tree:

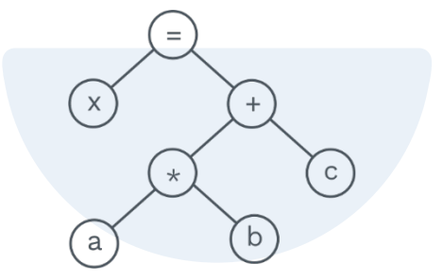


Fig. 5.6 Syntax Tree

Q8 **x = a * b ÷ c * d**
Syntax tree for the above expression when operators have their usual precedence and associativity.

Sol: Both '*' and '÷' have the same precedence. Thus, syntax tree will be formed on the basis of associativity.
Syntax Tree:

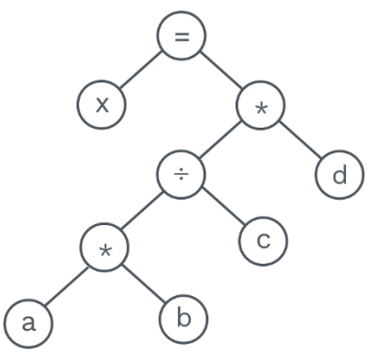


Fig. 5.7 Syntax Tree

Directed acyclic graphs:

- It is similar to a syntax tree except that common sub-expression are represented by a single node.
- Common subexpression is represented with a single node to get the optimised code, i.e., instead of evaluating again and again, we can evaluate common expression only once.

SOLVED EXAMPLES

Q9 **x = t + t + t + t**
For the given expression form DAG.

Sol: Here identifier 't' is used many times in the expression, and 't+t' is evaluated twice so in the graph, we will use only one node for the identifier 't' and 't+t' evaluation.

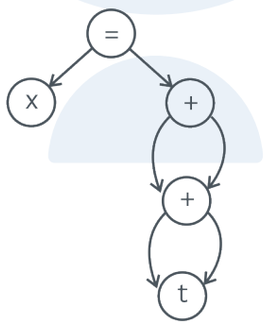


Fig. 5.8 DAG

Q10 **x = a * b * c +b * a**
All operator have their usual precedence and associativity, then form the DAG for the above expression.

Sol: Here (a * b) can be used again as it is repeated in the expression there again forming the node for (a * b), we have reused it.

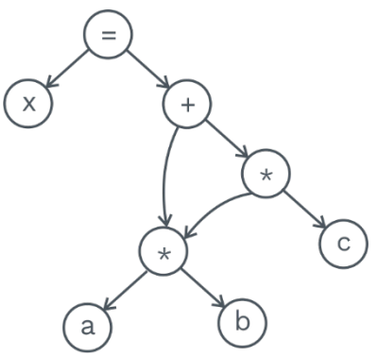


Fig. 5.9 DAG

5.3 STATIC SINGLE ASSIGNMENT

- Static Single Assignment (SSA) is the property of intermediate code representation where each variable should be assigned exactly once and each variable should be defined before its use.
- Each expression must be in 3-address representation only and following the above two conditions.

SOLVED EXAMPLES

Q11 **Check whether given 3-address code in SSA representation or not.**

$T_1 = a * a$
$T_2 = y * a$
$T_2 = z * b$

Sol: Given 3-address code is not in SSA representation because variable T_2 is assigned more than one time.

Q12 **Check whether the given code is in SSA representation or not.**

$T_1 = a * b$
$T_2 = c * T_1$
$T_3 = d + T_2$
$T_4 = T_1 + T_2$

Sol: Given 3-address code is in SSA representation because each variable has been assigned exactly once.

Q13 **Write SSA representation of given expression**
x = b * c + d + a + b

Sol: '*' is computed first because multiplication has higher precedence than addition.

$T_1 = b * c$
$T_2 = T_1 + d$
$T_3 = T_2 + a$
$T_4 = T_3 + b$
$x = T_4$

Q14 **Given a three address code**
x = a + b
y = c + d
x = y + d
c = c + b
z = c + y
Convert the given 3-address code into a single static assignment.

Sol: Here in this 3-address code representation, variables x and c are assigned twice; thus, it is not in SSA.

SSA representation of above code:

$x = a + b$
$y = c + d$
$T_1 = y + d$
$T_2 = c + b$
$z = T_2 + y$

Previous Years' Question

Consider the following code segment

$$x = u - t;$$
$$y = x * v;$$
$$x = y + w;$$
$$y = t - z;$$
$$y = x * y;$$

Minimum number of variables required to convert the above code in single static assignment.

Sol: 10 variables **[GATE: CS-2016]**

Previous Years' Question

Consider the following intermediate program in three address code:

$$p = a - b$$
$$q = p * c$$
$$p = u * v$$
$$q = p + q$$

Which of the following corresponds to a static single assignment from the above code?

a)
$p_1 = a - b$
$q_1 = p_1 * c$
$p_1 = u * v$
$q_1 = p_1 + q_1$

b)
$p_3 = a - b$
$q_4 = p_3 * c$
$p_4 = u * v$
$q_5 = p_4 + q_4$

c)
$p_1 = a - b$
$q_1 = p_2 * c$
$p_2 = u * v$
$q_2 = p_4 + q_3$

d)
$p_1 = a - b$
$q_1 = p * c$
$p_2 = u * v$
$q_2 = p + q$

Sol: b) **[GATE: CS-2016]**

Previous Years' Question

Consider the basic block given below:

a = b + c

c = a + d

d = b + c

e = d – b

a = e + b

The minimum number of nodes and edges present in the DAG representation of the above basic block respectively are.

a) 6 and 6 **b)** 8 and 10

c) 9 and 12 **d)** 4 and 4

Sol: a) **[GATE: CS-2014]**

5.4 CONTROL FLOW GRAPHS

Backpatching:

There is a problem when writing the code for the flow of control statements that match the jump instruction with the target of the jump.

The solution to this problem is back patching, in which when we don't know the target of the jump statement, we leave it unspecified.

When we know the target address, then we fill that unspecified blank space.

For eg:

100: if a < 100 goto ____
101: goto ____
102: if a > 200 goto 104
103: goto ____
104: if a ! = b goto ____
105: goto ____

After backpatching 104 goes to instruction 102.

100: if a < 100 goto ____
101: goto 102
102: if a > 200 goto 104
103: goto ____

104: if a ! = b goto ____
105: goto ____

After backpatching 102 into instruction 101.

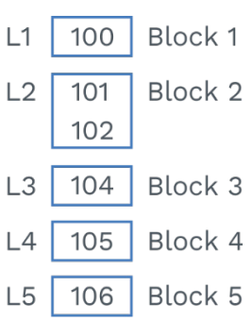
Leader:
1) The first statement of the intermediate code is the leader.
2) Target statement of the goto is the leader.
3) Next statement after goto is the leader.

Basic block:
- Basic block is the set of statements of intermediate code which consist of one entry point and one exit point in between which there can be no halt or jump.
- Each basic block consists of one leader.
- Basic block starts from the leader statement and ends one statement before the next leader.
- Number of Basic blocks = Number of leaders.

SOLVED EXAMPLES

Q15 **Draw the control flow graph of the given code.**
100: if x < 100 goto 104
101: x = a + b ;
102: x = x + 7 ;
104: if x > 200 goto 106
105: x = x + 1
106: z = x + 200

Sol:



Control flow graph:

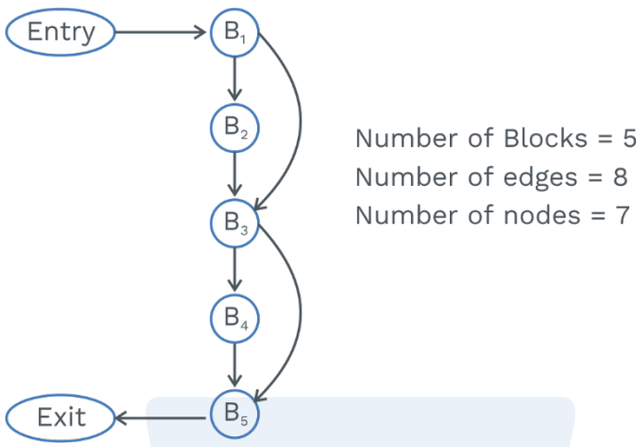


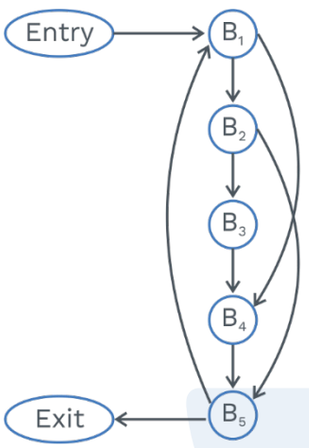
Fig. 5.10 Control Flow Graph

Q16 **100: i = 1 ;**
101: j = 2 ;
102: if i > 100 goto 106
103: i = i + 1 ;
104: if j > 100 goto 107
105: j = j + 1 ;
106: z = z + 200
107: z = z + 200
108: if z < 500 goto 101
The total number of nodes and edges will be there in the control flow graph of above code.

Sol:

L1	100 101 102	Block 1
L2	103 104	Block 2
L3	105	Block 3
L4	106	Block 4
L5	107 108	Block 5

Control flow graph:



Number of nodes = 7
Number of edges = 9

Fig. 5.11 Control Flow Graph

∴ Total number of nodes and edges = 7 + 9 = 16

Previous Years' Question

Consider the intermediated code given below:

1) i = 1
2) j = 1
3) $t_1 = 5 * i$
4) $t_2 = t_1 + j$
5) $t_3 = 4 * t_2$
6) $t_4 = t_3$
7) $a[t_4] = -1$
8) j = j + 1
9) if j < = 5 goto (3)
10) i = i + 1
11) if i < 5 goto (2)

The number of nodes and edges in the control-flow graph constructed for the above code respectively are

a) 5 and 7 **b)** 5 and 5 **c)** 6 and 7 **d)** 7 and 8

Sol: c)

Chapter Summary

- Intermediate code is used to translate the source code into the machine code. It lies between high level language and machine language.
- Intermediate code is present in two forms:
  - **i)** Linear form:
    - **a)** 3-address code
    - **b)** Postfix notation
  - **ii)** Tree form:
    - **a)** Syntax tree
    - **b)** Directed Acyclic Graph (DAG)
- 3-address code can have a maximum of 3-address in its expression and is built on two concept address and instruction.
- 3-address codes are of the following types:
  i) Quadruples
  ii) Tripple
  iii) Indirect tripple
- Postfix notation is the conversion of expression in the postfix form.
- Syntax tree represented in the form of tree where internal nodes represent operators and leaf nodes represent operands.
- Static single assignment (SSA) is the representation of the code in which each variable can only be assigned once, and the variable should be defined before its use.