Unit-4 Intermediate Code generation

1. What is intermediate code?

Ans: During the translation of a source program into the object code for a target machine, a compiler may generate a middle-level language code, which is known as intermediate code or intermediate text. The complexity of this code lies between the source language code and the object code. The intermediate code can be represented in the form of postfix notation, syntax tree, directed acyclic graph (DAG), three-address code, quadruples, and triples.

2. Write down the benefits of using an intermediate code generation over direct code generation?

Ans: The benefits of using an intermediate code over direct code generation are as follows:

- Intermediate code is machine independent, which makes it easy to retarget the compiler to generate code for newer and different processors.
- Intermediate code is nearer to the target machine as compared to the source language so it is easier to generate the object code.
- The intermediate code allows the machine-independent optimization of the code. Several specialized techniques are used to optimize the intermediate code by the front end of the compiler.
- Syntax-directed translation implements the intermediate code generation; thus, by augmenting the parser, it can be folded into the parsing.

3. What are the two representations to express intermediate languages?

Ans: The two representations of intermediate languages are categorized as follows:

1. **High-level intermediate representation:** This representation is closer to the source program. Thus, it represents the high-level structure of a program, that is, it depicts the natural hierarchical structure of the source program. The examples of this representation are directed acyclic graphs (DAG) and syntax trees. This representation is suitable for static type checking task. The critical features of high-level representation are given as follows:

    - It retains the program structure as it is nearer to the source program.
    - It can be constructed easily from the source program.
    - It is not possible to break the source program to extract the levels of code sharing due to which the code optimization in this representation becomes a bit complex.

2. **Low-level intermediate representation:** This representation is closer to the target machine where it represents the low-level structure of a program. It is appropriate for machine-dependent tasks like register allocation and instruction selection. A typical example of this representation is three-address code. The critical features of low-level representation are given as follows:

   - It is near to the target machine.
   - It makes easier to generate the object code.
   - High effort is required by the source program to generate the low-level representation.

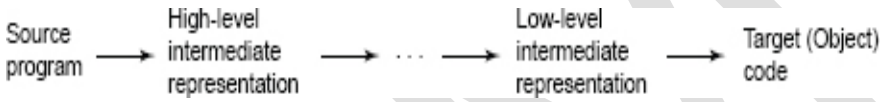


Figure 4.1 A Sequence of Intermediate Representation

4. What is postfix notation? Explain with example.

Ans: Generally, we use infix notation to represent an arithmetic expression such as multiplication of two operands a and b. In infix notation, operator is always placed between the two operands, as a * b. But in postfix notation (also known as reverse polish or suffix notation), the operator is shifted to the right end, as ab*. In postfix notation, parentheses are not required because the position and the number of arguments of the operator allow only a single way of decoding the postfix expression. The postfix notation can be applied to k-ary operators for any k > 1. If β is a k-ary operator and a 1, a2,…, ak are any postfix expressions, then after applying β to the expressions, the expression in postfix notation is represented as a1 a2…akβ.

For example, consider the following infix expressions and their corresponding postfix notations:

- (l + m) * n is an infix expression, the postfix notation will be l m + n *.
- p * (q + r) is an infix expression, the postfix expression will be p q r + *.
- (p - q) * (r + s) + (p - q) is an infix expression, the postfix expression will be p q - r s + * p q - +.

5. Convert the following expression to the postfix notation and evaluate it.

P + (–Q + R * S)

Ans: The postfix notation for the given expression is:

PQ - RS * ++

The step-by-step evaluation of this postfix expression is shown in Figure 4.2.

S. no.	String and scan symbol	Previous stack content	Rule in use	New stack content
1.	PQ – RS * ++			
2.	P		Rule 1	P
3.	Q	P	Rule 1	PQ
4.	–	PQ	Rule 3	P(–Q)
5.	R	P(–Q)	Rule 1	P(–Q)R
6.	S	P(–Q)R	Rule 1	P(–Q)RS
7.	*	P(–Q)RS	Rule 2	P(–Q)(R * S)
8.	+	P(–Q)(R * S)	Rule 2	P(–Q + R * S)
9.	+	P(–Q + R * S)	Rule 2	P +(–Q + R * S)

Figure 4.2 Evaluation of Postfix Expression PQ – RS * + +

The desired result is P +(–Q + R * S).

7. What is a three-address code? What are its types? How it is implemented?

Ans: A string of the form X: = Y OP Z, in which op is a binary operator, Y and Z are the addresses of the operands, and X is the address of the result of the operation, is known as three-address statement. The operator op can be a fixed or floating-point arithmetic operator, or a logical operator. X, Y, and Z can be considered either as constants or as predefined names by the programmer or temporary names generated by the compiler. This statement is named as the "three-address statement" because of the usage of three addresses, one for the result and two for the operands. The sequence of such three-address statements is known as three-address code. The complicated arithmetic expressions are not allowed in three-address code because only a single operation is allowed per statement. For example, consider an expression A + B * C, this expression contains more than one operator so the representation of this expression in a single three-address statement is not possible. Hence, the three-address code of the given expression is as follows:

T1: = B * C

T2: = A + T1

where, T1 and T2 are the temporary names generated by the compiler.

TYPES OF THREE-ADDRESS STATEMENTS:

There are some cases where a statement consists of less than three addresses and is still known as three- address statement. Hence, the different forms of three-address statements are given as follows:

★ **Assignment statements:** These statements can be represented in the following forms:

- X: = Y op Z, where op is any logical/arithmetic binary operator.

- X: = op Y, where op is an unary operator such as logical negation, conversion operators, and shift operators.
- X: = Y, where the value of Y is assigned to operand X.

★ **Indexed assignment statements:** These statements can be represented in the following forms:

- X: = Y[I]
- X[I]: = Y, where X, Y and I refer to the data objects and are represented by pointers to the symbol table.

★ **Address and pointer assignment statements:** These statements can be represented in the following forms:

- X: = addr Y defines that X is assigned the address of Y.
- X: = *Y defines that X is assigned the content of location pointed to by Y.
- *X: = Y sets the r-value of the object pointed to by X to the r-value of Y.

★ **Jump statements:** Jump statements are of two types-conditional and unconditional that works with relational operators and are represented in the following forms:

- The unconditional jump is represented as goto L, where L being a label. This instruction means that the Lth three-address statement is the next to be executed.
- The conditional jumps such as if X relop Y goto L, where relop signifies the relational operator (≤, =, >, etc.) applied between X and Y. This instruction implies that if the result of the expression X relop Y is true then the statement labeled L is executed. Otherwise, the statement immediately following the if X relop Y goto L is executed.

★ **Procedure call/return statements:** These statements can be defined in the following forms:

- param X and call P, n, where they are represented and typically used in the three- address statement as follows:

param X1

param X2

.

.

.

param Xn

call P, n

Here, the sequence of three-address statements is generated as a part of call of the procedure P(X1, X2,…, Xn), and n in call P, n is defined as an integer specifying the total number of actual parameters in the call.

- Y = call p, n represents the function call.
- return Y, represents the return statement, where Y is a returned value.

IMPLEMENTATION OF THREE-ADDRESS STATEMENTS:

The three-address statement is an abstract form of intermediate code. Hence, the actual implementation of the three-address statements can be done in the following ways:

- Quadruples
- Triples
- Indirect triples

8. Explain quadruples with the help of a suitable example.

Ans: Quadruple is defined as a record structure used to represent a three-address statement. It consists of four fields. The first field contains the operator, the second and third fields contain the operand 1 and operand 2, respectively, and the last field contains the result of that three-address statement. For better understanding of quadruple representation of any statement, consider a statement, S = -z/a * (x + y), where -z stands for unary minus z.

To represent this statement into quadruple representation, we first construct the three-address code as follows:

t_1: = x + y
t_2 = a * t_1
t_3 : = - z
t_4 : = t_3/t_2
S: = t_4

The quadruple representation of this three-address code is shown in Figure 4.3.

	Operator	Operand 1	Operand 2	Result
0	+	x	Y	t_1
1	*	a	t_1	t_2
2	-	Z		t_3
3	/	t_3	t_2	t_4
4	: =	t_4		S

Figure 4.3 Quadruple Representation for S = -z/a * (x + y)

9. Define triples and indirect triples. Give suitable examples for each.

Ans: Triples: A triple is also defined as a record structure that is used to represent a three-address statement. In triples, for representing any three-address statement three fields are used, namely, operator, operand 1 and operand 2, where operand 1 and operand 2 are pointers to either symbol table or they are pointers to the records (for temporary variables) within the triple representation itself. In this representation, the result field is removed to eliminate the use of temporary names referring to symbol table entries. Instead, we refer the results by their positions. The pointers to the triple structure are represented by parenthesized numbers, whereas the symbol-table pointers are represented by the names themselves. The triples representation of the expression is shown in Figure 4.4.

	Operator	Operand 1	Operand 2
0	+	x	Y
1	*	a	(0)
2	-	z	
3	/	(2)	(1)
4	:=	S	(3)

Figure 4.4 Triple Representation for S = -z/a * (x + y)

In triple representation, the ternary operations X[I] : = Y and X : = Y[I] are represented by using two entries in the triple structure as shown in Figure 4.5(a) and (b) respectively. For the operation X[I] : = Y, the names X and I are put in one triple, and Y is put in another triple. Similarly, for the operation X : = Y[I], we can write two instructions, t: = Y[I], and X: = t. Note that instead of referring the temporary t by its name, we refer it by its position in the triple.

Indirect triples: An indirect triple representation consists of an additional array that contains the pointers to the triples in the desired order. Let us define an array A that contains pointers to triples in desired order. Indirect triple representation for the statement S given in the previous question is shown in Figure 4.6.

	Operator	Operand 1	Operand 2
0	[] =	X	I
1		Y	

(a) Triple Representation of X[I] : = Y

	Operator	Operand 1	Operand 2
0	= []	Y	I
1	: =	(0)	X

(b) Triple Representation of X : = Y[I]

Figure 4.5 More Triple Representations

	A
101	(0)
102	(1)
103	(2)
104	(3)
105	(4)

	Operator	Operand 1	Operand 2
0	+	x	y
1	*	a	(0)
2	-	z	
3	/	(2)	(1)
4	:=	S	(3)

Figure 4.6 Indirect Triples Representation of S = -z/a * (x + y)

The main advantage of indirect triple representation is that an optimizing compiler can move an instruction by simply reordering the array A, without affecting the triples themselves.

19. Translate the expression X = - (a + b) * (c + d) + (a + b + c) into quadruples and triples.
Ans: The three-address code for the given expression is given below:

$t_1 := a + b$
$t_2 := -t_1$
$t_3 := c + d$
$t_4 := t_2 * t_3$
$t_5 := t_1 + c$
$t_6 := t_4 + t_5$
$X := t_6$

The quadruple representation is shown in Figure 4.7.

	Operator	Operand 1	Operand 2	Result
0	+	a	b	t_1
1	-	t_1		t_2
2	+	c	d	t_3
3	*	t_2	t_3	t_4
4	+	t_1	c	t_5
5	+	t_4	t_5	t_6
6	=	t_6		

Figure 4.7 Quadruple Representation for X = -(a + b) * (c + d) + (a + b + c)

The triple representation for the given expression is shown in Figure 4.8.

	Operator	Operand 1	Operand 2
0	+	a	b
1	-	(0)	
2	+	c	d
3	*	(1)	(2)
4	+	(0)	c
5	+	(3)	(4)
6	=	(5)	

Figure 4.8 Triple Representation for X = -(a + b) * (c + d) + (a + b + c)

20. Generate the three-address code for the following program segment.

```
main( )
{
    int k = 1;
    int a[5];
    while (k <= 5)
  {
    a[k] = 0;
    k++;
  }
}
```

Ans: The three-address code for the given program segment is given below:

1. k: = 1
2. if k <= 5 goto (4)
3. goto (8)
4. t_1: = k * width
5. t_2: = addr(a)-width
6. $t_2[t_1]$: = 0
7. t_3 : = k + 1
8. k: = t_3
9. goto (2)
10. Next

21. Generate the three-address code for the following program segment

```
while(x < z and y > s) do
if x = 1 then
    z = z + 1
else
    while x <= s do
        x = x + 10;
```

Ans: The three-address code for the given program segment is given below:

1. if x < z goto (3)
2. goto (16)
3. if y > s goto (5)
4. goto (16)
5. if x = 1 goto (7)
6. goto (10)
7. t_1: = z + 1
8. z: = t_1
9. goto (1)
10. if x <= s goto (12)
11. goto (1)
12. t_2: = x + 10
13. x: = t_2
14. goto (10)
15. goto (1)
16. Next

21. Generate the three-address code for the following program segment

```
while(x < z and y > s) do
if x = 1 then
    z = z + 1
else
    while x <= s do
        x = x + 10;
```

Ans: The three-address code for the given program segment is given below:

1. if x < z goto (3)
2. goto (16)
3. if y > s goto (5)
4. goto (16)
5. if x = 1 goto (7)
6. goto (10)
7. t_1: = z + 1
8. z: = t_1
9. goto (1)

10. if x <= s goto (12)
11. goto (1)
12. t_2: = x + 10
13. x: = t_2
14. goto (10)
15. goto (1)
16. Next

22. Consider the following code segment and generate the three-address code for it.
for (k = 1; k <= 12; k++)
if x < y then a = b + c;

Ans: The three-address code for the given program segment is given below:

1. k: = 1
2. if k <= 12 goto (4)
3. goto (11)
4. if x < y goto (6)
5. goto (8)
6. t_1: = b + c
7. a: = t_1
8. t_2: = k + 1
9. k: = t_2
10. goto (2)
11. Next

23. Translate the following statement, which alters the flow of control of expressions, and generate the three-address code for it.

while(P < Q)do
if(R < S) then a = b + c;

Ans: The three-address code for the given statement is as follows:

1. if P < Q goto (3)
2. goto (8)
3. if R < S goto (5)
4. goto (1)
5. t_1: = b + c
6. a: = t_1
7. goto (1)
8. Next

24. Generate the three-address code for the following program segment where, x and y are arrays of size 10 * 10, and there are 4 bytes/word.

begin
add = 0
a = 1
b = 1
do

```
        begin
          add = add + x[a,b] * y[a,b]
          a = a + 1
          b = b + 1
        end
      while a <= 10 and b <= 10
    end
```

Ans: The three-address code for the given program segment is given below:

1. add: = 0
2. a: = 1
3. b: = 1
4. t_1: = a * 10
5. t_1: = t_1 + b
6. t_1: = t_1 * 4
7. t_2: = addr(x) - 44
8. t_3: = t_2 [t_1]
9. t_4: = b * 10
10. t_4: = t_4 + a
11. t_4: = t_4 * 4
12. t_5: = addr(y) - 44
13. t_6: = t_5[t_4]
14. t_7: = t_3 * t_6
15. t_7: = add + t_7
16. t_8: = a + 1
17. a: = t_8
18. t_9: = b + 1
19. b: = t_9
20. if a <= 10 goto (22)
21. goto (23)
22. if b <= 10 goto(4)
23. Next

25. Translate the following program segment into three-address statements:

```
switch(a + b)
{
  case 2: {x = y; break;}
  case 5: switch x
          {
            case 0: {a = b + 1; break;}
            case 1: {a = b + 3; break;}
            default: {a = 2; break;}
          }
   break;
```

case 9: {x = y - 1; break;}
default: {a = 2; break;}
}

Ans: The three-address code for the given program segment is given below:

1. t_1: = a + b
2. goto (23)
3. x: = y
4. goto (27)
5. goto (14)
6. t_3: = b + 1
7. a: = t_3
8. goto (27)
9. t_4: = b + 3
10. a: = t_4
11. goto (27)
12. a: = 2
13. goto (27)
14. if x = 0 goto (6)
15. if x = 1 goto (9)
16. goto (12)
17. goto (27)
18. t_5: = y - 1
19. a: = t_5
20. goto (27)
21. a: = 2
22. goto (27)
23. if t = 2 goto (3)
24. if t = 5 goto (5)
25. if t = 9 goto (18)
26. goto (21)
27. Next

MULTIPLE-CHOICE QUESTIONS

1. Which of the following is not true for the intermediate code?
(a) It can be represented as postfix notation.
(b) It can be represented as syntax tree, and or a DAG.
(c) It can be represented as target code.
(d) It can be represented as three-address code, quadruples, and triples.
2. Which of the following is true for intermediate code generation?
(a) It is machine dependent.

(b) It is nearer to the target machine.
(c) Both (a) and (b)
(d) None of these

3. Which of the following is true in the context of high-level representation of intermediate languages?
(a) It is suitable for static type checking.
(b) It does not depict the natural hierarchical structure of the source program.
(c) It is nearer to the target program.
(d) All of these

4. Which of the following is true for the low-level representation of intermediate languages?
(a) It requires very few efforts by the source program to generate the low-level representation.
(b) It is appropriate for machine-dependent tasks like register allocation and instruction selection.
(c) It does not depict the natural hierarchical structure of the source program.
(d) All of these

5. The reverse polish notation or suffix notation is also known as _____ .
(a) Infix notation
(b) Prefix notation
(c) Postfix notation
(d) None of above

6. In a two-dimensional array A[i][j], where i is a element of width w_1 and j is of width w_2, the relative address of A[i][j] can be calculated by the formula .
(a) $i * w_1 + j * w_2$
(b) $base + i * w_1 + j * w_2$
(c) $base + i * w_1 + j * w_1$
(d) $base + (i + j) * (w_1 + w_2)$

ANSWERS

1.(c) 2. (c) 3. (a) 4. (b) 5. (c) 6. (b)

Q.1 What is left recursion? Remove left recursion from following grammar.

$$S \rightarrow (L) / a$$
$$L \rightarrow L, S / S$$

ns → Left Recursion Defn -

$$S \rightarrow (L) / a$$
$$L \rightarrow L, S / S$$
$$[A \rightarrow A\alpha / \beta]$$

Rule:- ~~A~~ → If $A \rightarrow A\alpha / \beta$ then

$$A \rightarrow \beta A'$$
$$A' \rightarrow \alpha A' / \epsilon$$

Hence

$$S \rightarrow (L) / a$$
$$L \rightarrow S L'$$
$$L' \rightarrow , S L' / \epsilon$$

2

Q.2 Construct two diff parse tree for string abaabb

G(S) ∫: S → aSbS | bSaS | ∈

Ans → Parse Tree ↓→

S → aSbS

→ abSaSbS

→ abaSbS

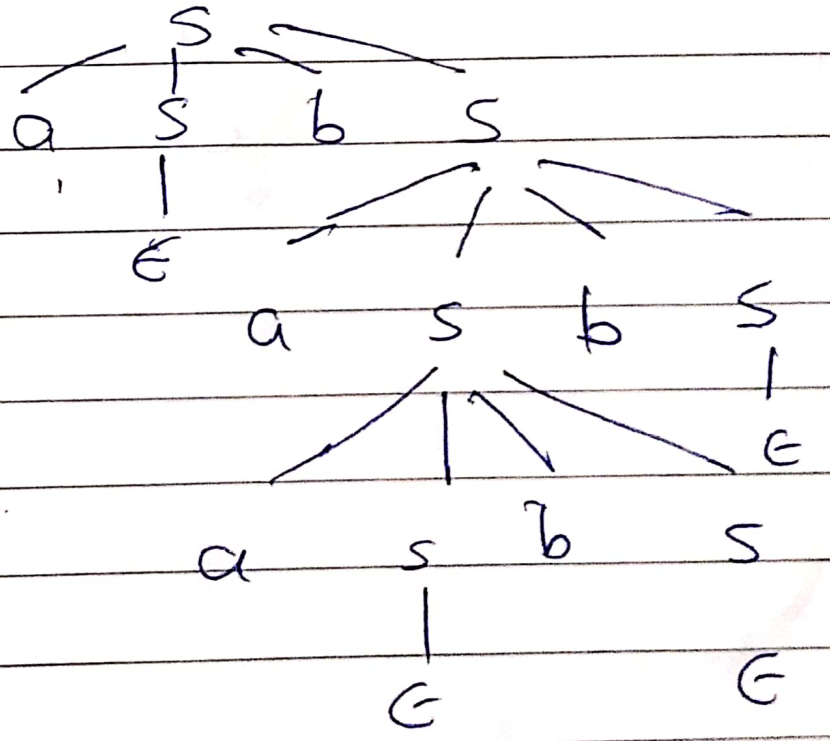
S → aSbS

→ abS

→ abaSbS

→ abaaSbSbS

→ abaabb



Parse Tree 2 →

$S \rightarrow aSbS$

$\rightarrow abSaSbS$

$\rightarrow abaSbS$

$\rightarrow abaabS$

$\rightarrow abaabb$