

Department of Computer Science & Engineering



PUNJABI UNIVERSITY, PATIALA

Compiler Design

Sec-B

Part-1

Contents

Intermediate Code Generation	2
Type Checking	16
Runtime Administration.....	21
Symbol Table.....	28
Code Optimization and Code Generation	37

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:

❑ **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.

❑ **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.

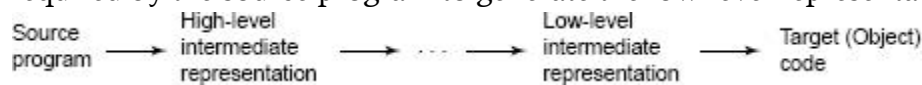


Figure 7.1 A Sequence of Intermediate Representation

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, a_2, \dots, a_k are any postfix expressions, then after applying β to the expressions, the expression in postfix notation is represented as $a_1 a_2 \dots a_k \beta$.

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. Explain the process of evaluation of postfix expressions.

Ans: The postfix notations can easily be evaluated by using a stack, and generally the evaluation process scans the postfix code left to right.

1. If the scan symbol is an operand, then it is pushed onto the stack, and the scanning is continued.
2. If the scan symbol is a binary operator, then the two topmost operands are popped from the stack. The operator is applied to these operands, and the result is pushed back to the stack.
3. If the scan symbol is a unary operator, it is applied to the top of the stack and the result is pushed back onto the stack.

Note: The result of a unary operator can be shown within parenthesis. For example, $(-X)$.

6. 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 7.2.

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 7.2 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 $X := Y \text{ OP } Z$, 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:

$T_1 := B * C$

$T_2 := A + T_1$

where, T_1 and T_2 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:

- $X := Y \text{ op } Z$, where op is any logical/arithmetic binary operator.
- $X := \text{op } Y$, 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:

- $X := \text{addr } Y$ 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 L^{th} 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 ($\leq, =, >$, etc.) applied between X and Y . This instruction implies that if the result of the expression $X \text{ relop } Y$ 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 $P(X_1, X_2, \dots, X_n)$, 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:

```

t1 := x + y
t2 = a * t1
t3 := - z
t4 := t3/t2
S := t4

```

The quadruple representation of this three-address code is shown in Figure 7.3.

	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 7.3 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 (in Question 7) is shown in Figure 7.4.

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

Figure 7.4 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 7.5(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 7.6.

	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 7.5 More Triple Representations

	A		Operator	Operand 1	Operand 2
101	(0)	0	+	x	y
102	(1)	1	*	a	(0)
103	(2)	2	-	z	
104	(3)	3	/	(2)	(1)
105	(4)	4	:=	S	(3)

Figure 7.6 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.

10. Explain Boolean expressions. What are the different methods available to translate Boolean expression?

Ans: Boolean operators, like AND(&&), OR(| |) and NOT(!), play an important role in constructing the Boolean expressions. These operators are applied to either relational expressions or Boolean variables. In programming languages, Boolean expressions serve two main functions, given as follows:

□ Boolean expressions can be used as conditional expressions in the statements that alter the flow of control, such as in while- or if-then-else statements.

□ Boolean expressions are also used to compute the logical values.

Boolean expressions can be generated by using the following grammar:

$S \rightarrow S \text{ or } S \mid S \text{ and } S \mid \text{not } S \mid (S) \mid \text{id} \mid \text{id rel op id} \mid \text{true} \mid \text{false}$

where the attribute rel op is used to indicate any of $<$, \leq , $=$, \neq , $>$, \geq . Generally, we consider that the operators **or** and **and** are left-associative, and that the operator **not** has the highest precedence, then **and**, and then **or**.

Methods of Translating a Boolean Expression into Three-Address Code:

There are two methods available to translate a Boolean expression into three-address code, as given below:

□ **Numerical representation:** The first method of translating Boolean expression into three-address code comprises encoding true and false numerically and then evaluating the Boolean expression

similar to an arithmetic expression. True is often denoted by 1 and false by 0. Some other encodings are also possible where any non-zero or non-negative quantity indicates true and any negative or zero number indicates false. Expressions will be calculated from left to right like arithmetic expressions. For example, consider a Boolean expression X and Y or Z , the translation of this expression into three-address code is as follows:

$t_1 := X$ and Y

$t_2 := t_1$ or Z

Now, consider a relational expression if $X > Y$ then 1 else 0, the three-address code translation for this expression is as follows:

1. if $X > Y$ goto (4)
2. $t_1 := 0$
3. goto (5)
4. $V = 1$
5. Next

Here, t_1 is a temporary variable that can have the value 1 or 0 depending on whether the condition is evaluated to true or false. The label Next represents the statement immediately following the else part.

□ **Control-flow representation:** In the second method, the Boolean expression is translated into three-address code based on the flow of control. In this method, the value of a Boolean expression is represented by a position reached in a program. In case of evaluating the Boolean expressions by their positions in program, we can avoid calculating the entire expression. For example, if a Boolean expression is X and Y , and if X is false, then we can conclude that the entire expression is false without having to evaluate Y . This method is useful in implementing the Boolean expressions in control-flow statements such as if-then-else and while-do statements. For example, we consider the Boolean expressions in context of conditional statements such as

- If X then S_1 else S_2
- while X do S

In the first statement, if X is true, the control jumps to the first statement of the code for S_1 , and if X is false, the control jumps to the first statement of S_2 as shown in Figure 7.7(a). In case of second statement, when X is false, the control jumps to the statement immediately following the while statement, and if X is true, the control jumps to the first statement of the code for S as shown in Figure 7.7(b).

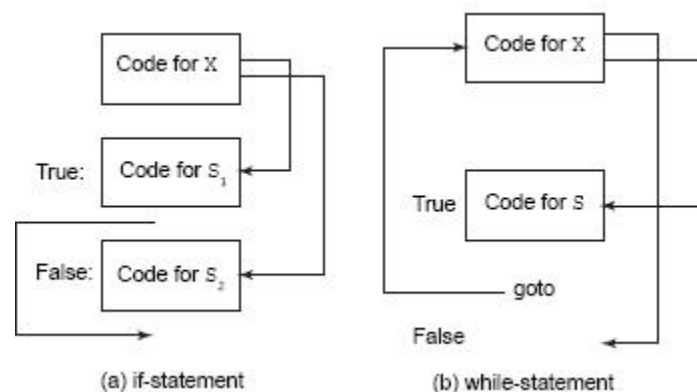


Figure 7.7 Control-flow Translation of Boolean Expressions

11. What is postfix translation?

Ans: A translation scheme is said to be postfix translation if, for each production $S \rightarrow \alpha$, the transition rule for S.CODE consists of the concatenation of the CODE translations of the non-terminals in α , in the same order as the non-terminals appear in α , followed by a tail of output. It is easy to use the postfix translation of CODE as it reduce space requirements, otherwise we have to follow a long scheme to construct the intermediate language form of any program like generating a parse tree followed by a walk of the tree.

12. Explain the array references in arithmetic expressions.

Or

Discuss the addressing of array elements.

Ans: An array is a collection of items having similar data type, which are stored in a block of consecutive memory locations. In case of languages like C and Java, array consists of n elements, numbered $0, 1, \dots, n - 1$. For a single-dimensional array, the address of i^{th} element of the array is calculated as $\text{base} + i * w$, where base is the relative address of array element $X[0]$, and w is the width of each array element. In case of a two-dimensional array, the relative address of the array element $X[i_1][i_2]$ is calculated as $\text{base} + i_1 * w_1 + i_2 * w_2$ where w_1 is the width of a row and w_2 is the width of an element in a row. In general, for a k -dimensional array, the formula can be written as follows:

$$\text{base} + i_1 * w_1 + i_2 * w_2 + \dots + i_k * w_k \quad (1)$$

We can also determine the relative address of an array element in terms of the number of elements n_k along k dimensions of the array with each element of width w . In this case, the address calculations are done on the basis of row-major or column-major layout of the arrays. Consider a two-dimensional array $X[2][3]$, which can be stored either in a row-major form or in a column-major form as shown in Figure 7.8.

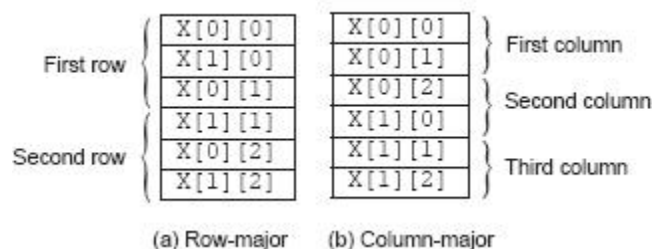


Figure 7.8 Layouts of a Two-dimensional Array

If the elements of a two-dimensional array $X[n_1][n_2]$ are stored in a row-major form, the relative address of an array element $X[i_1][i_2]$ is calculated as follows:

$$\text{base} + (i_1 * n_2 + i_2) * w$$

On the other hand, if the elements are stored in a column-major form, the relative address of $X[i_1][i_2]$ is calculated as follows:

$$\text{base} + (i_1 + i_2 * n_1) * w$$

The row-major and column-major forms can be generalized to k-dimensions. If we generalize row-major form, then elements are stored in such a way that when we scan down a block of storage, the rightmost scripts appear to vary fastest. On the other hand, in case of column-major form, the leftmost scripts appear to vary fastest.

In general, the array elements in one-dimensional array need not be numbered as $0, 1, \dots, n - 1$, rather they can be numbered as $low, low + 1, \dots, high$. Now, the address of an array reference $X[i]$ can be calculated as $base + (i - low) * w$. Here, base is the relative address of $A[low]$.

13. What is backpatching? Explain.

Ans: The syntax directed definitions can be easily implemented by using two passes. In the first pass, we construct a syntax tree for the input, and in the second pass, we traverse the tree in depth first order to complete the translations in the given definition. Generating code for flow of control statements and Boolean expressions is difficult in single pass. This is because we may not be able to know the labels that the control must goto during the generation of jump statements. Thus, the generated code would be a series of branching statements in which the targets of the jumps are temporarily left unspecified. To overcome this problem, we use back patching, which is a technique to solve the problem of replacing symbolic names into the goto statements by the actual target addresses. However, some languages do not permit to use symbolic names in the branches, for this we maintain a list of branches that have the same target labels and then replace them once they are defined. To manipulate the lists of jumps, the following three functions are used:

▣ **makelist(i):** This function creates a new list containing an index i into the array of statements and then returns a pointer pointing to the newly created list.

▣ **merge(p₁, p₂):** This function concatenates the two lists pointed to by the pointers p₁ and p₂ respectively, and then returns a pointer to the concatenated list.

▣ **backpatch(p, i):** This function inserts i as the target labels for each of the statements on the list pointed to by p.

14. Translate the expression $a := -b * (c + d)/e$ into quadruples and triple representation.

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

$t_1 := -b$ (here, '-' represents unary minus)

$t_2 := c + d$

$t_3 := t_1 * t_2$ $t_4 := t_3 / e$

$a := t_4$

The quadruple representation of this three-address code is shown in Figure 7.11.

	Operator	Operand 1	Operand 2	Result
0	-	b		t_1
1	+	c	d	t_2
2	*	t_1	t_2	t_3
3	/	t_3	e	t_4
4	=	t_4		a

Figure 7.11 Quadruple Representation for $a := -b * (c + d)/e$

The triple representation for the expression is given in Figure 7.12.

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

Figure 7.12 Triple Representation for $a := -b * (c + d)/e$

15. 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:

```

t1 := a + b
t2 := -t1
t3 := c + d
t4 := t2 * t3
t5 := t1 + c
t6 := t4 + t5
X := t6

```

The quadruple representation is shown in Figure 7.13.

	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 7.13 Quadruple Representation for $X = -(a + b) * (c + d) + (a + b + c)$

The triple representation for the given expression is shown in Figure 7.14.

	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 7.14 Triple Representation for $X = -(a + b) * (c + d) + (a + b + c)$

16. 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. t₁: = k * width
5. t₂: = addr(a)-width
6. t₂[t₁]: = 0
7. t₃ : = k + 1
8. k: = t₃
9. goto (2)
10. Next

17. 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. t₁: = z + 1
8. z: = t₁
9. goto (1)
10. if x <= s goto (12)
11. goto (1)
12. t₂: = x + 10
13. x: = t₂
14. goto (10)
15. goto (1)
16. Next

18. 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 $k \leq 12$ 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

19. 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

20. 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. t1: = a * 10
5. t1: = t1 + b
6. t1: = t1 * 4
7. t2: = addr(x) - 44
8. t3: = t2 [t1]
9. t4: = b * 10
10. t4: = t4 + a
11. t4: = t4 * 4
12. t5: = addr(y) - 44
13. t6: = t5[t4]
14. t7: = t3 * t6
15. t7: = add + t7
16. t8: = a + 1
17. a: = t8
18. t9: = b + 1
19. b: = t9
20. if a <= 10 goto (22)
21. goto (23)
22. if b <= 10 goto(4)
23. Next

```

21. 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. t1: = 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
```

Type Checking

1. What is a type system? List the major functions performed by the type systems.

Ans: A **type system** is a tractable syntactic framework to categorize different phrases according to their behaviors and the kind of values they compute. It uses logical rules to understand the behavior of a program and associates types with each compound value and then it tries to prove that no type errors can occur by analyzing the flow of these values. A type system attempts to guarantee that only value-specific operations (that can match with the type of value used) are performed on the values. For example, the floating-point numbers in C uses floating-point specific operations to be performed over these numbers such as floating-point addition, subtraction, multiplication, etc.

The language design principle ensures that every expression must have a type that is known (at the latest, at run time) and a type system has a set of rules for associating a type to an expression. Type system allows one to determine whether the operators in an expression are appropriately used or not. An implementation of type system is called **type checker**.

There are two type systems, namely, *basic type system* and *constructed type system*.

❑ **Basic type system:** Basic type system contains atomic types and has no internal structure. They contain integer, real, character, and Boolean. However, in some languages like Pascal, they can have subranges like 1...10 and enumeration types like orange, green, yellow, amber, etc.

❑ **Constructed type system:** Constructed type system contains arrays, records, sets, and structure types constructed from basic types and/or from other constructed types. They also include pointers and functions.

Type system provides some functions that include:

❑ **Safety:** A type system allows a compiler to detect meaningless or invalid code which does not make a sense; by doing this it offers more strong typing safety. For example, an expression 5/"Hi John" is treated as invalid because arithmetic rules do not specify how to divide an integer by a string.

❑ **Optimization:** For optimization, a type system can use static and/or dynamic type checking, where static type checking provides useful compile-time information and dynamic type checking verifies and enforces the constraints at runtime.

❑ **Documentation:** The more expressive type systems can use types as a form of documentation to show the intention of the programmer.

❑ **Abstraction (or modularity):** Types can help programmers to consider programs as a higher level of representation than bit or byte by hiding lower level implementation.

2. Define type checking. Also explain the rules used for type checking.

Ans: **Type checking** is a process of verifying the type correctness of the input program by using logical rules to check the behavior of a program either at compile time or at runtime. It allows the programmers to limit the types that can be used for semantic aspects of compilation. It assigns types to values and also verifies whether these values are consistent with their definition in the program.

Type checking can also be used for detecting errors in programs. Though errors can be checked dynamically (at runtime) if the target program contains both the type of an element and its value, but a sound type system eliminates the need for dynamic checking for type errors by ensuring that these errors would not arise when the target program runs.

If the rules for type checking are applied strongly (that is, allowing only those automatic type conversions which do not result in loss of information), then the implementation of the language is said to be *strongly typed*; otherwise, it is said to be *weakly typed*. A strongly typed language implementation guarantees that the target program will run without any type errors.

Rules for type checking: Type checking uses syntax-directed definitions to compute the type of the derived object from the types of its syntactical components. It can be in two forms, namely, *type synthesis* and *type inference*.

□ **Type synthesis:** Type synthesis is used to build the type of an expression from the types of its subexpressions. For type synthesis, the names must be declared before they are used. For example, the type of expression $E_1 * E_2$ depends on the types of its sub-expressions E_1 and E_2 . A typical rule is used to perform type synthesis and has the following form:

if expression f has a type $s \rightarrow t$ **and** expression x has a type s , **then** expression $f(x)$ will be of type t
Here, $s \rightarrow t$ represents a function from s to t . This rule can be applied to all functions with one or more arguments. This rule considers the expression $E_1 * E_2$ as a function, $\text{mul}(E_1, E_2)$ and uses E_1 and E_2 to build the type of $E_1 * E_2$.

□ **Type inference:** Type inference is the analysis of a program to determine the types of some or all of the expressions from the way they are used. For example,

```
public int mul(int E1, int E2)
    return E1 * E2;
```

Here, E_1 and E_2 are defined as integers. So by type inference, we just need definition of E_1 and E_2 . Since the resulting expression $E_1 * E_2$ uses $*$ operation, which would be taken as integer because it is performed on two integers E_1 and E_2 . Therefore, the return type of mul must be an integer. A typical rule is used to perform type inference and has the following form:

if $f(x)$ is an expression,
then for some type variables α and β , f is of type $\alpha \rightarrow \beta$ **and**
 x is of type α

3. Explain type expressions.

Ans: Type expressions are used to represent structure of types and can be considered as a textual representation for types. A type expression can either be of basic type or can be created by applying an operator (known as **type constructor**) to a type expression.

For example, a type expression for the array type $\text{array int}[3][5]$ considers it as an “array of 3 arrays having 5 integers in each of them”, and its type expression can be written as “array (3, array (5, integer))”. The type 3 array expression uses a tree to represent the type structure. The tree representation of array type $\text{int}[3][5]$ is shown in Figure 8.1, where an operator **array** takes two arguments: a number and a type.

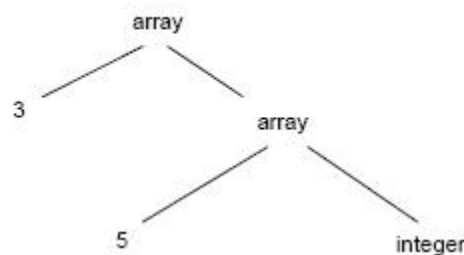


Figure 8.1 Type Expression of $\text{Int}[3][5]$

Type expressions can be defined as follows:

□ **Basic types:** Every basic types such as Boolean, char, integer, float, void, etc., is a type expression.

□ **Type names:** Every type name is a type expression.

❑ **Constructed types:** A constructed type applies constructors to the type expressions, which can be:

- **Arrays:** A type expression can be constructed by applying an array type constructor to a number and a type expression. It can be represented as $\text{array}(I, T)$, where I is an index type and T is the type of array elements. For example, $\text{array}(1 \dots 10, \text{integer})$.
- **Cartesian product:** For any type expressions T_1 and T_2 , the Cartesian product $T_1 \times T_2$ is also a type expression.
- **Record with field names:** A record type constructor is applied to the field names to form a type expression. For example, $\text{record}\{\text{float } a, \text{int } b\} \rightarrow X$;
- **Function types:** A type constructor \rightarrow is used to form a type expression for function types. For example, $A \rightarrow B$ denotes a function from type A to type B . For example, $\text{real} \times \text{real} \rightarrow \text{real}$.
- **Pointers:** A type expression pointer (T_1) represents a pointer to an object of type T_1 .

4. Define these terms: static type checking, dynamic type checking, and strong typing.

Ans: Static type checking: In static type checking, most of the properties are verified at compile time before the execution of the program. The languages C, C++, Pascal, Java, Ada, FORTRAN, and many more allow static type checking. It is preferred because of the following reasons:

- ❑ As the compiler uses type declarations and determines all types at compile time, hence catches most of the common errors at compile time.
- ❑ The execution of output program becomes fast because it does not require any type checking during execution.

The main disadvantage of this method is that it does not provide flexibility to perform type conversions at runtime. Moreover, the static type checking is conservative, that is, it will reject some programs that may behave properly at runtime, but that cannot be statically determined to be well-typed.

Dynamic type checking: Dynamic type checking is performed during the execution of the program and it checks the type at runtime before the operations are performed on data. Some languages that support dynamic type checking are Lisp, Java Script, Smalltalk, PHP, etc. Some advantages of the dynamic type checking are as follows:

- ❑ It can determine the type of any data at runtime.
- ❑ It gives some freedom to the programmer as it is less concerned about types.
- ❑ In dynamic typing, the variables do not have any types associated with them, that is, they can refer to a value of any type at runtime.
- ❑ It checks the values of all the data types during execution which results in more robust code.
- ❑ It is more flexible and can support union types, where the user can convert one type to another at runtime.

The main disadvantage of this method is that it makes the execution of the program slower by performing repeated type checks.

Strong typing: A type checking which guarantees that no type errors can occur at runtime is called **strong type checking** and the system is called **strongly typed**. The strong typing has certain disadvantages such as:

- ❑ There are some checks like array bounds checking which require dynamic checking.
- ❑ It can result into performance degradation.
- ❑ Generally, these type systems have holes in the type systems, for example, variant records in Pascal.

5. Write down the process to design a type checker.

Ans: A type checker is an implementation of a type system. The process includes the following steps:

1. **Identifying the available types in the language:** We have two types language.

- Base types (integer, double, Boolean, string, and so on)
- Compound types (arrays, classes, interfaces, and so on)

2. Identifying the language constructs with associated types: Each programming language consists of some constructs and each of them is associated with a type as discussed below:

- **Constants:** Every constant has an associated type. A scanner identifies the types and associated lexemes of a constant.
- **Variables:** A variable can be global, local, or an instance of a class. Each of these variables must have a declared type, which can either be one of the base types or the supported compound types.
- **Functions:** The functions have a return type, and the formal parameters in function definition as well as the actual arguments in the function call also have a type associated with them.
- **Expressions:** An expression can contain a constant, variable, functional call, or some other operators (like unary or binary operators) that can be applied in an expression. Hence, the type of expression depends on the type of constant, variable, operands, function return type, and on the type of operators.

3. Identifying the language semantic rules: The production rules to parse variable declarations can be written as:

Variable Declaration \rightarrow Variable

Variable \rightarrow Type identifier

Type \rightarrow int|double|Boolean|string|identifier|Type[]

The parser stores the name of an identifier lexeme as an attribute attached to the token. The name associated with the identifier symbol, and the type associated with the identifier and type symbol are used to reduce the variable production. A new variable declaration is created by declaring an identifier of that type and that variable is stored in the symbol table for further lookup.

6. What is type equivalence?

Ans: Type equivalence is used by the type checking to check whether the two type expressions are equivalent or not. It can be done by checking the equivalence between the two types. The rule used for type checking works as follows:

if two type expressions are equal **then** return a certain type **else** return a type error()

When two type expressions are equivalent, we need a precise definition of both the expressions. When names are given to type expressions, and these names are further used in subsequent type expressions, it may result in potential ambiguities. The key issue is whether a name in a type expression stands for itself, or it is an abbreviation for another type expression.

There are two schemes to check type equivalence of expressions:

Structural equivalence: Structural equivalence needs a graph to represent the type expressions. The two type expressions are said to be structurally equivalent if and only if they hold any of the following conditions:

- They are of identical basic type.
- Same type constructor has been applied to equivalent types to construct the type expressions.
- A type name of one represents the other.

Name equivalence: If type names are treated as standing for themselves, then the first two conditions of structural equivalence lead to another equivalence of type expressions called name equivalence. In other words, name equivalence considers types to be equal if and only if the same type names are used and one of the first two conditions of structure equivalence holds.

For example, consider the following few types and variable declarations.

typedef double Value

...

...

Value var1, var2

Sum var3, var4

In these statements, var1 and var2 are name equivalent, so are var3 and var4, because their type names are same. However, var1 and var3 are not name equivalent, because their type names are different.

7. Explain type conversion.

Ans: Type conversion refers to the conversion of a certain type into another by using some semantic rules. Consider an expression $a + b$, where a is of type `int` and b is of type `float`. The representations of floats and integers are different within a computer, and an operation on integers and floats uses different machine instructions. Now, the primary task of the compiler is to convert one of the operands of $+$ to make both of the operands to same type. For example, an expression $5 * 7.14$ has two types, one is of float type and other one is of type `int`. To convert integer type constant into float type, we use a unary operator (`float`) as shown here:

```
x =(float)5
```

```
y = x * 7.14
```

The type conversion can be done implicitly or explicitly. The conversion from one type to another is called **implicit**, if it is automatically done by the computer. Usually, implicit conversions of constants can be done at compile time and it results in an improvement in the execution time of the object program. Implicit type conversion is also known as **coercion**. A conversion is said to be **explicit** if the programmer must write something to cause the conversion. For example, all conversions in Ada are explicit. Explicit conversions can be considered as a function applications to a type checker. Explicit conversion is also known as **casts**.

Conversion in languages can be considered as *widening conversions* and *narrowing conversions*, as shown in Figure 8.2(a) and (b), respectively.

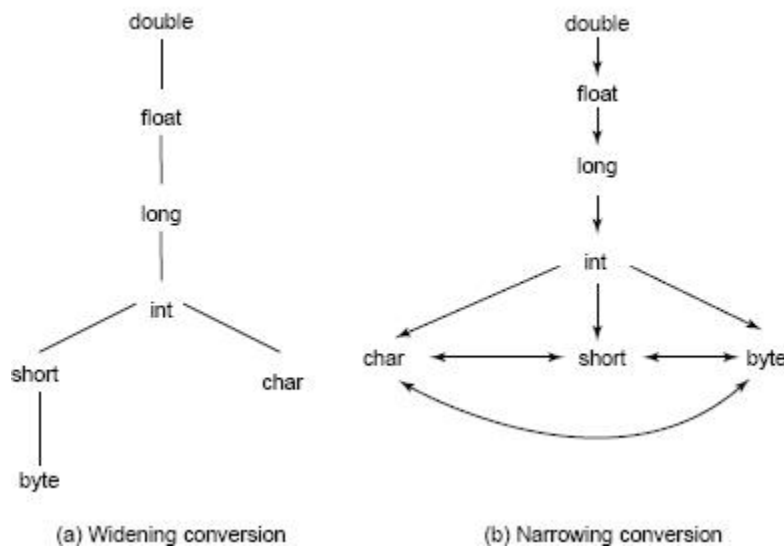


Figure 8.2 Conversion Between Primitive Types in Java

The rules used for widening is given by the hierarchy in Figure 8.2(a) and are used to preserve the information. In widening hierarchy, any lower type can be widened into a higher type like a byte can be widened to a short or to an `int` or to a float, but a short cannot be widened to a char.

The narrowing conversions, on the other hand, may result in loss of information. The rules used for narrowing is given by the hierarchy in Figure 8.2(b), in which a type x can be narrowed to type y if and only if there exists a path from x to y . Note that `char`, `short`, and `byte` are pairwise convertible to each other.

Runtime Administration

1. Define runtime environment. What are the issues in runtime environment?

Ans: The source language definition contains various abstractions such as names, data types, scopes, bindings, operators, parameters, procedures, and flow of control constructs. These abstractions must be implemented by the compiler. To implement these abstractions on target machine, compiler needs to cooperate with the operating system and other system software. For the successful execution of the program, compiler needs to create and manage a **runtime environment**, which broadly describes all the runtime settings for the execution of programs.

In case of compilation of a program, the runtime environment is indirectly controlled by generating the code to maintain it. However, in case of interpretation of the program, the runtime environment is directly maintained by the data structures of the interpreter.

The runtime environment deals with several issues which are as follows:

- ❑ The allocation and layout of storage locations for the objects used in the source program.
- ❑ The mechanisms for accessing the variables used by the target program.
- ❑ The linkages among procedures.
- ❑ The parameter passing mechanism.
- ❑ The interface to input/output devices, operating systems and other programs.

2. What are the important elements in runtime environment?

Ans: The important elements that constitute a runtime environment for a program are as follows:

❑ **Memory organization:** During execution, a program requires certain amount of memory for storing the local and global variables, source code, certain data structures, and so on. The way memory is organized for storing these elements is an important characteristic of runtime environment. Different programming languages support different memory organization schemes. For example, C++ supports the use of pointers and dynamic memory with the help of `new()` and `delete()` functions, whereas FORTRAN 77 does not support pointers and usage of dynamic memory.

❑ **Activation records:** The execution of a procedure in a program is known as the **activation** of the procedure. The activation of procedures or functions is managed with the help of a contiguous block of memory known as **activation record**. Activation record can be created statically or dynamically. Statically, a single activation record can be constructed, which is common for any number of activations. Dynamically, number of activation records can be constructed, one for each activation. The activation record contains the memory for all the local variables of the procedure, depending on the way by which activation record is created, the target code has to be generated accordingly to access the local variables.

❑ **Procedure calling and return sequence:** Whenever a procedure is invoked or called, certain sequence of operations need to be performed, which include evaluation of function arguments, storing it at a specified memory location, transferring the control to the called procedure, etc. This sequence of operations is known as **calling sequence** and **procedure calling**. Similarly, when the activated procedure terminates, some other operations need to be performed such as fetching the return value from a specified memory location, transferring the control back to the calling procedure, etc. This sequence of operations is known as **return sequence**. The calling sequence and return sequence differ from one language to another, and in some cases even from one compiler to another for the same language.

❑ **Parameter passing:** The functions used in a program may accept one or more parameters. The values of these parameters may or may not be modified inside the function definition. Moreover, the modified values may or may not be reflected in the calling procedure depending on the language used. In some languages like PASCAL and C++, some rules are specified which determine whether the modified value should be reflected in the calling procedure. In certain languages like FORTRAN77 the modified values are always reflected in the calling procedure. There are several techniques by which parameters can be passed to functions. Depending on the parameter passing technique used, the target code has to be generated.

3. Give subdivision of runtime memory.

Or

What is storage organization?

Or

Explain stack allocation and heap allocation?

Or

What is dynamic allocation? Explain the techniques used for dynamic allocation (stack and heap allocation).

Ans: The target program (already compiled) is executed in the runtime environment within its own logical address space known as **runtime memory**, which has a storage location for each program value. The compiler, operating system, and the target machine share the organization and management of this logical address. The runtime representation of the target program in the logical address space comprises data and program areas as shown in Figure 9.1. These areas consist of the following information:

- ❑ The generated target code
- ❑ Data objects
- ❑ Information to keep track of procedure activations.

Since the size of the target code is fixed at compile time, it can be placed in a statically determined area named *Code* (see Figure 9.1), which is usually placed in the low end of memory. Similarly, the memory occupied by some program data objects such as global constants can also be determined at compile time. Hence, the compiler can place them in another statically determined area of memory named *Static*. The main reason behind the static allocation of as many data objects as possible is that the compiler could compile the addresses of these objects into the target code. For example, all the data objects in FORTRAN are statically allocated.

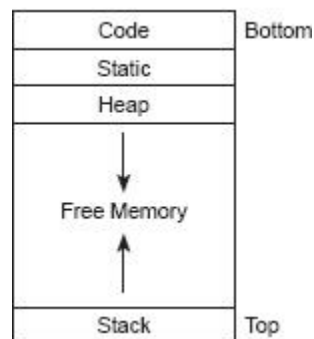


Figure 9.1 Subdivision of Runtime Memory

The other two areas, namely, *Stack* and *Heap* are used to maximize the utilization of space of runtime. The size of these areas is not fixed, that is, as the program executes their size can change. Hence, these areas are dynamic in nature.

Stack allocation: The stack (also known as **control stack** or **runtime stack**) is used to store activation records that are generated during procedure calls. Whenever a procedure is invoked, the activation record corresponds to that procedure is pushed onto the stack and all local items of the

procedure are stored in the activation record. When the execution of procedure is completed, the corresponding activation record is popped from the stack and the values of locals are deleted.

The stack is used to manage and allocate storage for the active procedure such that

- ❑ On the occurrence of a procedure call, the execution of the calling procedure is interrupted, and the activation record for the called procedure is constructed. This activation record stores the information about the status of the machine.

- ❑ On receiving control from the procedure call, the values in the relevant registers are restored and the suspended activation of the calling procedure is resumed, and then the program counter is updated to the point immediately after the call. The stack area of runtime storage is used to store all this information.

- ❑ Some data objects which are contained in this activation and their relevant information are also stored in the stack.

- ❑ The size of the stack is not fixed. It can be increased or decreased according to the requirement during program execution.

Runtime stack is used in C and Pascal.

Heap allocation: The main limitation of stack area is that it is not possible to retain the values of non-local variables even after the activation record. This is because of last-in-first-out nature of stack allocation. To retain the values of such local variables, heap allocation is used. The heap allocates a contiguous memory locations as and when required for storing the activation records and other data elements. When the activation ends, the memory is deallocated, and this free space can be further used by the heap manager. The heap management can be made efficient by creating a linked list of free blocks. Whenever some memory is deallocated, the free block is appended in the linked list, and when memory needs to be allocated, the most suitable (best-fit) memory block is used for allocation. The heap manager dynamically allocates the memory, which results into a runtime overhead of taking care of defragmentation and garbage collection. The garbage collection enables the runtime system to automatically detect unused data elements and reuse their storage.

4. Explain static allocation. What are its limitations?

Ans: An allocation is said to be static if all data objects are stored at compile time. It has the following properties:

- ❑ The binding of names is performed during compilation and no runtime support package is required.

- ❑ The binding remains same at runtime as well as compile time.

- ❑ Each time a procedure is invoked, the names are bounded to the same storage locations. The values of local variables remain unchanged before and after the transfer of controls.

- ❑ The storage requirement is determined by the type of a name.

Limitations of static allocation are given as follows:

- ❑ The information like size of data objects and constraints on their memory position needs to be present during compilation.

- ❑ Static allocation does not support any dynamic data structures, because there is no mechanism to support run-time storage allocation.

- ❑ Since all the activations of a given procedure use the same bindings for local names, recursion is not possible in static allocation.

5. Explain in brief about control stack.

Ans: A stack representing procedure calls, return, and flow of control is called a **control stack** or **runtime stack**. Control stack manages and keeps track of the activations that are currently in progress. When the activation begins, the corresponding activation node is pushed onto the stack

and popped out when the activation ends. The control stack can be nested as the procedure calls or activations nest in time such that if p calls q , then the activation of q is nested within the activation of p .

6. Define activation tree.

Ans: During the execution of program, activation of the procedures can be represented by a tree known as **activation tree**. It is used to depict the flow of control between the activations. Activations are represented by the nodes in activation tree where each node corresponds to one activation, and the root node represents the activation of the main procedure that initiates the program execution. Figure 9.2 shows that the `main()` activates two procedures P_1 & P_2 . The activations of procedures P_1 & P_2 are represented in the order in which they are called, that is, from left to right. It is important to note that the left child node must finish its execution before the activation of right node can begin. The activation of P_2 further activates two procedures P_3 and P_4 . The flow of control between the activations can be depicted by performing a depth first traversal of the activation tree. We start with the root of the tree. Each node is visited before its child nodes are visited and the child nodes are visited from left to right. When all the child nodes of a particular node have been visited, we can say that the procedure activation corresponding to a node is completed.

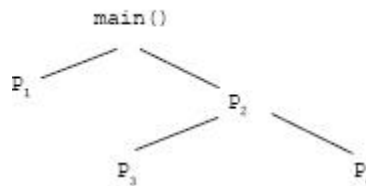


Figure 9.2 Activation Tree

7. Discuss in detail about activation records.

Ans: The **activation record** is a block of memory on the control stack used to manage information for every single execution of a procedure. Each activation has its own activation record with the root of activation tree at the bottom. The path from one activation to another in the activation tree is determined by the corresponding sequence of activation records on the control stack.

Different languages have different activation record contents. In FORTRAN, the activation records are stored in the static data area while in C and Pascal, the activation records are stored in stack area. The contents of activation records are shown in Figure 9.3.

The various fields of activation record are as follows:

- ❑ **Temporaries:** The temporaries are used to store intermediate results that are generated during the evaluation of an expression and cannot be held in registers.
- ❑ **Local data:** This field contains local data like local variables, which are local to the execution of a procedure stored in the activation record.
- ❑ **Saved machine status:** This field contains the information regarding the state of a machine just before the procedure is called. This information consists of the machine register contents and the return address (program counter value).
- ❑ **Access link:** It is an optional field and also called **static link** field. It is a link to non-local data in some other activation record which is needed by the called procedure.
- ❑ **Control link:** It is also an optional field and is called dynamic link field. It points to the activation record of the calling procedure.
- ❑ **Returned value:** It is also an optional field. It is not necessary that all the procedures return a value, but if the procedure does, then for better efficiency this field stores the return value of the called procedure.

❑ **Actual parameters:** This field contains the information about actual parameters which are used by the calling procedure.

Actual parameters
Returned values
Control link (Dynamic link)
Access link (Static link)
Saved machine status
Local data (variables)
Temporaries

Figure 9.3 Activation Record Model

8. Explain register allocation.

Ans: On the target machine, registers are the fastest for the computation as fetching the data from the registers is easy and efficient as compared to fetching it from the memory. So the instructions that involve the use of registers are much smaller and faster than those using memory operands. The main problem with the usage of the register is that the system has limited number of registers which are not enough to hold all the variables. Thus, an efficient utilization of registers is very important to generate a good code. The problem of using registers is divided into two sub problems:

❑ **Register allocation:** It includes selecting a set of variables to be stored within the registers during program execution.

❑ **Register assignment:** It includes selecting a specific register to store a variable.

The purpose of register allocation is to map a large number of variables into a few numbers of registers, which may result in sharing of single register by several variables. However, since two variables in use cannot be kept in the same register at the same time, therefore, the variables that cannot be assigned to any register must be kept in the memory.

Register allocation can be done either in intermediate language or in machine language. If register allocation is done during intermediate language, then the same register allocator can be used for several target machines. Machine languages, on the other hand, initially use symbolic names for registers, and register allocation turns these symbolic names into register numbers.

It is really difficult to find an optimal assignment of registers. Mathematically, the problem to find a suitable assignment of registers can be considered as a NP-Complete problem. Sometimes, the target machine's operating system or hardware uses certain register-usage convention to be observed, which makes the assignment of registers more difficult.

For example, in case of integer division and integer multiplication, some machines use even/odd register pairs to store operands and the results. The general form of a multiplication instruction is as follows:

M a,b

Here, operand a is a multiplicand, and is in the odd register of an even/odd register pair and b is the multiplier, and it can be stored anywhere. After multiplication, the entire even/odd register pair is occupied by the product.

The division instruction is written as

D a,b

x = a - b	x = a - b
x = x * c	x = x - c
x = x/d	x = x/d
(a)	(b)

Figure 9.4 Two Three-address Code Sequences

Here, dividend a is stored in the even register of an even/odd register pair and the divisor b can be stored anywhere. After division, remainder is stored in the even register and quotient is stored in the odd register.

Now, consider the two three-address code sequences given in Figure 9.4(a) and (b).

These three-address code sequences are almost same; the only difference is the operator in the second statement. The assembly code sequences for these three-address code sequences are given in Figure 9.5(a) and (b).

L	R ₁ , a	L	R ₀ , a
S	R ₁ , b	S	R ₀ , b
M	R ₀ , c	S	R ₀ , c
D	R ₀ , d	SRDA	R ₀ , 32
ST	R ₁ , x	D	R ₀ , d
		ST	R ₁ , x
(a)		(b)	

Figure 9.5 Assembly Code (Machine Code) Sequences

Here, L, ST, S, M, and D stand for load, store, subtract, multiply, and divide respectively. R_0 and R_1 are machine registers and SRDA stands for Shift-Right-Double-Arithmetic. SRDA R_0 , 32 shifts the dividend into R_1 and clears R_0 to make all bits equal to its sign bit.

9. Explain the various parameter passing mechanisms of a high-level language.

Or

What are the various ways to pass parameters in a function?

Ans: When one procedure calls another, the communication between the procedures occurs through non-local names and through parameters of the called procedure. All the programming languages have two types of parameters, namely, *actual parameters* and *formal parameters*. The **actual parameters** are those parameters which are used in the call of a procedure; however, **formal parameters** are those which are used during the procedure definition. There are various parameter passing methods but most of the recent programming languages use *call by value* or *call by reference* or both. However, some older programming languages also use another method *call by name*.

❑ **Call by value:** It is the simplest and most commonly used method of parameter passing. The actual parameters are evaluated (if expression) or copied (if variable) and then their r-values are passed to the called procedure. **r-value** refers to the value contained in the storage. The values of actual parameters are placed in the locations which belong to the corresponding formal parameters of the called procedure. Since the formal and actual parameters are stored in different memory locations, and formal parameters are local to the called procedure, the changes made in the values of formal parameters are not reflected in the actual parameters. The languages C, C++, Java, and many more use call by value method for passing parameters to the procedures.

❑ **Call by reference:** In call by reference method, parameters are passed by reference (also known as **call by address** or **call by location**). The caller passes a pointer to the called procedure, which points to the storage address of each actual parameter. If the actual parameter is a name or an expression having an l-value, then the l-value itself is passed (here, l-value represents the address of the actual parameter). However, if the actual parameter is an expression like $a + b$ or 2, having no l-value, then that expression is calculated in a new location, and the address of that new location is passed. Thus, the changes made in the calling procedure are reflected in the called procedure.

❑ **Call by name:** It is a traditional approach and was used in early programming languages, such as ALGOL 60. In this approach, the procedure is considered as a macro, and the body of the procedure

is substituted for the call in the caller and the formals are literally substituted by the actual parameters. This literal substitution is called **macro expansion** or **in-line expansion**. The names of the calling procedure are kept distinct from the local names of the called procedure. That is, each local name of the called procedure is systematically renamed into a distinct new name before the macro expansion is done. If necessary, the actual parameters are surrounded by parentheses to maintain their integrity.

10. What is the output of this program, if compiler uses following parameter passing methods?

(1) Call by value (2) Call by reference (3) Call by name

The program is given as:

```
void main (void)
{
    int a, b;
    void A(int, int, int);
    a = 2, b = 3;
    A(a + b, a, a);
    printf ("%d", a);
}

void A (int x, int y, int z)
{
    y = y + 1;
    z = z + x;
}
```

Ans: Call by value: In call by value, the actual values are passed. The values $a = 2$ and $b = 3$ are passed to the function A as follows:

$A(2 + 3, 2, 2);$

The value of a is printed as 2, because the updated value is not reflected in main().

Call by reference: In call by reference, both formal parameters y and z have the same reference that is, a. Thus, in function A the following values are passed.

$x = 5$

$y = 2$

$z = 2$

After the execution of $y = y + 1$, the value of y becomes

$y = 2 + 1 = 3$

Since y and z are referring to the same memory location, z also becomes 3. Now after the execution of statement $z = z + x$, the value of z becomes

$z = 3 + 5 = 8$

When control returns to main(), the value of a will now become 8. Hence, output will be 8.

Call by name: In this method, the procedure is treated as macro. So, after the execution of the function

$x = 5$

$y = y + 1 = 2 + 1 = 3$

$z = z + x = 2 + 5 = 7$

When control returns to main(), the value of a becomes 7. Hence, output will be 7.

Symbol Table

1. What is symbol table and what kind of information it stores? Discuss its capabilities and also explain the uses of symbol table.

Ans: A **symbol table** is a compile time data structure that is used by the compiler to collect and use information about the source program constructs, such as variables, constants, functions, etc. The symbol table helps the compiler in determining and verifying the semantics of given source program. The information in the symbol table is entered in the lexical analysis and syntax analysis phase, however, is used in later phases of compiler (semantic analysis, intermediate code generation, code optimization, and code generation). Intuitively, a symbol table maps names into declarations (called attributes), for example, mapping a variable name `a` to its data type `char`.

Each time a name is encountered in the source program, the compiler searches it in the symbol table. If the compiler finds a new name or new information about an existing name, it modifies the symbol table. Thus, an efficient mechanism must be provided for retrieving the information stored in the table as well as for adding new information to the table. The entries in the symbol table consists of (name, information) pair. For example, for the following variable declaration statement, `char a;`

The symbol table entry contains the name of the variable along with its data type.

More specifically, the symbol table contains the following information:

- ❑ The character string (or lexeme) for the name. If the same name is assigned to two or more identifiers which are used in different blocks or procedures, then an identification of the block or procedure to which this name belongs to must also be stored in the symbol table.
- ❑ For each type name, the type definition is stored in the symbol table.
- ❑ For each variable name, its type (int, char, or real), its form (label, simple variable, or array), and its location in the memory must also be stored. If the variable is an array, then some other attributes such as its dimensions, and its upper and lower limits along each dimension are also stored. Other attributes such as storage class specifier, offset in activation record, etc. can also be stored.
- ❑ For each function and procedure, the symbol table contains its formal parameter list and its return type.
- ❑ For each formal parameter, its name, type and type of passing (by value or by reference) is also stored.

A symbol table must have the following capabilities:

- ❑ **Lookup:** To determine whether a given name is in the table.
- ❑ **Insert:** To add a new name (a new entry) to the table.
- ❑ **Access:** To access the information related with the given name.
- ❑ **Modify:** To add new information about a known name.
- ❑ **Delete:** To delete a name or group of names from the table.

The information stored in the symbol table can be used during several stages of compilation process as discussed below:

- ❑ In semantic analysis, it is used for checking the usage of names that are consistent with respect to their implicit and explicit declaration.
- ❑ During code generation, it can be used for determining how much and what kind of runtime storage must be allocated to a name.
- ❑ The information in the symbol table also helps in error detection and recovery. For example, we can determine whether a particular error message has been displayed before, and if already displayed then avoid displaying it again.

2. What are the symbol table requirements? What are the demerits in the uniform structure of symbol table?

Ans: The basic requirements of a symbol table are as follows:

- ❑ **Structural flexibility:** Based on the usage of identifier, the symbol table entries must contain all the necessary information.
- ❑ **Fast lookup/search:** The table lookup/search depends on the implementation of the symbol table and the speed of the search should be as fast as possible.
- ❑ **Efficient utilization of space:** The symbol must be able to grow or shrink dynamically for an efficient usage of space.
- ❑ **Ability to handle language characteristics:** The characteristic of a language such as scoping and implicit declaration needs to be handled.

Demerits in Uniform Structure of Symbol Table:

- ❑ The uniform structure cannot handle a name whose length exceed upper bound or limit of name field.
- ❑ If the length of a name is small, then the remaining space is wasted.

3. Write down the operations performed on a symbol table.

Ans: The following operations can be performed on a symbol table:

❑ **Insert:** The insert operation inserts a new name into the table and returns an index of new entry. The syntax of insert function is as follows:
`insert(String key, Object binding)`
 For example, the function `insert(s,t)` inserts a new string `s` in the table and returns an index of new entry for string `s` and token `t`.

❑ **Lookup:** This operation searches the symbol table for a given name. The syntax of lookup function is as follows:
`object_lookup(string key)`
 For example, the function `object_lookup(s)` returns an index of the entry for the string `s`; if `s` is not found, it returns 0.

❑ **Search/Insert:** This operation searches for a given name in the symbol table, and if not found, it inserts it into the table.

❑ **begin_scope () and end_scope ():** The `begin_scope()` begins a new scope, when a new block starts, that is, when the token `{` is encountered. The `end_scope()` removes the scope when the scope terminates, that is, when the token `}` is encountered. After removing a scope, all the declarations inside this scope are also removed.

❑ **Handling reserved keywords:** Reserved keywords like 'PLUS', 'MINUS', 'MUL', etc., are handled by the symbol table in the following manner.

```
insert ("PLUS", PLUS);
insert ("MINUS", MINUS);
insert ("MUL", MUL);
```

The first 'PLUS', 'MINUS', and 'MUL' in the insert operation indicate lexeme and second one indicate the token.

4. Explain symbol table implementation.

Ans: The implementation of a symbol table needs a particular data structure, depending upon the symbol table specifications. Figure 10.1 shows the data structure for implementation of a symbol table.

The character string forming an identifier is stored in a separate array `arr_lexeme`. Each string is terminated by an EOS (end of string character), which is not a part of identifiers. Each entry in the symbol table `arr_symbol_table` is a record having two or more fields, where first field named `lexeme_pointer` points to the beginning of the lexeme, and the second field `Token` consists of the token name. Symbol table also contains two more fields, namely `attribute`, which holds the attribute values, and `position`, which indicates the position of a lexeme in the symbol table are used.

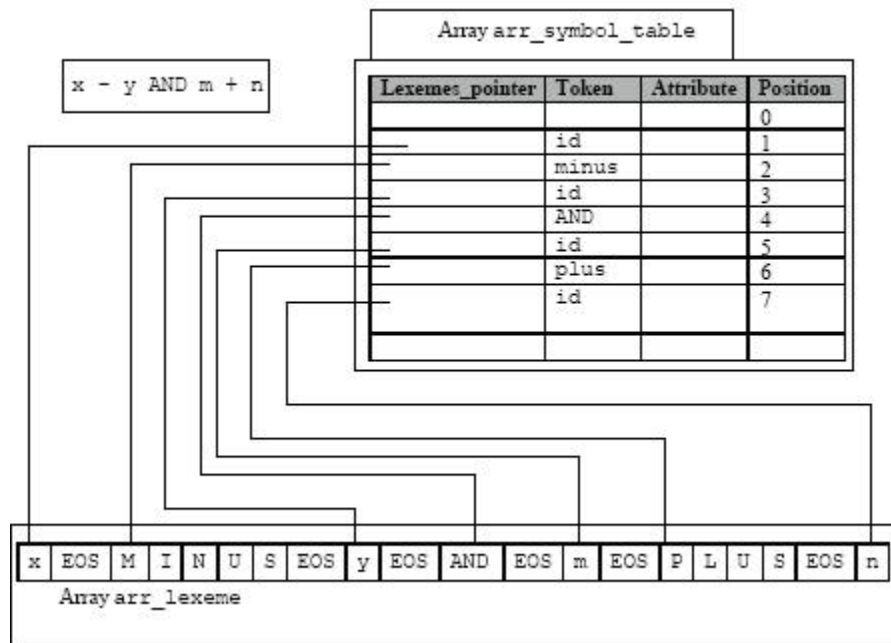


Figure 10.1 Implementation of Symbol Table

Note that the 0th entry in the symbol table is left empty, as a lookup operation returns 0, if the symbol table does not have an entry for a particular string. The 1st, 3rd, 5th and 7th entries are for the x, y, m, and n respectively. The 2nd, 4th and 6th entries are reserved keyword entries for MINUS, AND and PLUS respectively.

Whenever the lexical analyzer encounters a letter in an input string, it starts storing the subsequent letters or digits in a buffer named `lex_buffer`. It then scans the symbol table using the `object_lookup()` operation to determine whether the collected string is in the symbol table. If the lookup operation returns 0, that is, there is no entry for the string in `lex_buffer`, a new entry for the identifier is created using `insert()`. After the insert operation, the index `n` of symbol table entry for the entered string is passed to the parser by setting the `tokenval` to `n`, and an entry in the `Token` field of the token is returned.

5. Discuss various approaches used for organization of symbol table.

Or

Explain the various data structure used for implementing the symbol table.

Ans: The various data structures used for implementing the symbol table are linear list, self-organizing list, hash table, and search tree. The organization of symbol table depends on the selection of the data structure scheme used to implement the symbol table. The data structure schemes are evaluated on the basis of access time, simplicity, storage and performance.

❑ **Linear list:** A linear list of records is the simplest data structure and it is easiest-to-implement data structure as compared to other data structures for organizing a symbol table. A single array or collection of several arrays is used to store names and their associated information. It uses a simple linear linked list to arrange the names sequentially in the memory.

The new names are added to the table in the order of their arrival. Whenever a new name is added, the whole table is searched linearly or sequentially to check whether the name is already present in the symbol table or not. If not, then a record for the new name is created and added to the linear list at a location pointed to by the space pointer, and the pointer is incremented to point to the next empty location (See Figure 10.2).

Variable	Information (type)	Space (byte)
a	int	2
b	char	1
c	float	4
d	long	4




Figure 10.2 Symbol Table as a Linear List

To access a particular name, the whole table is searched sequentially from its beginning until it is found. For a symbol table having n entries, it will take on average $n/2$ comparisons to find a particular name.

❑ **Self-organizing list:** We can reduce the time of searching the symbol table at the cost of a little extra space by adding an additional LINK field to each record or to each array index. Now, we search the list in the order indicated by links. A new name is inserted at a location pointed to by space pointer, and then all other existing links are managed accordingly. A self-organizing list is shown in Figure 10.3, where the attributes id_1 is related to id_2 and id_3 is related to id_1 , and are linked by the LINK pointer.

Variable	Information
id_1	Info 1
id_2	Info 2
id_3	Info 3

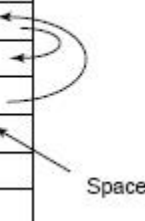


Figure 10.3 Symbol Table as Self Organizing List

The main reason for using the self-organizing list is that if a small set of names is heavily used in a section of program, then these names can be placed at the top while that section is being processed by the compiler. However, if references are random, then the self-organizing list will cost more time and space.

Demerits of self-organizing list are as follows:

- It is difficult to maintain the list if a large set of names is frequently used.
- It occupies more memory as it has a LINK field for each record.
- As self-organizing list organizes it itself, so it may cause problems in pointer movements.

❑ **Hash Table:** A hash table is a data structure that associates keys with values. The basic hashing scheme has two parts:

- A hash table consisting of a fixed array of k pointers to table entries.
- A storage table with the table entries organized into k separate linked lists and each record in the symbol table appears on exactly one of these lists.

To add a name in the symbol table, we need to determine the hash value of that name with the help of a hash function, which maps the name to the symbol table by assigning an integer between 0 to $k - 1$. To search a given name into the symbol, a hash function is applied to that name. Thus, we need to search only that list to determine whether that name exists in the symbol table or not. There is no need to search the entire symbol table. If the name is not present in the list, we create a new record for that name and then insert that record at the head of the list whose index is computed by applying the hash function to the name.

A hash function should be chosen in such a way that it distributes the names uniformly among the k lists, and it can be computed easily for the names comprising character strings. The main advantage of using hash table is that, we can insert or delete any name in $O(n)$ time and search any name in $O(1)$ time. However, in the worst case it can be as bad as $O(n)$.

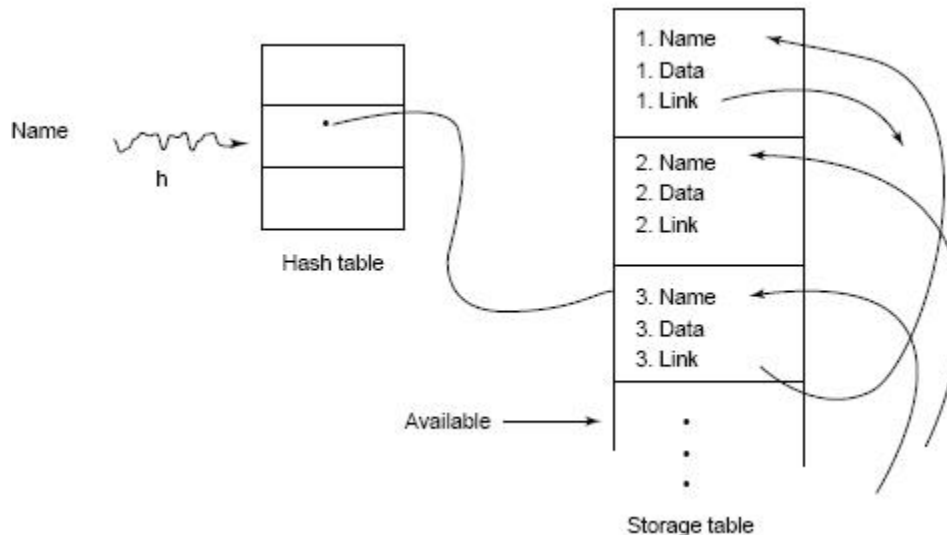


Figure 10.4 Symbol Table as Hash Table

❑ **Search Tree:** Search tree is an approach to organize symbol table by adding two link fields, LEFT and RIGHT, to each record. These two fields are used to link the records into a binary search tree. All names are created as child nodes of root node that always follow the properties of a binary search tree.

- The name in each node is a key value, that is, no two nodes can have identical names.
- The names in the nodes of left sub tree, if exists, is smaller than the value in the root node.
- The names in the nodes of right sub tree, if exists, is greater than the value in the root node.
- The left and right sub trees, if exists, are also binary search trees.

For example: $\text{name} < \text{name}_i$ and $\text{name}_i < \text{name}$. These two statements show that all name smaller than name_i must be left child of name_i ; and all name greater than name_i must be right child of name_i . To insert, search and delete in search tree, the binary search tree insert, search and deletion algorithms are followed respectively.

6. Create list, search tree and hash table for given program.

```

int i, j, k;
int mul (int a, int b)
{
    i = a * b;
    Return (i)
}
main ()
{
    int x;
    x = mul (2, 3);
}

```

Ans: List

Variable	Information	Space
x	integer	2 bytes
i	integer	2 bytes
j	integer	2 bytes
k	integer	2 bytes
a	integer	2 bytes
b	integer	2 bytes
mul	integer	2 bytes
		← Space

Figure 10.5 List Symbol Table for Given Program

Hash Table

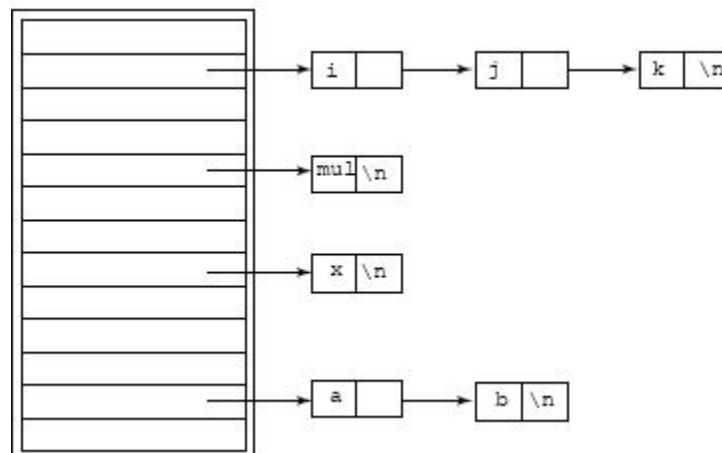


Figure 10.6 Hash Symbol Table for Given Program

Search Tree

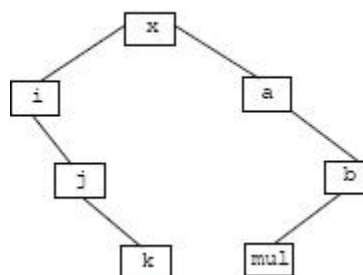


Figure 10.7 Search Tree Symbol Table for the Given Program

7. Discuss how the scope information is represented in a symbol table.

Ans: Scope information characterizes the declaration of identifiers and the portions of the program where it is allowed to use each identifier. Different languages have different scopes for declarations. For example, in FORTRAN, the scope of a name is a single subroutine, whereas in ALGOL, the scope of a name is the section or procedure in which it is declared. Thus, the same identifier may be declared several times as distinct names, with different attributes, and with different intended storage locations. The symbol table is thus responsible for keeping different declaration of the same identifier distinct.

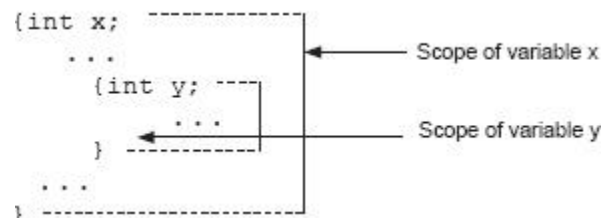
To make distinction among the declarations, a unique number is assigned to each program element that in return may have its own local data. Semantic rules associated with productions that can recognize the beginning and ending of a subprogram are used to compute the number of currently active subprograms.

There are mainly two semantic rules regarding the scope of an identifier.

- ❑ Each identifier can only be used within its scope.
- ❑ Two or more identifiers with same name and are of same kind cannot be declared within the same lexical scope.

The scope declaration of variables, functions, labels and objects within a program is shown here.

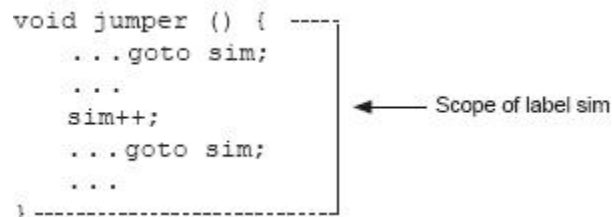
Scope of variables in statement blocks:



❑ Scope of formal arguments of functions:

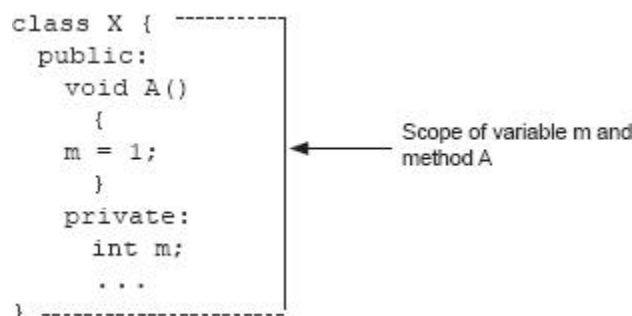


❑ Scope of labels:



❑ Scope in class declaration (scope of declaration): The portion of the program in which a declaration can be applied is called the scope of that declaration. In a procedure a name is said to be local to the procedure if it is in the scope of declaration within the procedure, otherwise the name is said to be non-local.

Scope of object fields and methods:



8. Differentiate between lexical scope and dynamic scope.

Ans: The differences between lexical scope and dynamic scope are given in Table 10.1.

Table 10.1 Difference between lexical and dynamic scope

Lexical Scope	Dynamic Scope
<ul style="list-style-type: none"> ● The binding of name occurrences to declarations is done statically at compile time. ● The structure of the program defines the binding of variables. ● A free variable in a procedure gets its value from the environment in which the procedure is defined. 	<ul style="list-style-type: none"> ● The binding of name occurrences to declarations is done dynamically at run time. ● The binding of variables is defined by the flow of control at the run time. ● A free variable gets its value from where the procedure is called.

9. Explain error detection and recovery in lexical phase, syntactic phase, and semantic phase.

Ans: The classification of errors is given in Figure 10.8.

These errors should be detected during different phases of compiler. Error detection and recovery is one of the main tasks of a compiler. The compiler scans and compiles the entire program, and errors detected during scanning need to be recovered as soon as they are detected.

Usually, most of the errors are encountered during syntax and semantic analysis phase. Every phase of a compiler expects the input to be in particular format, and an error is returned by the compiler whenever the input is not in the required format. On detection of an error, the compiler scans some of the tokens ahead of the point of error occurrence. A compiler is said to have better error-detection capability if it needs to scan only a few numbers of tokens ahead of the point of error occurrence.

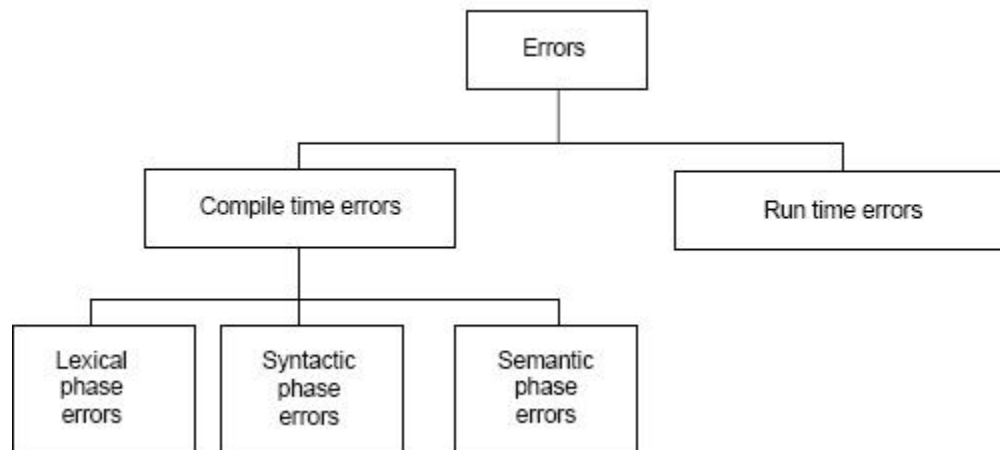


Figure 10.8 Classification of Errors

A good error detection scheme reports errors in an intelligible manner and should possess the following properties.

- ❑ The error message should be easily understandable.
- ❑ The error message should be produced in terms of original source program and not in any internal representation of the source program. For example, each error message should have a line number of the source program associated with it.
- ❑ The error message should be specific and properly localize the error. For example, an error message should be like, “A is not declared in function sum” and not just “missing declaration”.
- ❑ The same error message should not be produced again and again, that is, there is no redundancy in the error messages.

Error detection and recovery in lexical Phase: The errors where the remaining input characters do not form any token of the language are detected by the lexical phase of compiler. Typical lexical

phase errors are spelling errors, appearance of illegal characters and exceeding length of identifier or numeric constant.

Once an error is detected, the lexical analyzer calls an error recovery routine. The simplest error recovery routine skips the erroneous characters in the input until the lexical analyzer finds a synchronizing token. But this scheme causes the parser to have a deletion error, which would result in several difficulties for the syntax analysis and for the rest of the phases.

The ability of lexical analyzer to recover from errors can be improved by making a list of legitimate tokens (in the current context) which are accessible to the error recovery routine. With the help of this list, the error recovery routine can decide whether the remaining input characters match with a synchronizing token and can be treated as that token.

Error detection and recovery in syntactic phase: The errors where the token stream violates the syntax of the language and the parser does not find any valid move from its current configuration are detected during the syntactic phase of the compiler. The LL(1) and LR(1) parsers have valid prefix property capability, that is, they report an error as soon as they read an input character which is not a valid continuation of the previous input prefix. In this way, these parsers reduce the amount of erroneous output to be passed to next phases of the compiler.

To recover from these errors, *panic mode recovery* scheme or *phrase level recovery* scheme (discussed in chapter 04) can be used.

Error detection and recovery in semantic phase: The language constructs that have the right syntactic structure but have no meaning to the operation involved are detected during semantic analysis phase. Undeclared names, type incompatibilities and mismatching of actual arguments with formal arguments are the main causes of semantic errors. When an undeclared name is encountered first time, a symbol table entry is created for that name with an attribute that is suitable to the current context.

For example, if semantic phase detects an error like “missing declaration of A in function sum”, then a symbol table entry is created for A with an attribute that is suitable to the current context. To indicate that an attribute has been added to recover from an error and not in response to the declaration of A, a flag is set in the A symbol table record.

Code Optimization and Code Generation

1. Explain the various issues in the design of code generator.

Or

Discuss the various factors affecting the code generation process.

Ans: The various factors that affect the code generation process are as follows:

❑ **Input:** The intermediate code produced by the intermediate code generator or code optimizer of the compiler is given as input to the code generator. At the time of code generation, the source program is assumed to be scanned, parsed, and translated into a relatively low-level intermediate representation. Type conversion operators are assumed to be inserted wherever required, and that semantic errors have also been detected. The code generation phase, therefore, proceeds on the assumption that the input to the code generator is free from errors. We also assume that the operators, data types, and the addressing modes appearing in the intermediate representation can be directly mapped to the target machine representation. If such straightforward mappings exist, then the code generation is simple, otherwise a significant amount of translation effort is required.

❑ **Structure of target code:** The efficient construction of a code generator depends mainly on the structure of the target code which further depends on the instruction-set architecture of the target machine. RISC (reduced instruction set computer) and CISC (complex instruction set computer) are the two most common target machine architectures. The target program code may be absolute machine language code, relocatable machine language code, or assembly language code.

- If the target program code is absolute machine language code, then it can be placed in a fixed memory location and can be executed immediately. The fixed location of program variables and code makes the absolute code generation relatively easier.

- If the target program code is relocatable machine language code (also known as **object module**), then the code generation becomes a bit difficult as relocatable code may or may not be supported by the underlying hardware. In case the target machine does not support relocation automatically, it is the responsibility of compiler to explicitly insert the code for ensuring smooth relocation. However, producing a relocatable code requires subprograms to be compiled separately. After compilation, all the relocatable object modules can be linked together and loaded for execution by a linking loader.

- If the output is assembly language program, then it can be converted into an executable version by an assembler. In this case, the code generation can be made simpler by utilizing the features of assembler. That is, we can generate symbolic instruction code and use the macro facilities of the assembler to help the code generation process.

❑ **Selection of instruction:** The nature of the instruction set of the target machine is an important factor to determine the complexity of instruction selection. The uniformity and completeness of the instruction set, instruction speed, and machine idioms are the important factors that are to be considered. If we are not concerned with the efficiency of the target program, then instruction selection becomes easier and straightforward. The two important factors that determine the quality of the generated code are its speed and size.

For example, the three-address statement of the form,

$$A = B + C$$

$$X = A + Y$$

can be translated into a code sequence as given below:

LD R₀,B

```
ADD R0,R0,C
LD  R0,A
ADD R0,R0,Y
ST  X,R0
```

The main drawback of this statement by statement code generation is that it produces redundant load and store statements. For example, the fourth step in the above code is redundant as the value that has been stored just before is loaded again. If the target machine provides a rich set of instructions then there will be several ways of implementing a given instruction. For example, if the target machine has an increment instruction, $X = X + 1$, then instead of multiple load and store instructions, we can have simple instruction **INC X**. Note that deciding which machine-code sequence is suitable for a given set of three-address instructions may require knowledge about the context in which those instructions appear.

▣ **Allocation of registers:** Assigning the values to the registers is the key problem during code generation. So, generation of a good code requires the efficient utilization of registers. In general, the utilization of registers is subdivided into two phases, namely, register allocation and register assignment. **Register allocation** is the process of selecting a set of variables that will reside in CPU registers. **Register assignment** refers to the assignment of a variable to a specific register. Determining the optimal assignment of registers to variables even with single register values is difficult because the allocation problem is NP-complete. In certain machines, even/odd register pairs are required for some operands and results which make the problem further complicated. In integer multiplication, the multiplicand is placed in the odd register, however, the multiplier can be placed in any other single register, and the product (result) is placed in the entire even/odd register pair. Register allocation becomes a nontrivial task because of these architecture-specific issues.

▣ **Evaluation order:** The performance of the target code is greatly affected by the order in which computations are performed. For some computation order, only a fewer registers are required to hold the intermediate results. Hence, deciding the optimal computation order is again difficult since the problem is NP-complete. The problem can be avoided initially by generating the code for the three-address statements in the same order as that of produced by the intermediate code generator.

2. Define basic block.

Ans: A **basic block** is a sequence of consecutive three-address statements in which the flow of control enters only from the first statement of the basic block and once entered, the statements of the block are executed without branching, halting, looping or jumping except at the last statement. The control will leave the block only from the last statement of the block. For example, consider the following statements.

```
t1: = X * Y
t2: = 5 * t1
t3: = T1 * t2
```

In the above sequence of statements, the control enters only from the first statement, $t_1 := X * Y$. The second and third statements are executed sequentially without any looping or branching and the control leaves the block from the last statement. Hence, the above statements form a basic block.

3. Write the steps for constructing leaders in a basic block.

Or

How can you find leaders in basic blocks?

Ans: The first statement in the basic block is known as the leader. The rules for finding leaders are as follows:

- (i) The first statement in the intermediate code is leader.
- (ii) The target statement of a conditional and unconditional jump is a leader.
- (iii) The immediate statement following an unconditional or conditional jump is a leader.

4. Write an algorithm for partitioning of three-address instructions into a basic block. Give an example also.

Ans: A sequence of three-address instructions is taken as input and the following steps are performed to partition the three-address instructions into basic blocks:

Step 1: Determine the set of leaders.

Step 2: Construct the basic block for each leader that consists of the leader and all the instructions till the next leader (excluding the next leader) or the end of the program.

The instructions that are not included in a block are not executed and may be removed, if desired. For example, consider the following code segment that computes a dot product between two integer arrays X and Y.

```
begin
    PRODUCT := 0
    j := 1
    do
        begin
            PRODUCT := PRODUCT + X[j] * Y[j]
            j := j + 1
        end
    while j <= 20
end
```

The corresponding three-address code for the above code segment is given as follows:

1. PRODUCT := 0
2. j := 1
3. $t_1 := 4 * j$ /* assuming that the elements of integer array take 4 bytes */
4. $t_2 := X[t_1]$ /* computing X[j] */
5. $t_3 := 4 * j$
6. $t_4 := Y[t_3]$ /* computing Y[j] */
7. $t_5 := t_2 * t_4$ /* computing X[j] * Y[j] */
8. $t_6 := \text{PRODUCT} + t_5$
9. PRODUCT := t_6
10. $t_7 := j + 1$
11. j := t_7
12. if j <= 20 goto (3)

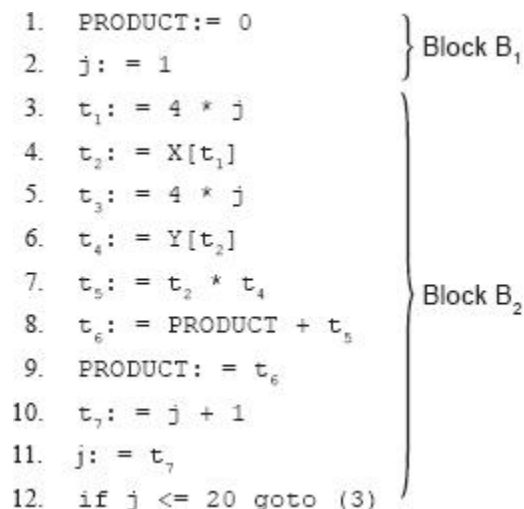


Figure 11.1 Basic Blocks

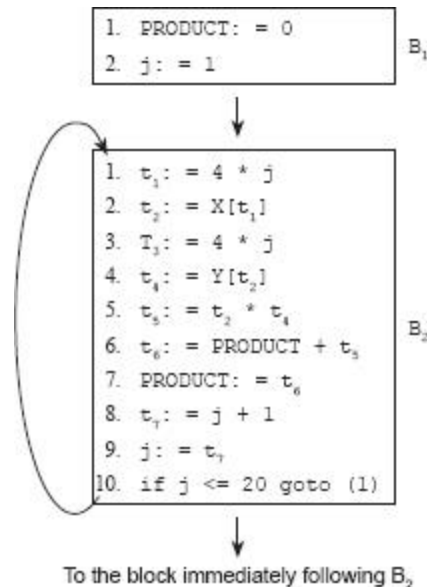


Figure 11.2 Flow Graph

Now, we can determine the basic blocks of the above three-address code by following the previous algorithm. Considering the rules for finding the leaders, according to rule (i) statement (1) is a leader. According to rule (ii), statement (3) is a leader. According to rule (iii), the statement following the 12th statement, if any, is a leader. Hence, the statements (1) and (2) form the first basic block and the rest of the program starting with statement (3) forms the second basic block as shown in Figure 11.1.

5. Explain the role of flow graph in basic blocks.

Ans: A flow graph is a directed graph that represents flow of control between the basic blocks. The basic block represents the nodes of the graph, and the edges define the control transfer. The flow graph for the program given in previous question is given in Figure 11.2.

If block B₂ immediately follows block B₁ then, there is an edge from block B₁ to B₂. B₁ is said to be the predecessor of B₂ and B₂ is said to be the successor of B₁ if, any of the following two conditions are satisfied.

- ❑ There is an unconditional or conditional jump from the last instruction of the block B₁ to the starting instruction of block B₂.
- ❑ B₁ does not end in an unconditional jump and block B₁ is immediately followed by block B₂

To the block immediately following B₂ Figure 11.2 Flow Graph

6. Explain code optimization. What are the objectives of the code optimization?

Ans: Code optimization is an attempt of compiler to produce a better object code (target code) with high execution efficiency than the input source program. In some cases, code optimization can be so simple that it can be carried out without much difficulty. However, in some other cases, it may require a complete analysis of the program. Code optimization may require various transformation of the source program. These transformations should be carried out in such a way that all the translations of a source program are semantic equivalent, as well as the algorithm should not be modified in any case. The efficiency and effectiveness of a code optimization technique is determined by the time and space required by the compiler to produce the target program. Code optimization can be machine dependent or machine independent (discussed in Question 15).

Objectives of the Code Optimization:

- ❑ Production of target program with high execution efficiency.

- ❑ Reduction of the space occupied by the program.
- ❑ Time efficient program that takes lesser compilation time.

7. Write a short note on optimizing transformations.

Ans: Optimizing transformations are of two types, namely, *local* and *global*. Local optimization is performed on each basic block and global optimization is applied over the large segments of a program consisting of loops or procedure/function. Local optimization involves transforming a basic block into a DAG and global optimization involves the data flow analysis. Though local optimization requires less amount of code analysis, however, it does not allow all kinds of code optimizations (for example, loop optimizations cannot be performed locally). However, local optimization can be merged with the initial phase of global optimization to simplify the global optimization.

8. What is constant folding?

Ans: Constant folding is used for code optimization. It evaluates the constant expressions during compile time and replaces the expressions by their values. For example, consider the following statements:

X: = 5 + 2

A: = X + 2

After constant folding, the statements can be written as follows:

X: = 7

A: = 9

9. Explain loop optimization.

Ans: Loop optimization is a technique in which inner loops are taken into consideration for the code optimization. Only the inner loops are considered because a large amount of time is taken during the execution of these inner loops. The various loop optimization techniques are *loop-invariant expression elimination*, *induction variable elimination*, *strength reduction*, *loop unrolling* and *loop fusion*.

❑ **Loop-invariant expression elimination:** An expression is said to be *loop-invariant* expression if it produces the same result each time the loop is executed. During loop-invariant expression elimination, we eliminate all such expressions by placing them at the entry point of the loop. For example, consider the following code segment:

```
if (i > Min + 2)
{
    sum = sum + x[i];
}
```

In this code segment, the expression `Min + 2` is evaluated each time the loop is executed, however, it always produces the same result irrespective of the iteration of the loop. Thus, we can place this expression at the entry point of the loop as follows:

```
n = Min + 2
if (i > n)
{
    sum = sum + x[i];
}
```

Since in loop-invariant expression elimination, the expression from inside the loop is moved outside the loop, this method is also known as **loop-invariant code motion**.

❑ **Induction variable elimination:** A variable is said to be *induction variable* if its value gets incremented or decremented by some constant every time the loop is executed. For example, consider the following for loop statement:

```
for (j = 1; j <= 10; j++)
```

Here, the value of *j* is incremented every time the loop is executed. Hence, *j* is an induction variable. If there are more than one induction variables in a loop then, it is possible to get rid of all but one. This process is known as induction variable elimination.

❑ **Strength reduction:** Replacing an expensive operation with an equivalent cheaper operation is called **strength reduction**. For example, the * operator can be replaced by a lower strength operator +. Consider the following code segment:

```
for (j = 1; j <= 10; j++)
{
    ...
    cnt = j * 5;
    ...
}
```

After strength reduction, the code can be written as follows:

```
temp = 5;
for (j = 1; j <= 10; j++)
{
    ...
    cnt = temp;
    temp = temp + 5;
    ...
}
```

❑ **Loop unrolling:** The number of jumps can be reduced by replicating the body of the loop if the number of iterations is found to be constant (that is, the number of iterations is known at compile time). For example, consider the following code segment:

```
int j = 1;
while (j <= 50)
{
    x[j] = 0;
    j = j + 1;
}
```

This code segment performs the test 50 times. The number of tests can be reduced to 25 by replicating the code inside the body of the loop as follows:

```
int j = 1;
while (j <= 50)
{
    x[j] = 0;
    j = j + 1;
    x[j] = 0;
    j = j + 1;
}
```

The main problem with loop unrolling is that if the body of the loop is big, then unrolling may increase the code size, which in turn, may affect the system performance.

❑ **Loop fusion:** It is also known as **loop jamming** in which the bodies of the two loops are merged together to form a single loop provided that they do not make any references to each other. For example, consider the following statements:

```
int i,j;
for(i = 1; i <= n; i++)
    A[i] = B[i];
for(j = 1; j <= n; j++)
    C[j] = A[j];
```

After loop fusion, this code can be written as follows:

```
int i,j;
for(k = 1; k <= n; k++)
{
    A[k] = B[k];
    C[k] = A[k];
}
```

10. Define DAG (Directed Acyclic Graph). Discuss the construction of DAG of a given basic block.

Ans: A DAG is a directed acyclic graph that is used to represent the basic blocks and to implement transformations on them. It represents the way in which the value computed by each statement in a basic block is used in the subsequent statements in the block. Every node in a flow graph can be represented by a DAG. Each node of a DAG is associated with a label. The labels are assigned by using these rules:

- ❑ The leaf nodes are labeled by unique identifiers which can be either constants or variable names. The initial values of names are represented by the leaf nodes, and hence they are subscripted with 0 in order to avoid confusion with labels denoting current values of names.
- ❑ An operator symbol is used to label the interior nodes.
- ❑ Nodes are also labeled with an extra set of identifiers where the interior nodes represent the computed values and the identifier labeling that node contains the computed value.

The main difference between a flow graph and a DAG is that a flow graph consists of several nodes where each node stands for a basic block, whereas a DAG can be constructed for each node (or the basic block) in the flow graph.

Construction of DAG

While constructing a DAG, we consider a function node(identifier) which returns the most recently created node associated with an identifier. Assume that there are no nodes initially and node() is undefined for all arguments. Let the three-address statements be either

- (i) $X = Y \text{ op } Z$ or
- (ii) $X = \text{op } Y$ or
- (iii) $X = Y$

The steps followed for the construction of DAG are as follows:

1. Create a leaf labeled Y if node(Y) is undefined and let that node be node(Y). If node(Z) is undefined for three-address statement (i) then, create a leaf labeled Z and let it be node(Z).
2. Determine if there is any node labeled op, where node(Y) as its left child and node(Z) as its right child for three-address statement (i). If such a node is not found, then create such a node and let it be n. For three-address statement (ii), determine if there is any node labeled op, whose only child is node(Z). If such a node is not found then, create a node and let it be n. Similarly, create a node n for node(Y) for three-address statement (iii).

3. Delete X from the list of attached identifiers for node(X). Append X to the list of attached identifiers for node n created or found in step 2 and set node(X) to n.

For example, consider the block B₂ shown in Figure 11.2. For the first statement, $t_1 := 4 * j$, leaves labeled 4 and j are created. In the next step, a node labeled * is created and t₁ is attached to it as an identifier. The DAG representation is shown in Figure 11.3(a).

For the second statement $t_2 := X[t_1]$, a new leaf labeled X is created. Since, we have already created node(t₁) in the previous step, we will not create a new node t₁. However, we create a new node for [], and attach X and t₁ as its child nodes. Now, the third statement $t_3 := 4 * j$ is same as that of first statement, therefore, we will not create any new node; rather we give the existing * node an additional label t₃. The DAG representation for this is shown in Figure 11.3(b).

For the fourth $t_4 := Y[t_3]$ we create a node [] and attach Y as its left child node. The corresponding DAG representation is shown in Figure 11.3(c). For the fifth statement $t_5 := t_2 + t_4$, we create a new node + and attach the already created nodes labeled t₄ and t₂ as its left and right child respectively. The resultant DAG is shown in Figure 11.3(d).

For the sixth statement, we create a new node labeled +, and attach a leaf labeled PRODUCT0 as its left child. The already created node(*) is attached as its right child. For the seventh statement, $PRODUCT := t_6$, we assign the additional label PRODUCT to the existing + node. The resultant DAG is shown in Figure 11.3(e).

For the eighth statement $t_7 := j + 1$, we create a new node labeled +, and make j_0 as its left child. Now, we create a new leaf labeled 1 and make this leaf as its right child. For the ninth statement, we will not create any new node; rather we give node + the additional label j . Finally, for the last statement we create a new node labeled \leq and attach an identifier (1) with it. Now, we create a new leaf labeled 20 and make this node as the right child of node(\leq). The left child of this node is node(+). The final DAG is shown in Figure 11.3(f).

or

- ❑ It helps in determining the instructions that compute a value which is never used. It is referred to as **dead code elimination**.

- ❑ It provides a way to determine those names which are evaluated outside the block, however, are used inside the blocks.
- ❑ It helps in determining those statements of the block which could have their computed values used outside the block.
- ❑ It helps in determining those statements which are independent of one another and hence, can be reordered.

12. Give the primary structure-preserving transformations on basic blocks.

Ans: The primary structure-preserving transformations on basic blocks are as follows:

❑ **Common subexpression elimination:** Transformations are performed on basic blocks by eliminating the common subexpressions. For example, consider the following basic block:

```
X: = Y * Z
Y: = X + A
Z: = Y * Z
A: = X + A
```

In the given basic block, the right side of the first and third statement appears to be same, however, $Y * Z$ is not a common subexpression because the value of Y has been modified in the second statement. The right side of the second and fourth statement is also same, and the value of X is not modified, so we can replace the $X + A$ by Y in the fourth statement. Now, the equivalent transformed block can be written as follows:

```
X: = Y * Z
Y: = X + A
Z: = Y * Z
A: = Y
```

❑ **Dead code elimination:** A variable is said to be dead (useless) at a point in a program if its value cannot be used subsequently in the program. Similarly, a code is said to be dead if the value computed by the statements is never get used. Elimination of dead code does not affect the program behavior. For example, consider the following statements:

```
flag: = false
If (flag) print some information
```

Here, the print statement is dead as the value of flag is always *false* and hence the control never reaches to the print statement. Thus, the complete if statement (test and the print operation) can be eliminated easily from the object code.

❑ **Renaming temporary variables:** A statement of the form $t_1 := X + Y$, where t_1 is a temporary variable can be changed to a statement $t_2 := X + Y$ where, t_2 is a new temporary variable. Thus, all the instances of t_1 can be changed to t_2 without affecting the value of the block.

❑ **Interchange of statements:** Two statements can be interchanged in the object code if they make no references to each other, and their order of execution does not affect the value of the block. For example, consider the following statements:

```
t1: = A + B
t2: = X + Y
```

If neither X nor Y is t_1 and neither A nor B is t_2 , then the two statements can be interchanged without affecting the value of the block.

❑ **Code motion:** Moving the code from one part of the program to another so that the resultant program is equivalent to the original one is referred to as code motion. Code motion is performed to reduce the size of the program and to reduce the execution frequency of the code which is moved. For example, consider the following code segment:

```
if (x < y)
    result: = x * 2
else
    result: = x * 2 + 50
```

In this code segment, the subexpression $x * 2$ is evaluated twice, thus, it can be moved before if statement as shown below:

```
temp: = x * 2
if (x < y)
    result: = temp
else
    result: = temp + 50
```

❑ **Variable propagation:** In variable propagation, a variable is replaced by another variable having identical value. For example, consider the following statements:

```
X: = Y
A: = X * B
C: = Y * B
```

The statement $X: = Y$ specifies that the values of X and Y are equal. Since the value of X or Y is not modified further, the second statement can hence be written as $A: = Y * B$ by propagating the variable Y to it. This propagation makes $Y * B$ as a common subexpression in the last two statements, and hence possibly evaluated only once.

❑ **Algebraic transformations:** In algebraic transformations, the algebraic identities are used to optimize the code. Some of the common algebraic identities are given below:

```
A + 0 = A (Additive identity)
A * 1 = A (Multiplicative identity)
A * 0 = 0 (Multiplication with 0)
```

These identities are generally applied on a single intermediate code statement. For example, consider the following statements:

```
Y: = X + 0
Y: = X * 1
Y: = X * 0
```

After algebraic transformations, the expensive addition and multiplication operations involved in these statements can be replaced by cheaper assignment operations as given below:

```
Y: = X
Y: = X
Y: = 0
```

❑ **Induction variables and strength reduction:** Refer Question 9

13. Discuss in detail about a simple code generator with the appropriate algorithm.

Or

Explain code generation phase with simple code generation algorithm.

Ans: A simple code generator generates the target code for the three-address statements. The main issue during code generation is the utilization of registers since the number of registers available is

limited. The code generation algorithm takes the sequence of three-address statements as input and assumes that for each operator, there exists a corresponding operator in target language. The machine code instruction takes the required operands in registers, performs the operation and stores the result in a register. Register and address descriptors are used to keep track of register contents and addresses.

❑ **Register descriptors** are used to keep track of the contents of each register at a given point of time. Initially, we assume that a register descriptor shows that all registers are empty and as the code generation proceeds, each register holds the value of zero or more names at some point.

❑ **Address descriptors** are used to trace the location of the current value of the name at run time. The location may be memory address, register, or a stack location, and this information can be stored in the symbol table to determine the accessing method for a name.

The code generation algorithm for a three-address statement $X := Y \text{ op } Z$ is given below:

1. Call `getreg()` to obtain the location L where the result of $Y \text{ op } Z$ is to be stored. L can be a register or a memory location.
2. Determine the current location of Y by consulting the address descriptor for Y and let it be Y' . If both the memory and register contains the value of Y , then prefer the register for Y' . If the value is not present in L , then generate an instruction `MOV Y', L`.
3. Determine the current location of Z , say, Z' and generate the instruction `OP Z', L`. In this case also, if both the memory and the register hold the value of Z , then prefer the register. Update the address descriptor of X to indicate that X is in L and if L is a register then, update its descriptor indicating that it holds the value of X . Delete X from other register descriptors.
4. If the current values of Y and/or Z are in registers, and if they have no further uses and are not live at the end of the block, then alter the register descriptor. This alteration indicates that Y and/or Z will no longer be present in those registers after the execution of $X := Y \text{ OP } Z$.

For the three-address statement $X := \text{OP } Y$, the steps are analogous to the above steps. However, for the three-address statement of the form $X := Y$, some modifications are required as discussed here.

❑ If Y is in a register then the register and address descriptors are altered to record that from now onwards the value of X is found only in the register that holds the value of Y .

❑ If Y is in the memory, the `getreg()` function is used to determine a register in which the value of Y is to be loaded, and that register is now made as the location of X .

Thus, the instruction of the form $X := Y$ could cause the register to hold the value of two or more variables simultaneously.

Implementing the Function `getreg`

For the three-address statement, $X := Y \text{ OP } Z$, the function `getreg()` returns a location L as follows:

1. If Y is in a register and it is not live and has no next uses after the execution of three-address statement, then return the register of Y for L . The address descriptor of Y is then updated to indicate that Y is no more present in L .
2. If Y is not in a register, then return an empty register for L (if exists).
3. If X is to be used further in the block or OP is the operator that requires a register, then find an occupied register R_0 that may contain one or more values. For each variable in R_0 , issue a `MOV R0, M` instruction to store the value of R_0 into a memory location M . Then, update the address descriptor for M , and return R_0 . Though there are several ways to choose a suitable occupied register, the simplest way is to choose the one whose data values are to be referenced furthest in the future.

4. If X is not used in the block or an occupied register could not be found, then select the memory location of X as L.

14. Explain the peephole optimization and its characteristics.

Ans: Peephole optimization is an efficient technique for optimizing either the target code or the intermediate code. In this technique, a small portion of the code (known as peephole) is taken into consideration and optimization is done by replacing the code by the equivalent code with shorter or faster sequence of execution. The statements within the peephole need not be contiguous, although some of the implementations require the statements to be contiguous. Each improvement in the code may explore the opportunities for some other improvements. So, multiple review of the code is necessary to get maximum benefit from the peephole optimization. Some characteristics of the peephole optimization are: *redundant- instruction elimination, unreachable code elimination, flow of control optimizations, strength reduction and use of machine idioms.*

❑ **Redundant-instruction elimination:** Consider the following instructions:

```
MOV R0, X
MOV X, R0
```

The second instruction can be deleted since the first instruction ensures that the value of X is already loaded into register R₀. However, it cannot be deleted in a situation when, it has a label which makes it difficult to identify that whether the first instruction is always executed before the second. To ensure that this kind of transformation in the target code would be safe, the two instructions must be in the same basic block.

❑ **Unreachable code elimination:** Removing an unlabeled instruction that immediately follows an unconditional jump is possible. This process eliminates a sequence of instructions when repeated. Consider the following intermediate code representation:

```
if error == 1 goto L1
goto L2
```

L1: Print error information

L2:

Here, the code is executed only if the variable error is equal to 1. Peephole optimization allows the elimination of jumps over jumps. Hence, the above code is replaced as follows irrespective of the value of the variable debug.

```
if error != 1 goto L2
```

Print error information

L2:

Now, if the value of the variable is set to 0, the code becomes:

```
if 0 != 1 goto L2
```

Print error information

L2:

Here, the first statement always evaluates to *true*. Hence, the statement printing the error information is unreachable and can be eliminated.

❑ **Flow of control optimizations:** The peephole optimization helps to eliminate the unnecessary jumps in the intermediate code. For example, consider the following code sequence:

```
goto L1
```

```
...
```

```
L1: goto L2
```

This sequence can be replaced by

```
goto L2
```


...

L1: goto L2

Now, if there are no jumps to L_1 and the statement $L_1: \text{goto } L_2$ is preceded by an unconditional jump, then this statement can be eliminated. Similarly, consider the following code sequence:

if (x < y) goto L1

...

L1: goto L2

This sequence can be rewritten as follows:

if (x < y) goto L2

...

L1: goto L2

❑ **Strength reduction:** Peephole optimization also allows applying strength reduction transformations to replace expensive operations by the equivalent cheaper ones. For example, the expression X_2 can be replaced by an equivalent cheaper expression $X * X$.

❑ **Use of machine idioms:** Some target machines provide hardware instructions to implement certain operations in a better and efficient way. Thus, identifying the situations that permit the use of hardware instructions to implement certain operations may reduce the execution time significantly. For example, some machines provide auto-increment and auto-decrement addressing modes, which add and subtract one respectively from an operand. These modes can be used while pushing or popping a stack, or for the statements of the form $X := X + 1$ or $X := X - 1$. These transformations greatly improve the quality of the code.

15. Explain the machine-dependent and machine-independent optimization.

Ans: Machine-dependent optimizations: An optimization is called *machine-dependent optimization* if it requires knowledge of the target machine to perform optimization. A machine-dependent optimization utilizes the target system registers more efficiently than the machine-independent optimization. Often, machine-dependent optimizations are local and are considered as most effective for the local machine as these optimizations best exploit the features of target platform.

Machine-independent optimizations: The optimization which is not intended for a target machine specific platform and optimizations can be carried out independently of the target machine is known as *machine-independent optimizations*. Machine-independent optimizations can be both local and global and operates on abstract programming concepts (like loops, objects, and structures).

16. Discuss the value number method for constructing a DAG. How does it help in code optimization?

Ans: While constructing a DAG, we check whether a node with given operator and given children exist or not. If such a node does not exist then we create a node with that given operator and given children. To determine the existence of a node instantly, we can use a hash table. This idea of using a hash table is named as **value number method** by Cocke and Schwartz [1970]. Basically, the nodes of a DAG are stored in the form of array of records, where each record in the array corresponds to a node in the DAG. Each record consists of several fields, where the first field is always an operation code, which indicates the label of the node. The leaves have one more field holding the lexical value which may be a symbol-table pointer or a constant. The interior nodes have two more fields, which indicate the left and right child nodes. An array representation of a DAG shown in Figure 11.4(a) is shown in Figure 11.4(b).

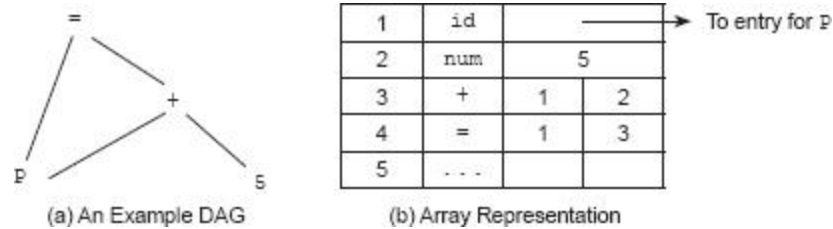


Figure 11.4 Value Number Method

In this array, the nodes are referred to by giving the integer index (called the value number) of the record for that node within the array. For instance, in Figure 11.4(b), the node labeled = has value number 4.

The value number method can also be used to implement certain optimizations based on algebraic laws (like *commutative*, *associative*, and *distributed laws*). For example, if we want to create a DAG node with its left child p and right child q, and operator *, we first check whether such a node exists by using value number method. As multiplication is commutative in nature, therefore, we also need to check the existence of a node labeled *, with its left child q and right child p.

The associative law can also be applied to improve the already generated code from a DAG. For example, consider the following statements:

P: = Q + R

S: = R + T + Q

The three-address code for these statements is given below:

1. $t_1 := Q + R$

2. $P := t_1$

3. $t_2 := R + T$

4. $t_3 := Q + t_2$

5. $S := t_3$

The DAG for this code is shown in Figure 11.5(a). If we assume that t_2 is not needed outside the block, then the DAG shown in Figure 11.5(a) can be changed to the one shown in Figure 11.5(b). Here, both the associative and commutative laws are used.

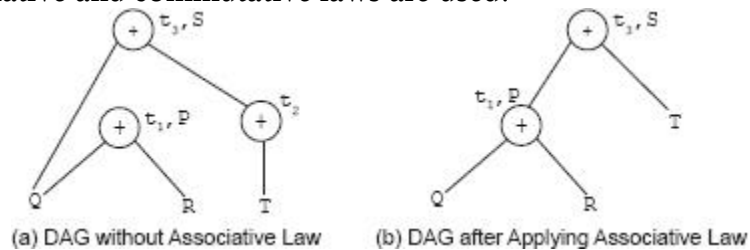


Figure 11.5 Use of Associative Law