Intermediate Code Generation

Introduction

In the first pass of the compiler, source program is converted into intermediate code. The second pass converts the intermediate code to target code. The intermediate code generation is done by intermediate code generation phase. It takes input from front end which consists of lexical analysis, syntax analysis and semantic analysis and generates intermediate code and gives it to code generator. Fig 6.1 shows the position of intermediate code generator in compiler. Although some source code can be directly converted to target code, there are some advantages of intermediate code. Some of these advantages are:

a. Target code can be generated to any machine just by attaching new machine as the back end. This is called retargeting.

b. It is possible to apply machine independent code optimization. This helps in faster generation of code.

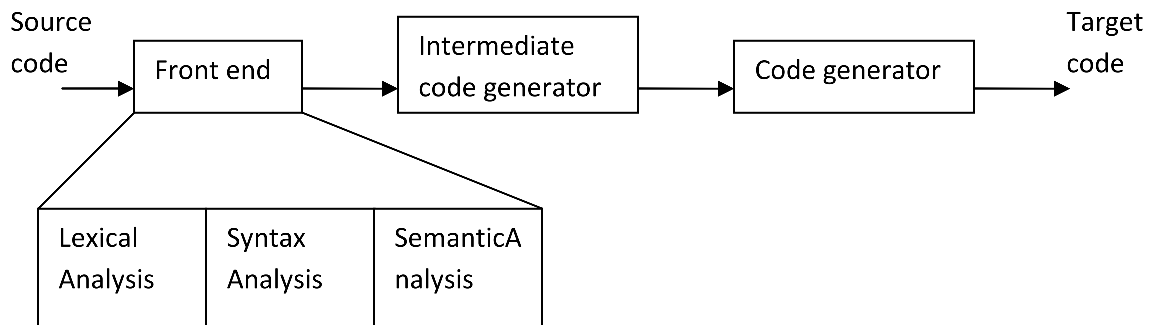


Fig 6.1 position of intermediate code generator in compiler

This chapter deals with the intermediate representation in the form of three address code. Other forms of intermediate representations are syntax tree, postfix notation or Directed Acyclic Graph (DAG). The semantic rule for syntax tree and three address code are almost similar.

Graphical and Linear representation

Intermediate representation can be either in linear or graphical form. Graphical form includes syntax tree and DAG where as linear representation may be postfix notation or three address code. Fig 6.2 shows the syntax tree for the expression a = b * c.

Syntax tree

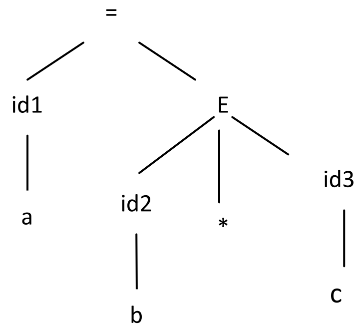


Fig 6.2 syntax tree for the expression a = -b * c

Directed Acyclic Graph (DAG)

For Example a = -b * C

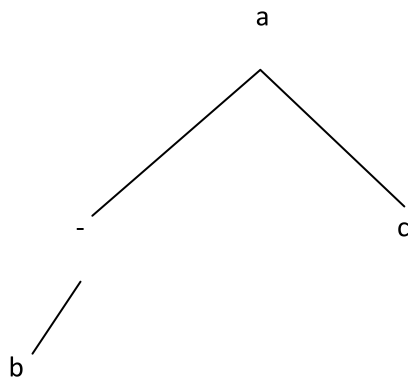


Fig 6.3 DAG for the expression a = -b * c

Linear representation: Postfix and three address code are the two forms of linear representation of any expression.

Postfix notation for the expression a = -b * c is b-c*a=

Three address code for the expression a = - b * c is

t1= -b

t2 = t1 * c

a = t2

Intermediate code can be of many forms. They can be either

- Language specific like P-code for Pascal, byte code for Java etc or
- Specific to machine on which implementation is performed or
- Independent of language being implemented and target machine

The three address code that we are considering here for intermediate representation is Independent of language being implemented and target machine.

Three address code

Most instruction of three address code is of the form

a = b op c

where b and c are operands and op is an operator. The result after applying operator op on b and c is stored in a. Operator op can be like +, -, * or ÷. Here operator op is assumed as binary operator. The operands b and c represents the address in memory or can be constants or literal value with no runtime address. Result a can be address in memory or temporary variable.

Example: a = b * c + 10

The three address code will be

t1 = b * C

t2 = t1 + 10

a = t2

Here t1 and t2 are temporary variables used to store the intermediate result.

Types of three address code

There are different types of statements in source program to which three address code has to be generated. Along with operands and operators, three address code also use labels to provide flow of control for statements like if-then-else, for and while. The different types of three address code statements are:

i. **Assignment statement** a = b op c

In the above case b and c are operands, while op is binary or logical operator. The result of applying op on b and c is stored in a.

ii. **Unary operation** a = op b

This is used for unary minus or logical negation.

Example: a = b * (- c) + d

Three address code for the above example will be

t1 = -c

t2 = t1 * b

t3 = t2 + d

a = t3

iii. **Copy Statement** a = b

The value of b is stored in variable a.

iv. **Unconditional jump** goto L

Creates label L and generates three-address code 'goto L'

v. **Conditional jump** if exp go to L

Creates label L, generate code for expression exp, If the exp returns value true then go to the statement labelled L. exp returns a value false go to the statement immediately following the if statement.

vi. **Function call** For a function fun with n arguments a1,a2,a3….an ie.,

fun(a1, a2, a3,…an),

the three address code will be

Param a1

Param a2

…

Param an

Call fun, n

Where param defines the arguments to function.

vii. **Array indexing**- In order to access the elements of array either single dimension or multidimension, three address code requires base address and offset value. Base address consists of the address of first element in an array. Other elements of the array can be accessed using the base address and offset value.

Example: x = y[i]

Memory location m = Base address of y + Displacement i

x = contents of memory location m

similarly x[i] = y

Memory location m = Base address of x + Displacement i

The value of y is stored in memory location m

viii. **Pointer assignment** x = &y x stores the address of memory location y

x = *y y is a pointer whose r-value is location

*x = y sets r-value of the object pointed by x to the r-value of y

Intermediate representation should have an operator set which is rich to implement most of the operations of source language. It should also help in mapping to restricted instruction set of target machine.

Data Structure

Three address code is represented as record structure with fields for operator and operands. These records can be stored as array or linked list. Most common implementations of three address code are- Quadruples, Triples and Indirect triples.

Quadruples- Quadruples consists of four fields in the record structure. One field to store operator **op**, two fields to store operands or arguments **arg1**and **arg2** and one field to store result **res**. **res = arg1 op arg2**

Example: a = b + c

b is represented as arg1, c is represented as arg2, + as op and a as res.

Unary operators like '-'do not use agr2. Operators like param do not use agr2 nor result. For conditional and unconditional statements res is label. Arg1, arg2 and res are pointers to symbol table or literal table for the names.

Example: a = -b * d + c + (-b) * d

Three address code for the above statement is as follows

t1 = - b

t2 = t1 * d

t3 = t2 + c

t4 = - b

t5 = t4 * d

t6 = t3 + t5

a = t6

Quadruples for the above example is as follows

Op	Arg1	Arg2	Res
-	B		t1
*	t1	d	t2
+	t2	c	t3
-	B		t4
*	t4	d	t5
+	t3	t5	t6
=	t6		a

Triples – Triples uses only three fields in the record structure. One field for operator, two fields for operands named as arg1 and arg2. Value of temporary variable can be accessed by the position of the statement the computes it and not by location as in quadruples.

Example: a = -b * d + c + (-b) * d

Triples for the above example is as follows

Stmt no	Op	Arg1	Arg2
(0)	-	b	
(1)	*	d	(0)
(2)	+	c	(1)
(3)	-	b	
(4)	*	d	(3)
(5)	+	(2)	(4)
(6)	=	a	(5)

Arg1 and arg2 may be pointers to symbol table for program variables or literal table for constant or pointers into triple structure for intermediate results.

Example: Triples for statement x[i] = y which generates two records is as follows

Stmt no	Op	Arg1	Arg2
(0)	[]=	x	i
(1)	=	(0)	y

Triples for statement x = y[i] which generates two records is as follows

Stmt no	Op	Arg1	Arg2
(0)	=[]	y	i
(1)	=	x	(0)

Triples are alternative ways for representing syntax tree or Directed acyclic graph for program defined names.

Indirect Triples – Indirect triples are used to achieve indirection in listing of pointers. That is, it uses pointers to triples than listing of triples themselves.

Example: a = -b * d + c + (-b) * d

	Stmt no
(0)	(10)
(1)	(11)
(2)	(12)
(3)	(13)
(4)	(14)
(5)	(15)
(6)	(16)

Stmt no	Op	Arg1	Arg2
(10)	-	b	
(11)	*	d	(0)
(12)	+	c	(1)
(13)	-	b	
(14)	*	d	(3)
(15)	+	(2)	(4)
(16)	=	a	(5)

When target code is generated each program variable and temporary variable is assigned a memory location. This address of the location is stored in symbol table. Quadruples use this address for output generation. If an assignment statement for b is moved from one location to another (inter change the instruction), it requires no change for regeneration of intermediate code. This is because it is possible to access symbol table for program variable or temporary variable directly. In case of triples, as there are no temporary variables stored in symbol table and all references are only to the position of statement and not location, the compiler should change all references to arg1 and arg2. Thus triples are not very efficient in optimizing compilers. In case of indirect triples, no pointer refers to temporary variable, hence no change in pointer required when instructions are interchanged. The statements can be moved by reordering the statement list.

Both indirect triples and quadruples give almost performance with respect to space and reordering code. However indirect triples can be space saving if temporary variables are reused. For the previous example statement no (13) can be removed and the value of statement no (1) can be reused.

Basic Intermediate Code Generation Technique

Program consists of assignment statements like a=b op c or control statements like if-then-else, while loop or for statements. This section deals with generation of three address code for assignment statement and control statements.

Assignment statement

This section deals with the generation of intermediate code for assignment statement. It describes the way in which symbol table can be searched for an identifier. Identifiers can be simple variable or single or multidimensional array or a constant value (stored in literal table). Next step is generation of three address code for the program statement.

- **Searching in symbol table**

During the process of generation of intermediate code, symbol table has to be searched for identifier. The lexeme for identifier is stored in variable id.name. Searching for identifier in symbol table is achieved through the function search(). Search() returns the pointer of identifier in symbol table, if id.name has an entry in symbol table. If search fails it returns null indicating id.name not found.

- **Generate code**

Intermediate code generator uses function called produce() to generate three address code and store it in output file. It also uses a variable E.value to store the name of E that holds the vale of E. All the intermediate results are to be stored in temporary variables. Hence the function newtemp() is used. It generates new temporary variables like t1,t2,… every time newtemp() is called.

Example: Consider the following grammar for assignment statement

S → id=E

E → E1 + E2

E → E1 * E2

E → -E1

E → (E1)

E → id

Translation scheme to produce three address code is as follows

Production	Translation rules

S → id=E	x = search(id.name);
	if x ≠ null then produce(x '=' E.value)
	else
	error

E → E1 + E2	E.value = newtemp() Produce(E.value '=' E1.value '+' E2.value)
E → E1 * E2	E.value = newtemp() Produce(E.value '=' E1.value '*' E2.value)
E → -E1	E.value = newtemp() Produce(E.value '=' '-' E1.value)
E → (E1)	E.value = E1.value
E → id	x = search(id.name) if x ≠ null then E.value = x else error

Example: Generate three address code or the following arithmetic expression

a = -b * c

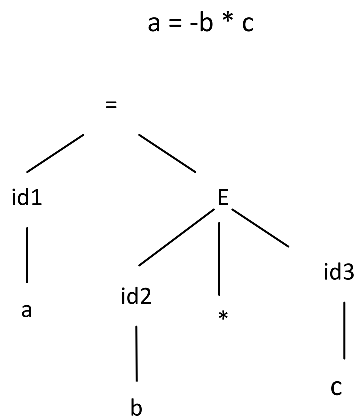


Fig 6.4 Syntax tree for the expression a = -b * c

Consider the Fig 6.4 which shows the syntax tree for the expression a = -b * c

1. Using the production E → id i.e.. E → b Check for entry b in the symbol table if not present display error msg., if present made E.value = x i.e., E.value = b (pointer to symbol table for the entry b)

2. Create a temporary variable t_1 with production 4, this produces an intermediate code

t1 = -b

3. Same as step 1, searches for entry c in symbol table & assigns E.value = c (pointer to symbol table for the entry c)

4. Using the production E → E_1 * E_2 it creates temporary variable t2 using newtemp(). Three address code E.value = E_1.value + E_2.value generates

$t_2 = t_1 +$ **c**

5. Using the production S → id = E searches for **a** in symbol table, assuming it is stored produces code x = E.value

$a = t_2$

Reusing temporary variables

The newtemp() function is used to create new temporary variable to store intermediate results. If the number of temporary variables increases to a large value, it becomes difficult to maintain these variables. In some cases the temporary variables values may not be required until the end. So, the memory reserved for theses variables goes unused, in order to use memory efficiently and reduce the number of temporary variables, reusing of temporary variables is done. The newtemp() function has to be modified to achieve this. Instead of always generating new temporary variables every time a newtemp() is called, it should find those temporary variable whose use is completed in the subsequent code, then the intermediate results can be stored is these temporary variables ie they are reused.

The other significance of reusing temporary is that, there can be some sub expressions being repeated in the whole expression. Instead of creating new temporary variable for all sub expressions, it can be reused.

Example: a = -b * c + (-b) * C

The three address code will be as follows:

(1) $t_1 = -b$

(2) $t_2 = t_1 + c$

(3) $t_3 = t_2 + d$

(4) $a = t_3$

after the statement (2) t_1 is not used hence instead of creating new. Temporary t_3, t_1 can be reused. Hence generates following code

⇓

(1) t_1 = - b

(2) $t_2 = t_1 + c$

(3) $t_1 = t_2 + d$

(3) a = t_1

number of temporary variables used is 2 instead of 3

Example:

(1)	t_1 = - b		(1)	t_1 = - b
(2)	$t_2 = t_1 * c$		(2)	$t_2 = t_1 * c$
(3)	t_3 = - b	can be changed to	(3)	$t_3 = t_2 + t_2$
(4)	$t_4 = t_3 * c$		(4)	a = t_5
(5)	$t_5 = t_2 + t_4$			
(6)	a = t_5			

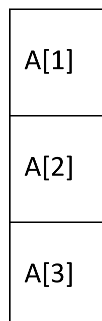
reducing number of temporary variables from 5 to 3 and reusing value of t_2

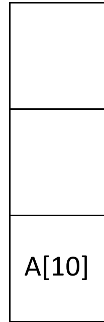
Addressing array elements

Elements of array are stored in consecutive memory location. If the array is n, and size of each element is s then the ith element of the array can be accessed at base + (i – low) * s

Where base is the base address of array or the address of the 1st element of array and low is the lower bound of array or the index of first element of array.

Example: Let A[10] be an array of 10 elements. Let size of each element is 2 ie., s=2 and the array is stored from memory location 1000 ie base address=1000.





3rd element of array

is at address

= 1000 + (3 -1) * 2

= 1000 + 2 * 2

= 1000 + 4=1005

Expression can be written as i * s + base – low * s where part I of expression is (i*s) and part II of the expression is (base - low * s).

In part II all the components are known before compilation hence they can be pre-computed stored. This reduces the time taken to generate address of ith element.

In case of multi-dimension array like matrix, elements are either stored as Row Major or Column Major.

Example: Consider Array A[3,3] with elements

(1,1)	(1,2)	(1,3)
(2,1)	(2,2)	(2,3)
(3,1)	(3,2)	(3,3)

It can be stored as

Row Major	Col Major
(1,1)	(1,1)
(1,2)	(2,1)
(1,3)	(3,1)
(2,1)	(1,2)
(2,2)	(2,2)
(2,3)	(3,2)
(3,1)	(1,3)
(3,2)	(2,3)
(3,3)	(3,3)

C language and Pascal uses row major storage where as Fortran language uses column major storage.

Address of element A [i, j] in row major storage is given by the expression as follows.

A[i,j] = base + ((i - low1) * n2 + j - low_2) * s (Exp 1)

where low1 and low2 are lower bounds of i & j and n2defines the number of columns. S defines the size of each element. Expression (Exp 1) can be written as

A[i,j] = ((i * n2) + j) * S + (base – ((low1 * n2) + low2) * s (Exp 2)

The second part of the Expression (Exp 2) can be pre-computed by knowing the value of base, low1, low2 and s. This helps in faster generation of address for A[i,j].

Arranges may be at various dimensions, hence the generalized production list is as follows

A → Alist] |id
Alist → Alist, E |id [E

Translation scheme for array with complete definition can be as follows

S → A = E
E → A
A → Alist]
A – id
Alist → Alist, E|id [E

A can be a simple name (has only one base address and no offset) or an indexed name (has base address and object) assignment to location.

	Production	Translation rule

1	S → A = E	If A.offset = null then
		Produce (A.value '=' E.value)
		else
		Produce (A.value '[' A.offset']' '=' E.value)

2	E → E1 + E2	E.value = newtemp() Produce(E.value '=' E1.value '+' E2.value)
3	E → E1 * E2	E.value = newtemp() Produce(E.value '=' E1.value '*' E2.value)
4	E → A	if A.offset = null then E.value = A.value else begin E.value = newtemp()

		Produce(E.value'=' A.value '[' A.offset ']') End
5	A → Alist]	A.value = newtemp() A.offset = newtemp() Produce (A.value '=' c (Alist.array) Produce(A.offset'='Alist.value'*'width(Alist.array))
6	A → id	A.value = id.value A.offset = null
7	Alist→ Alist1, E	t = newtemp() m = Alist1.dim + 1 Produce (t '=' Alist1.value '*' lmt(Alist1.array, m)) Produce(t '=' t ' +' E.value) Alist.array = Alist1.array Alist.value = t Alist.ndim = m
8	Alist → id [E	Alist.array = id.value Alist.value = E.value Alist.ndim = 1

L-value of A has two attributes viz A.offset and A.value. If A is a simple attribute then A.value points to symbol table and A.offset = null.

If A if array then A.offset is temporary variable which stores first part of expression (Exp 2). A.value stores second part of Exp(2). c defines the second component of expression (exp 2), m denotes the dimension of array (m=1 indicates single dimension array, m=2 defines 2-dimension array etc. Function lmt() defines the maximum number of elements present in the jth dimension of array. Width() defines the size of array.

Example: Let A be two-dimension matrix with the size 20 * 10 and lower bound of both be equal to one. ie $low_1 = low_2 = 1$ and $n_1 = 20$ $n_2 = 10$ and let the size of each element is 4 ie s = 4. Generate the address of x = A[y, z].

Fig 6.5 shows the annotated parse tree for the array assignment statement x = A[y,z]

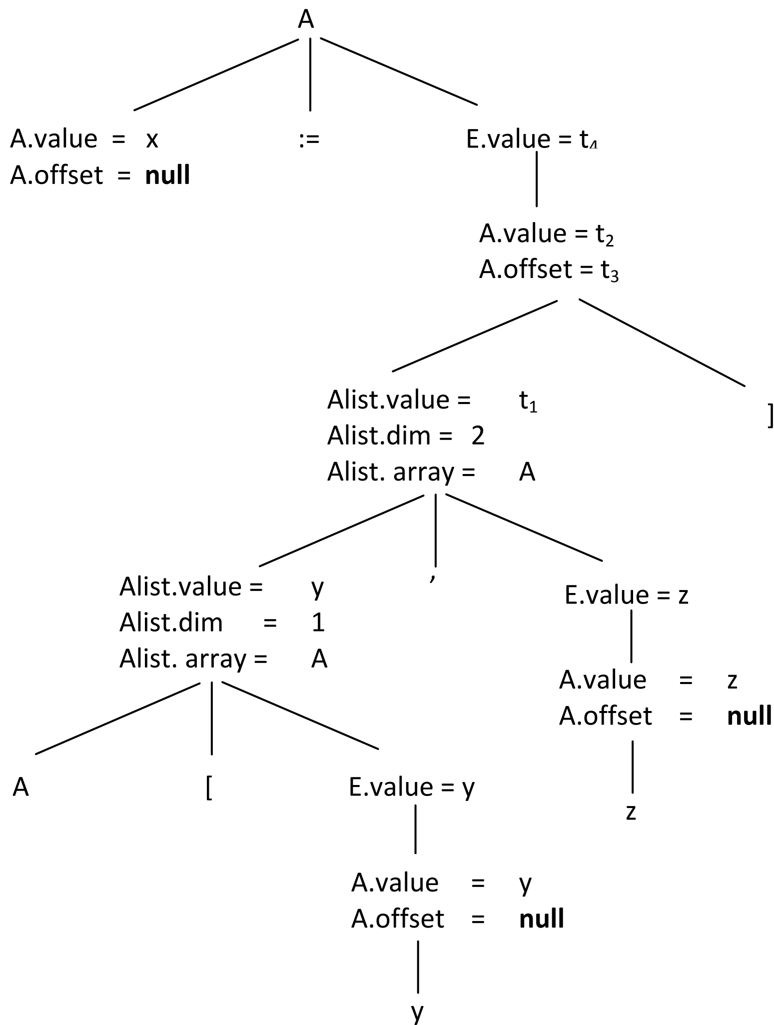


Fig 6.5 Annotated parse tree for x := A[y, z]

Logical Expression

An expression may consists of not only arithmetic operators like +, -, * etc., it may also have boolean / logical operation like and, or and not. An expression with relational operation like <, >,≤ , ≥ ≠ etc., along with logical operators are mainly used in flow control statements like if then else, while-do and repeat-until. **not** operations has the highest precedence-level followed by **and** and **or** is at least precedence level.

Logical expressions always results in values either true or false. True can be treated as non zero or non negative or 1 value. Whereas false may be 0 or negative value. Following is the translation scheme using a numerical representation for logical expression.

	Production	Translation rules
1	$E \rightarrow E_1$ or E_2	E.value = newtemp() Produce (E.value '=' E_1.value 'or' E_2.value)
2	$E \rightarrow E_1$ and E_2	E.value = newtemp() produce (E.value '=' E_1.value 'and' E_2.value)
3	$E \rightarrow$ not E_1	E.value = newtemp() produce (E.value '=' 'not' E_1.value)
4	$E \rightarrow (E_1)$	E.value = E_1.value
5	$E \rightarrow$ **id_1 relop id_2**	E.value = newtemp() produce ('if' **id_1**.value **relop**.op **id_2.**value 'goto' nextstat + 3) produce (E.value '=' '0') produce (**'goto'** nextstat + 2) produce (E.value '=' '1')
6	$E \rightarrow$ **true**	E.value = newtemp() produce (E.value '=' '1')
7	$E \rightarrow$ **false**	E.value = newtemp() Produce (E.value '=' '0')

Example:

1. a or b and not c

   Three address code for the above expression will be as follows

   t1 = not c

   t2 = b and t1

   t3 = a or t2

2. if a < b then 1else 0

   Three address code for the above statement is as follows

   10: if a < b go to 13

   11: t1 = 0

   12: go to 14

13: t1 = 1

14:

3. a < b or c < d and e < f

10: if a < b go to 13

11: t1= 0

12: go to 14

13: t1 = 1

14: if c < d go to 17

15: t2 = 0

16: go to

17: if e < f go to 20

18: t3 = 0

19: go to 21

20: t3 = 1

21:

Flow control statements

Control statements are used to alter the sequential flow of execution. Some the control statements are if-then-else statement, while statement. Following is the pictorial representation of flow control statements.

Control flow for if-then statement is as show in Fig 6.6

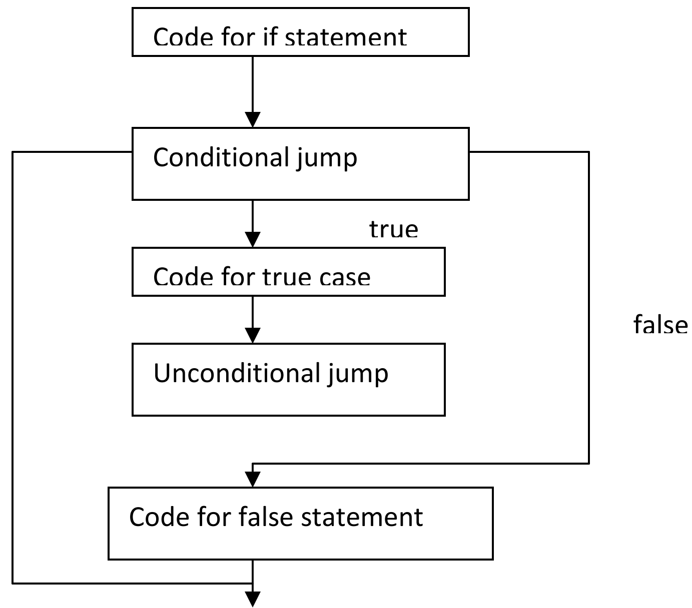


Fig 6.6 Control flow for if-then statement

Control flow for while statement is as shown in Fig 6.7

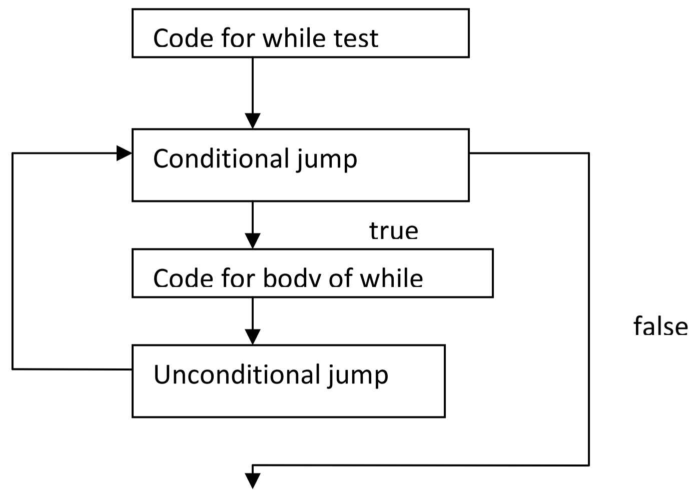
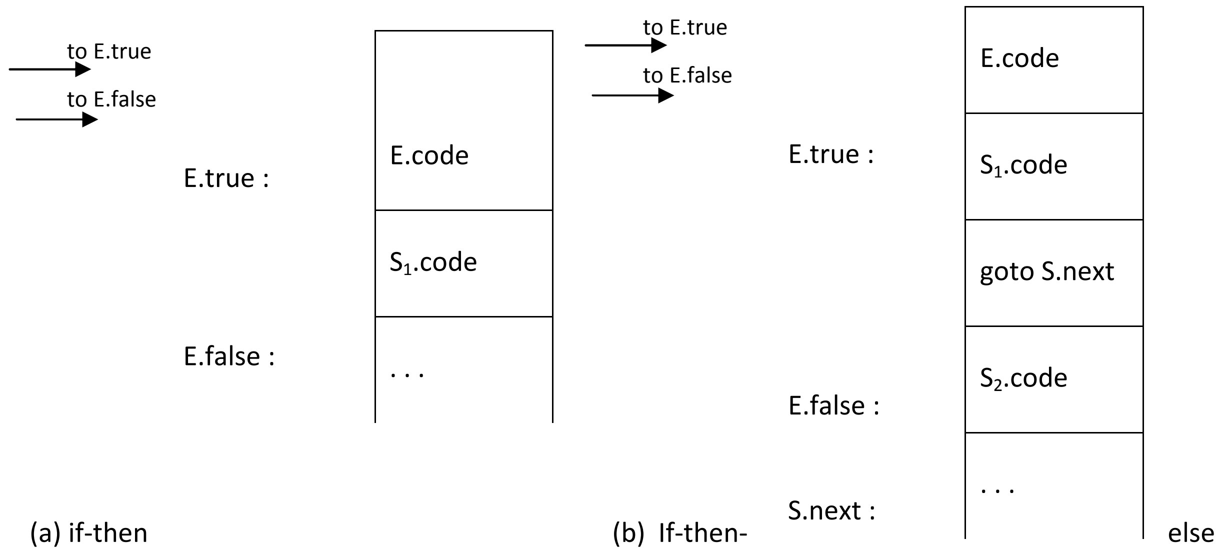
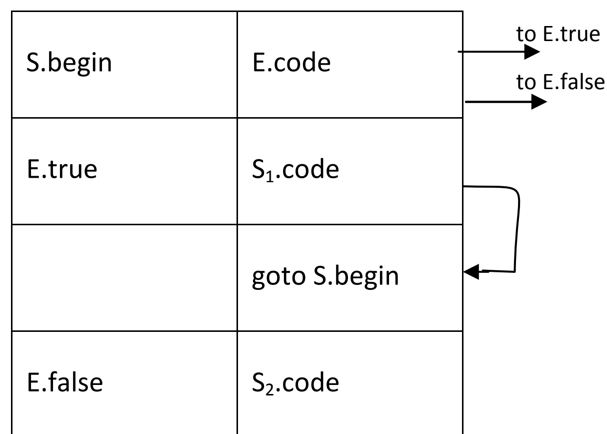


Fig 6.7 Control flow for while statement



(a) if-then (b) If-then-else



(c) while-do

Fig 6.8 Code for if-the, if-then-else and while statement

Code for if-then and if-then-else can be generated using the following translation rules.

Statement	**Translation rules**
S->if E then S_1	E.true = newlabel(); E.false = S.next; S_1.next = S.next; S.code = E.code || gen(E.True, ':')|| S_1.code
S->if E then S_1 else S_2	E.true = newlabel();

	E.false = newlabel(); S_1.next = S.next; S_2.next = S.next; S.code = E.code || gen(E.True, ‘:’)|| S_1.code || gen(‘GOTO’, S.next) || gen(E.false, ‘:’) || S_2.code
S->while E do S1	S.begin= newlabel(); E.true=newlabel(); E.false=S.next S1.next=S.begin S.code=gen(S.begin ‘:’)|| E.code||gen(E.true’:’)||S1.code||gen(‘GOTO’ S.begin)

Example:

Generate three address code for the following statement

while a<b do

if c< d then

x = y + z

else

x = y - z

Solution: the three address code will be as follows

```
L1:     if a<b then GOTO L2
        GOTO LNEXT
L2:     if c<d then GOTO L3
        GOTO L4
L3:     t1 =  y + z
        x = t1
```

GOTO L1

L4: t1= y - z

x = t1

GOTO L1

LNEXT:

Backpatching:

Forward mapping of statement numbers will be difficult, the concept of backpatching is used. Here list of statements for true and false are of Boolean expression are separately maintained. Once a specific statement is encountered, the compiler backpatches the address or statement number. Following functions are used for backpatching.

- Makelist(i) – creates a list containing only i, an index into array of instructions. Makelist returns a pointer to the newly created list.
- Merge(p1,p2) – concatenates the lists pointed by p1 and p2 and returns the pointer to the concatenated list
- Backpatch(p,i)- inserts i as the target label for each of the instructions on the list pointed by p

Consider the translation rules modified for boolean functions.

Production	Translation rule
B→ B_1 || MB_2	backpatch(B_1.falselist,M.instr) B.truelist=merge(B_1.truelist,B_2.truelist) B.falselist=B_2.falselist
B →B_1&& MB_2	backpatch(B_1.true,M.instr) B.falselist=merge(B_1.falselist,B_2.falselist) B.truelist=B_2.truelist
B→ !B_1	B.truelist=B_1.falselist B.falselist=B_1.truelist
B → (B_1)	B.truelist=B_1.truelist B.falselist=B_1.falselist

B → E1 rel E2	B.truelist=makelist(nextinstr) B.falselist=makelist(nextinstr+1) emit(‘if’ E1.addr rel.op E2.addr ‘goto’ ---) emit(‘goto’ ---)
B → true	B.truelist=makelist(nextinstr) emit(‘goto’ ---)
B → false	B.falselist = makelist(nextinstr) emit(‘goto’ ---)
M → ε	M.instr=nextinstr

Consider the following program statement

x<10 || x> 200 && x!=y

initially the three address code for each conditional statement is as follows

x<10

100: if x < 100 goto ---

101: goto ---

x>200

102: if x>200 goto ---

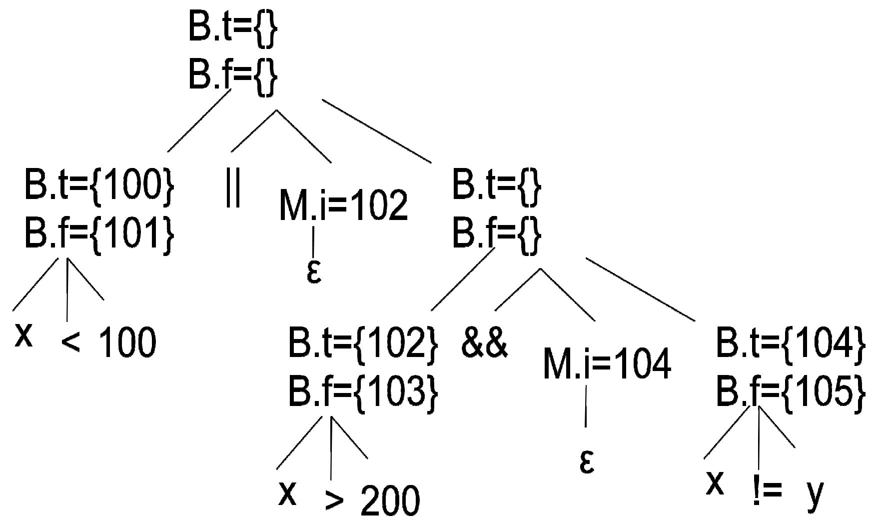
103: goto ---

x!=y

104: if x!=y goto ---

105: goto ---

Following tree represented the true list and false list for every expression



After backpatching the following code is generated

x<10 || x> 200 && x!=y

x<10

100: if x < 100 goto ---

101: goto 102

x>200

102: if x>200 goto 104

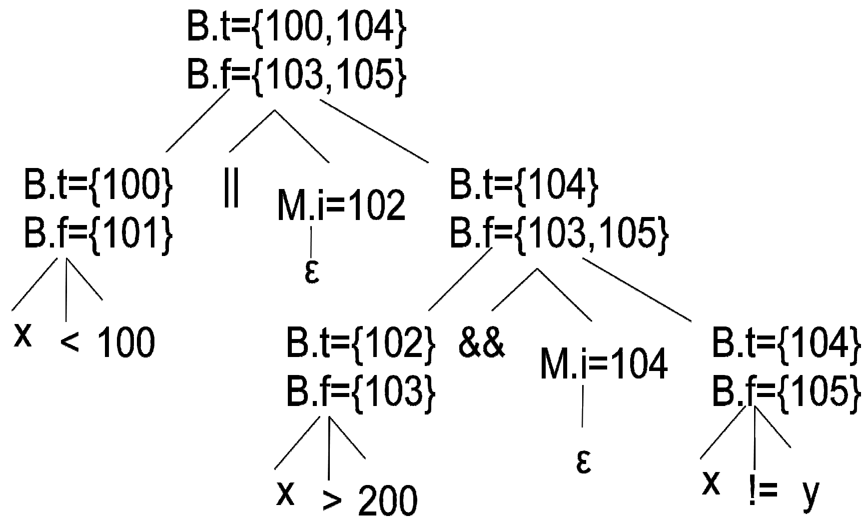
103: goto ---

x!=y

104: if x!=y goto ---

105: goto ---

Finally the following tree represents the complete list of true list and false list for all the expressions.



Three address code for switch statement

Consider the following switch statement

switch(E)

{ case V_1: S_1

case V_2: S_2

…

case V_{n-1}:S_{n-1}

default: S_n

}

The three address code for the above switch statement is as follows

Code to evaluate E into t

goto test

L_1: code for S_1

goto next

L_2: code for S_2

goto next

…

L_{n-1} code for S_{n-1}

goto next

L_n: code for S_n

goto next

test: if t=V_1 goto L_1

…

if t=V_{n-1} goto L_{n-1}

goto L_n

next:

Three address code for procedure call

Consider the statement

n=f(a[i])

where a is array of integers f is function from integers to integers

The three address code for the procedure call will be as follows

t1 = i * 4

t2 = a[t1]

param t2

t3 = call f,1

n = t3