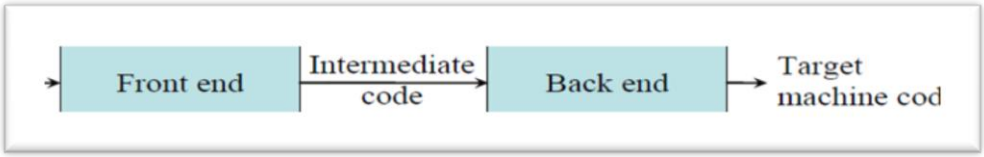
Chapter 6

INTERMEDIATE CODE GENERATOR

Intermediate Code Generation

The front end translates the source program into an intermediate representation from which the backend generates target code. Intermediate codes are machine independent codes, but they are close to machine instructions.



Advantages of using Intermediate code representation:

- If a compiler translates the source language to its target machine language without having the option for generating intermediate code, then for each new machine, a full native compiler is required.
- Intermediate code eliminates the need of a new full compiler for every unique machine by keeping the analysis portion same for all the compilers.
- The second part of compiler, synthesis, is changed according to the target machine.
- It becomes easier to apply the source code modifications to improve code performance by applying code optimization techniques on the intermediate code.

Intermediate Representations

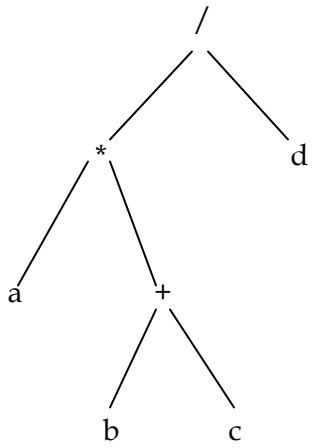
There are three kinds of intermediate representations:

1. *Graphical representations* (e.g. Syntax tree or Dag)
2. *Postfix notation*: operations on values stored on operand stack (similar to JVM byte code)
3. *Three-address code*: (e.g. triples and quads) Sequence of statement of the form $x = y$ op z

Syntax tree:

Syntax tree is a graphic representation of given source program and it is also called variant of parse tree. A tree in which each leaf represents an operand and each interior node represents an operator is called syntax tree.

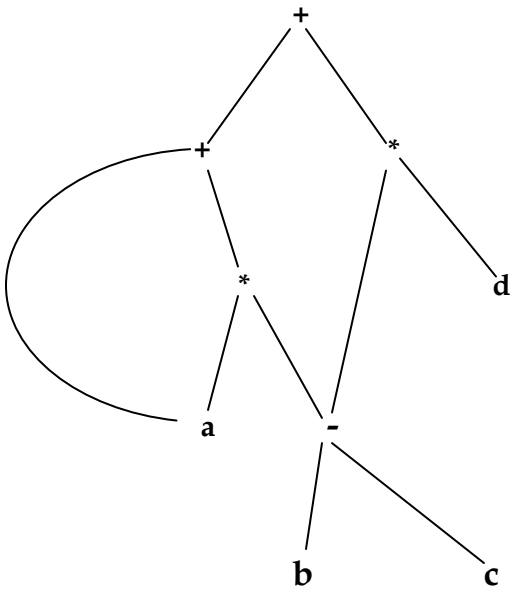
Example: Syntax tree for the expression a*(b + c)/d



Directed acyclic graph (DAG)

A DAG for an expression identifies the common sub expressions in the expression. It is similar to syntax tree, only difference is that a node in a DAG representing a common sub expression has more than one parent, but in syntax tree the common sub expression would be represented as a duplicate sub tree.

Example: DAG for the expression a + a * (b - c) + (b - c) * d



Postfix notation

The representation of an expression in operators followed by operands is called postfix notation of that expression. In general if x and y be any two postfix expressions and OP is a binary operator then the result of applying OP to the x and y in postfix notation by "x y OP".

Examples:

1. (a+ b) * c in postfix notation is: a b + c *
2. a * (b + c) in postfix notation is: a b c + *

- Postfix notation is the useful form of intermediate code if the given language is expressions.
- Postfix notation is also called as 'suffix notation' and 'reverse polish'.
- Postfix notation is a linear representation of a syntax tree.
- In the postfix notation, any expression can be written unambiguously without parentheses.
- The ordinary (infix) way of writing the sum of x and y is with operator in the middle: x * y. But in the postfix notation, we place the operator at the right end as xy *.
- In postfix notation, the operator follows the operand.

3. Three Address Code

The address code that uses at most three addresses, two for operands and one for result is called three code. Each instruction in three address code can be described as a 4-tuple: (operator, operand1, operand2, result).

A quadruple (Three address code) is of the form:

x **=** y ***op*** z *w*here x, y and z are names, constants or compiler-generated temporaries and ***op*** is any operator.

We use the term "three-address code" because each statement usually contains three addresses (two for operands, one for the result). Thus the source language like $x + y * z$ might be translated into a sequence

t1 = y * z

t2 = x + *t1* where *t1* and *t2* are the compiler generated temporary name.

- Assignment statements: x **=** y *op z, op is binary*
- Assignment statements: x **=** ***op*** y, *op is unary*
- Indexed assignments: x **=** y[i], x[i] **=** y
- Pointer assignments: x **=** **&**y, x **=** ***y, ***x **=** y
- Copy statements: x **=** y
- Unconditional jumps: **goto** *label*
- Conditional jumps: **if** x *relop* y **goto** *label*
- Function calls: **param** x… **call** p, n **return** y

Example: Three address code for expression: (B+A)*(Y-(B+A))

t1 = B + A
t2 = Y - t1
t3 = t1 * t2

Example 2: Three address code for expression: [TU]

i* = 2 * *n* + *k
While i do
i* = *i* – *k

Solution:

t1 = 2
t2 = t1 * n
t3 = t2 + k
i = t3
L1: if i = 0 goto L2
t4 = i - k
i = t4
goto L1
L2: ……..

Implementation of Three-Address Statements

A three-address statement is an abstract form of intermediate code. In a compiler, these statements can be implemented as records with fields for the operator and the operands. Three such address statements representations are **Quadruples and Triples.**

Quadruples

A quadruple is a record structure with four fields, which are, **op, arg1, arg2** and **result**. The **op** field contains an internal code for the operator. The three-address statement **x = y op z** is represented by placing **y** in **arg1**, **z** in **arg2** and **x** in result. The contents of fields i.e. **arg1**, **arg2** and **result** are normally pointers to the symbol-table entries for the names represented by these fields. If so, temporary names must be entered into the symbol table as they are created.

	op	**arg1**	**arg2**	**result**
(0)	-	c		t_1
(1)	*	b	t_1	t_2
(2)	-	c		t_3
(3)	*	b	t_3	t_4
(4)	+	t_2	t_4	t_5
(5)	=	t_5		a

Figure: Quadruple representation of three-address statement a = b * - c + b * - c

Triples

To avoid entering temporary names into the symbol table, we might refer to a temporary value by the position of the statement that computes it. If we do so, three-address statements can be represented by records with only three fields: **op, arg1 and arg2**. The fields **arg1** and **arg2**, for the arguments of **op**, are either pointers to the symbol table or pointers into the triple structure (for temporary values). Since three fields are used, this intermediate code format is known as triples.

	op	**arg1**	**arg2**
(0)	uminus	c	
(1)	*	b	(0)
(2)	uminus	c	
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	assign	a	(4)

Figure: Triple representation of three-address statement a = b * - c + b * - c

(0) -c

(1) b* (0)

(2) -c

(3) b*(2)

(4) (1)+(3)

(5) a=(4)

A ternary operation like **x[i] = y** requires two entries in the triple structure as shown as below while **x = y[i]** is naturally represented as two operations.

	op	**arg1**	**arg2**
(0)	[ ]	x	i
(1)	assign	(0)	y

(a)

	Op	**arg1**	**arg2**
(0)	[ ]	y	i
(1)	assign	x	(0)

(b)

Figure: Triple representation of three-address statement (a) **x[i]= y** and (b) **x= y[i]**

Example: Translate the expression x = (a + b) * (c + d) – (a + b + c)

a. **Quadraples**
b. **Triples**

Solution: Three address code is:

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

Quadruples:

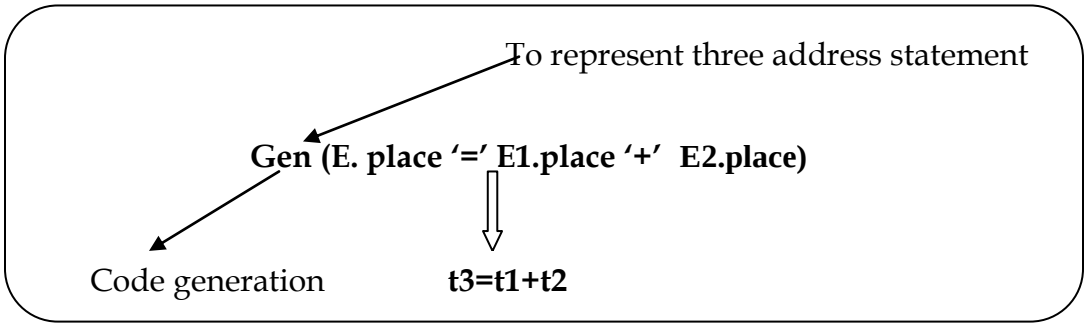
	op	arg1	arg2	result
(0)	+	a	b	t_1
(1)	+	c	d	t_2
(2)	*	t_1	t_2	t_3
(3)	+	t_1	c	t_4
(4)	-	t_3	t_4	t_5
(5)	=	t_5		x

Triple:

	op	arg1	arg2
(0)	+	a	b
(1)	+	c	d
(2)	*	(0)	(1)
(3)	+	(0)	c
(4)	-	(2)	(3)
(5)	=	x	(4)

Naming conventions for three address code

- *S*.code →three-address code for evaluating *S*
- *S*.begin → label to start of *S* or nil
- *S*.after →label to end of *S* or nil
- *E*.code → three-address code for evaluating *E*
- *E*.place → a name that holds the value of *E*



Syntax-Directed Translation into Three-Address Code

1. Assignment statements

Productions	**Semantic rules**
$S \rightarrow$ **id =** E	*S*.place = *newtemp*();
	S.code = *E*.code || *gen* (**id**.place '=' *E*.place); *S*.begin = *S*.after = nil
$E \rightarrow E1$ **+** $E2$	*E*.place = *newtemp*();
	E.code = *E*1.code || *E*2.code || *gen* (*E*.place '=' *E*1.place '+' *E*2.place)

$E \rightarrow E1 * E2$	*E*.place = *newtemp*();
	E.code = *E*1.code || *E*2.code || *gen* (*E*.place '=' *E*1.place '*' *E*2.place)
$E \rightarrow -E1$	*E*.place = *newtemp*();
	E.code = *E*1.code || *gen* (*E*.place '=' 'minus' *E*1.place)
$E \rightarrow (E1)$	*E*.place = *newtemp*();
	E.code = *E*1.code
$E \rightarrow$ **id**	*E* .place = **id**.name
	E.code = null

2. Boolean Expressions

Boolean expressions are used to compute logical values. They are logically used as conditional expressions in statements that alter the flow of control, such as if—then, if—the—else or while---do statements.

Control-Flow Translation of Boolean Expressions

Production	Semantic Rules
$B \rightarrow B_1 \mid\mid B_2$	B_1.true = B.true B_1.false = newlabel() B_2.true = B.true B_2.false = B.false B.code = B_1.code || label(B1.false) || B_2.code
$B \rightarrow B_1 \&\& B_2$	B_1.true = newlabel() B_1.false = B.false B_2.true = B.true B_2.false = B.false B.code = B_1.code || label(B1.true) || B_2.code
$B \rightarrow ! B_1$	B_1.true = B.false B_1.false = B.true B.code = B_1.code
$E \rightarrow id_1$ relop id_2	E.code = gen('if' id1.place relop.op id2.place 'goto' E.true) || gen('goto' E.false)
B → true	B.code = gen(goto B.true)
B → false	B.code = gen(goto B.false)

3. Flow of control statements

Control statements are 'if—then', 'if—then—else', and 'while---do'. Control statements are generated by the following grammars:

S → If exp then S1

S → If exp then S1 else S2

S → while exp do S1

Production	Semantic Rules
$S \rightarrow if (B) S_1$	B.true = newlabel() B.false = S.next S_1.next = S.next S.code = B.code || label(B.true) || S_1.code
$S \rightarrow if (B) S_1$ else S_2	B.true = newlabel() B.false = newlabel() S_1.next = S.next S_2.next = S.next S.code = B.code || label(B.true) || S_1.code || gen(goto S.next) || label(B.false) || S_2.code
$S \rightarrow while (B) S_1$	begin = newlabel() B.true = newlabel() B.false = S.next S_1.next = begin S.code = label(begin) || B.code || label(B.true) || S_1.code || gen(goto begin)

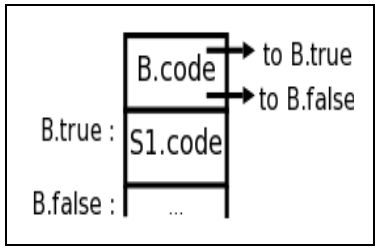


Fig: If--then

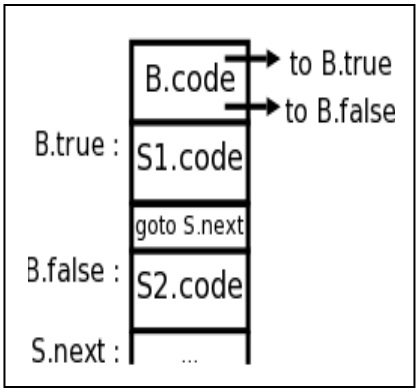


Fig: If—then—else

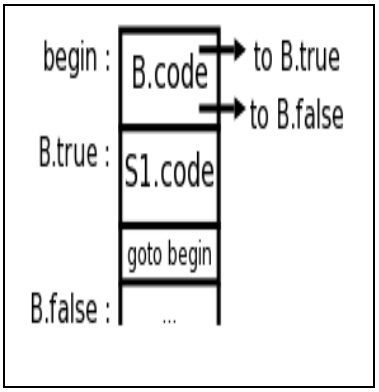


Fig: while---do

Example 1: Generate three address code for the expression

if (x < 5 || (x > 10 && x == y)) x = 3 ;

Solution:

L1: if x < 5 goto L_2

goto L_3

L_3: if x > 10 goto L_4

goto L_1

L_4: if x == y goto L_2

goto L_1

L_2: x = 3

Example 2: Generate three address code for the expression

if (x < 5 || (x > 10 && x == y))

{

x = 3

}

Solution:

L1: if X<5 goto L2

else

goto L3

L2: x=3 goto L5

L3: if x>10

goto L4

else

goto L5

L4: if x==y

goto L2

else

goto L5

L5: …….

Switch/ case statements

A switch statement is composed of two components: an expression E, which is used to select a particular case from the list of cases; and a case list, which is a list of n number of cases, each of which corresponds to one of the possible values of the expression E, perhaps including a default value.

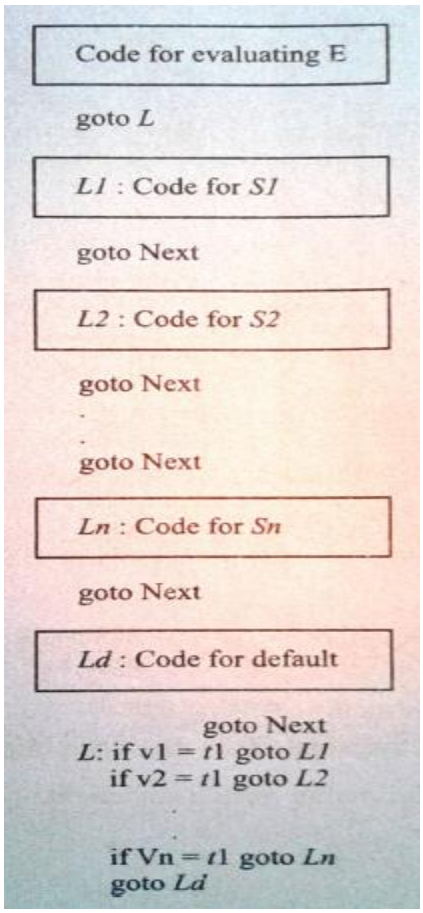


Fig: A switch / case three address translation

Syntax:

```
Switch (E)
{
    Case V1: S1
    Case V2: S2
    ……
    ……
    Case Vn : Sn
}
```

Example 1: Convert the following switch statement into three address code:

```
Switch (i + j)
{
        Case 1: x=y + z
        Case 2: u=v + w
        Case 3: p=q * w
        Default: s=u / v
```

}

Solution:

```
        t=i+j
L1:  if t==1    goto L2
        else    goto L3
L2:  x=y+z      goto L8
L3: if t==2     goto L4
        else    goto L5
L4: u=v + w     goto L8
L5: if t==3     goto L6
        else    goto L7
L6: p=q * w     goto L8
L7:  s=u / v goto L8
L8: ……..
```

Addressing array elements:

Elements of an array can be accessed quickly if the elements are stored in a block of consecutive locations. If the width of each array element is w, then the i^{th} element of array 'A' begins in location,

base + (i – low)* w

Where low is the lower bound on the subscript and base is the relative address of the storage allocated for the array. That is base is the relative address of A[low].

The given expression can be partially evaluated at compile time if it is rewritten as,

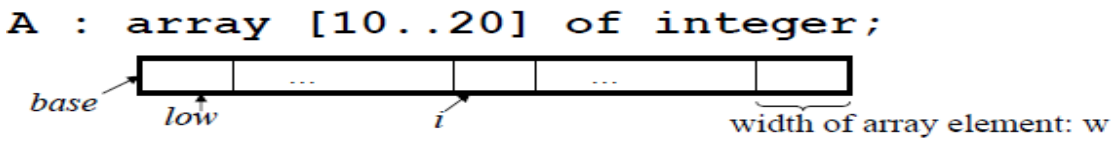
i * w + (base - low* w)

= i * w + C

Where C=base – low*w can be evaluated when the declaration of the array is seen.

We assume that C is saved in the symbol table entry for A, so the relative address of A[i] is obtained by simply adding i * w to C.

i.e A[i]= i* w + C



A[i] = base $_A$ + (i - low) * w

= i * w + c **where** c = base $_A$ - low * w **with** low = 10; w = 4

Example: address of 15^{th} element of array is calculated as below,

Suppose base address of array is 100 and type of array is integer of size 4 bytes and lower bound of array is 10 then,

A[15]=15 * 4 + (100 – 10 * 4)

= 60 + 60

= 120

Similarly for two dimensional array, we assume that array implements by using row major form, the relative address of $A[i_1, i_2]$ can be calculated by the formula,

$$A[i_1,i_2]=base_A + ((i_1 - low_1) * n_2 + i_2 - low_2) * w$$

Where low_1, low_2 are the lower bounds on the values i_1 and i_2, n_2 is the number of values that i_2 can take. Also given expression can be rewrite as,

$$= ((i_1 * n_2) + i_2) * w + base_A - ((low_1 * n_2) + low_2) * w$$

$$= ((i_1 * n_2) + i_2) * w + C \quad \textbf{where } C= base_A - ((low_1 * n_2) + low_2) * w$$

Example: Let A be a 10 X 20 array, there are 4 bytes per word, assume low1=low2=1.

Solution: Let X=A[Y, Z]

Now using formula for two dimensional array as,

$$((i_1 * n_2) + i_2) * w + base_A - ((low_1 * n_2) + low_2) * w$$

$$= ((Y * 20) + Z) * 4 + base_A - ((1 * 20) + 1) * 4$$

$$= ((Y * 20) + Z) * 4 + base_A - ((1 * 20) + 1) * 4$$

$$= ((Y * 20) + Z) * 4 + base_A – 84$$

We can convert the above expression in three address codes as below:

T1= Y * 20

T1= T1+Z

T2=T1*4

$T3=base_A$ -84

T4=T2+ T3

X= T4

4. Procedure Calls

The procedure is such an important and frequently used programming construct that it is imperative for a compiler to generate good code for procedure calls and returns. The run-time routines that handle procedure argument passing, calls and returns are part of the run-time support package.

param x call p return y

Here, p is a function which takes x as a parameter and returns y.

Procedure calls: They have the form

param x1

param x2

……..

param xn

call p, n

corresponding to the procdure call p(x1, x2, ..., xn)

Return statement

They have the form return y where y representing a returned value is optional.

Calling Sequences:

The translation for a call includes a calling sequence, a sequence of actions taken on entry to and exit from each procedure. The falling are the actions that take place in a calling sequence:

- When a procedure call occurs, space must be allocated for the activation record of the called procedure.
- The arguments of the called procedure must be evaluated and made available to the called procedure in a known place.
- Environment pointers must be established to enable the called procedure to access data in enclosing blocks.
- The state of the calling procedure must be saved so it can resume execution after the call.
- Also saved in a known place is the return address, the location to which the called routine must transfer after it is finished.
- Finally a jump to the beginning of the code for the called procedure must be generated.

5. Back patching

The easiest way to implement the syntax-directed definitions for boolean expressions is to use two passes. First, construct a syntax tree for the input, and then walk the tree in depth-first order, computing the translations. The main problem with generating code for Boolean expressions and flow-of-control statements in a single pass is that during one single pass we may not know the labels that control must go to at the time the jump statements are generated. Hence, a series of branching statements with the targets of the jumps left unspecified is generated. Each statement will be put on a list of goto statements whose labels will be filled in when the proper label can be determined. We call this subsequent filling in of labels **backpatching**.

To manipulate lists of labels, we use three functions:

1. **makelist(i)** creates a new list containing only **i**, an index into the array of quadruples; **makelist** returns a pointer to the list it has made.
2. **merge(p_1,p_2)** concatenates the lists pointed to by $\mathbf{p_1}$ and $\mathbf{p_2}$, and returns a pointer to the concatenated list.
3. **backpatch(p,i)** inserts i as the target label for each of the statements on the list pointed to by **p**.

If we decide to generate the three address code for given syntax directed definition using single pass only, then the main problem that occurs is the decision of addresses of the labels. 'goto' statements refer these label statements and in one pass it becomes difficult to know the location of these label statements. The idea to back-patching is to leave the label unspecified and fill it later, when we know what it will be.

If we use two passes instead of one pass then in one pass we can leave these addresses unspecified and in second pass this incomplete information can be filled up.