INTERMEDIATE CODE GENERATION

INTERMEDIATE CODES

Outcome :

- ✓ Understanding intermediate codes.
- ✓ Classification of intermediate codes.

How Java works

Source code → Javac → Byte code ⇒ (Intermediate code)

Byte code ↓ JVM

JVM → Machine code (windows) | Machine code (Mac) | Machine code (UNIX)

Classification of intermediate codes

Eg : (a+b) * (a+b+c)

Intermediate codes
- Linear Form
  - Postfix Expression
  - Three Address code (TAC)
- Tree Form
  - Syntax Tree
  - Directed Acyclic Graph (DAG)

Postfix Expression: ab+ab+c+*

Three Address code (TAC):

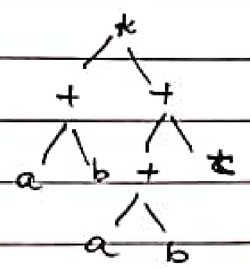
t1 = a+b

t2 = a+b

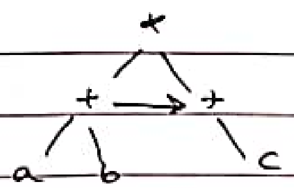
t3 = t2+c

t4 = t1*t3

Syntax Tree:



Directed Acyclic Graph (DAG):



Summary :

✓ Understanding IC ✓ Classification of IC.

THREE ADDRESS CODES (TAC)

Outcome :

- Different forms of TAC.
- Various representations of TAC.

Different forms of TAC :

The HLL expressions are converted into the intermediate code using any of the following TAC forms.

	TAC Form	Usage
1	$x = y \text{ op } z$	Binary operation, then assignment
2	$x = \text{op } z$	Unary operation, then assignment
3	$x = y$	simple assignment
4	if (x <rel op> y) GOTO L	conditional GOTO
5	GOTO L	unconditional GOTO
6	$A[i] = x$ $y = A[i]$	used for arrays
7	$x = *p$ $y = \&x$	'x' is a value pointed by the pointer 'p' Address of 'x' is stored in 'y'.

Representations of TAC :

Traditionally, 3 types of representations of TAC are popularly used.

1. Quadruples :

These use temporary variables (or temporary registers) for each individual portion of the expression in order to store the result.
Each instruction is splitted into the following 4 different fields : opr, op1, op2, result.

② **Triples :**

For these temporary variables are not dedicated, rather temporary registers are used on demand. The flow of evaluation is numbered and cannot be altered.

③ **Indirect Triples :**

It uses pointers to the listing of all references to computations which is made separately and stored in memory.
Temporaries are implicit and is re-arrangable like quadruples.

Expression : $-(a+b) * (c+d) + (a+b+c)$

TAC :

① $t1 = a+b$
② $t2 = -t1$
③ $t3 = c+d$
④ $t4 = t2 * t3$
⑤ $t5 = a+b$
⑥ $t6 = t5 + c$
⑦ $t7 = t4 + t6$

In quadruples
If a & b aren't available, we can execute ③ before ①.
In triples we cannot do that.
In indirect triples also we can do that.

(representations)

Quadruples

	opr	op1	op2	res
①	+	a	b	t1
②	−	t1		t2
③	+	c	d	t3
④	*	t2	t3	t4
⑤	+	a	b	t5
⑥	+	t5	c	t6
⑦	+	t4	t6	t7

Triples

	op	op1	op2
①	+	a	b
②	−	(1)	
③	+	c	d
④	*	(2)	(3)
⑤	+	a	b
⑥	+	(5)	c
⑦	+	(4)	(6)

Indirect Triples

	Reference
(i)	(1)
(ii)	(2)
(iii)	(3)
(iv)	(4)
(v)	(5)
(vi)	(6)
(vii)	(7)

__/__/__

Quadruples :

Pro : statements can be re-arranged.
Con : Too many registers are needed.

Triples :

Pro : Temporary registers are used on demand.
Con : Statements can't be rearranged.

Indirect Triples :

Pro : statements comprise only reference (pointers), thus rearrangeable
Con : 2 memory accesses.

Summary :

- Different forms of TAC.
- various representations of TAC.

BACKPATCHING & CONVERSION TO TAC

outcome :

- How backpatching works during the generation of TAC.
- Example of conversion from HLL to TAC.

Conversion to TAC :

if (a<b) then t = 1
else t = 0

consists of 4 addresses

we cannot use absolute addresses, we use relative addresses.

```
i+0 : if (a<b) GOTO (i+3)
i+1 : t = 0
i+2 : GOTO (i+4)
i+3 : t = 1
i+4 :
```

The procedure of leaving the labels empty and filling these later is called Backpatching.

Q : convert the HLL : $(a<b)$ && $(c<d)$ || $(e<f)$ into TAC using Back patching [Assume that absolute addressing is used from location 100 onwards]

t_1 will determine $a<b$

t_2 will determine $c<d$

t_3 will determine $e<f$

t_4 will determine t_1 & t_2

t_5 will determine t_4 | t_3

```
100 : if (a<b) GOTO 103
101 :   t1 = 0
102 :   GOTO 104
103 :   t1 = 1
104 : if(c<d) GOTO 107
105 :   t2 = 0
106 :   GOTO 108
107 :   t2 = 1
108 : if(e<f) GOTO 111
109 :   t3 = 0
110 :   GOTO 112
111 :   t3 = 1
112 :   t4 = t1 & t2
113 :   t5 = t4 | t3
```

summary :

- How backpatching works during the generation of TAC.
- Example of conversion from HLL to TAC.

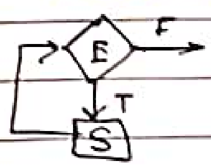
TAC - WHILE LOOP

outcome :

- conversion of while loop to TAC.
- Example problem.

while loop : while (E) do S

Flow chart



while (E)
{ S ⟶ single/block of statements
}

__/__/____

way 1 :

```
L:  if (E==0) GOTO L1
    S
    GOTO L
L1 :
```

way 2 :

```
L :  if (E)  GOTO L1
     GOTO L2
L1 : S
     GOTO L
L2 :
```

Note : After the HLL loop has been converted to TAC, it is impossible to recognize it, as it has been implemented using conditional & unconditional GOTO TAC statements.

Q : convert the HLL to TAC :

```
while (x<y)
{   a=b+c;
    x++;
}
```

(while loop)

way 2

```
L:  if (x<y) GOTO L1
    GOTO L2
L1: t1 = b + c
    a = t1
    t2 = x+1
    x = t2
    GOTO L
L2 :
```

way 1

```
L:  if (x>=y) GOTO L1
    t1 = b+c
    a = t1
    t2 = x+1
    x = t2
    GOTO L
L1 :
```

summary :

(v) conversion of while loop to TAC.

(✓) Example problem.

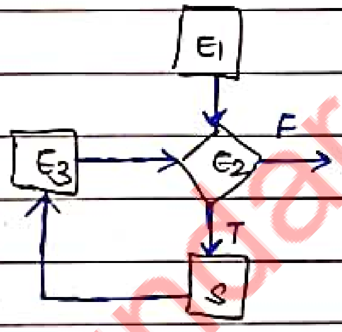
TAC - FOR LOOP

outcome :

1. conversion of For Loop to TAC

For Loop

```
for (E1 ; E2 ; E3)
{   S
}
```



E1 : initialization

E2 : condition

E3 : increment / decrement

Q :

```
for (int i = 0; i < 10; i++)
{      a = b + c;
}
```

```
    i = 0
L : if (i < 10) GOTO L1
    GOTO L2
L1 : t1 = b + c
     a = t1
     t2 = i + 1
     i = t2
     GOTO L
L2 :
```

Generalized way :

```
    E1
L : if (E2 == 0) GOTO L1
    S
    E3
    GOTO L
L1 :
```

Summary :

1. conversion of for loop to TAC. (Three Address code)

TAC - MULTIWAY BRANCHING

outcome :

1. conversion of switch-case to TAC.

TAC of switch - case :

Q:

```
switch (m+n)
{
    case (1) : a=b+c; break;
    case (2) : p=q+r; break;
    Default  : x=y+z; break;
}
```

```
      t = m+n
      GOTO T
l1 :  t1 = b+c
      a = t1
      GOTO l4
l2 :  t2 = q+r
      p = t2
      GOTO l4
l3 :  t3 = y+z
      x = t3
      GOTO l4
T :   if (t==1) GOTO l1
      if (t==2) GOTO l2
      GOTO l3
l4 :
```

Summary : 1. conversion of switch-case to TAC.

TAC - 2 DIMENSIONAL ARRAY

outcome :

1. Understanding memory representation of 2D arrays.
2. conversion of HLL code which accesses a 2D array to TAC.

_/_/___

2D Arrays

A [3x3]

00	01	02
10	11	12
20	21	22

→ This is how we the human beings think about arrays.

But computer memory is contiguous.

used by majority of the compilers ← Row major order

00	01	02	10	11	12	20	21	22

column major order

00	10	20	01	11	21	02	12	22

Row major, example accessing cell 21

crossing 2 rows = 2×3 = 6 elements
+ 1 column = 1 = 1 element
7 elements

A [10] [20]
x = A [y, z]
size of each cell = 4

t1 = y * 20
t2 = t1 + z
t3 = t2 * 4
t4 = base address of A
x = t4 [t3]

A						....

(base address = t4)

Summary :

✓ Understanding memory representation of 2D arrays.
✓ conversion of HLL code which accesses a 2D array to TAC.

__/__/__

TAC - SOLVED PROBLEMS (SET-1)

outcome :

✓ 2 solved problems on TAC.

Q: The least number of temporary variables required to create a TAC in static single assignment form for the expression $a = b*d - c + b*e - c$ is

(a) 3 (b) 4 (c) 5 (d) 6

$t1 = b * d$

$t2 = b * e$

$t3 = t1 + t2$

$t4 = t3 - 2*c$

$a = t4$

$a = (b*d) + (b*e) - (2*c)$

→ 4 variables

$\boxed{(b)\ 4}$

Q: In a simplified computer the instructions are :

OP R_j, R_i ⇒ Performs R_j OP R_i and stores the result in register R_j

OP m, R_i ⇒ Performs val op R_i and stores the result in register R_i. val is the content of memory location m.

MOV m, R_i ⇒ Moves the content of memory location m to register R_i

MOV R_i, m ⇒ Moves the content of register R_i to memory location m.

The computer has only 2 registers and OP is either ADD or SUB. consider the following basic block :

$t1 = a+b$ $t3 = e - t2$

$t2 = c+d$ $t4 = t1 - t3$

__/__/____

Assume that all operands are initially in memory. The final value of the computation should be in memory. What is the minimum number of MOV instructions in the code generated for this basic block?

(GATE 2007)

MOV	0, R_1	
MOV	2, R2	
ADD	1, R1	$R1 = a+b$
ADD	3, R2	$R2 = c+d$
SUB	e, R2	$R2 = e-(c+d)$
SUB	R_1, R_2	$R1 = (a+b)-[e-(c+d)]$
MOV	R1, 5	

store in memory location 5

(③ MOV instructions)

0	a
1	b
2	c
3	d
4	e
5	?

store result

summary :

✓ 2 solved problems on TAC.

TAC - SOLVED PROBLEMS (SET-2)

outcome :

✓ 2 solved problems on TAC.

Q: Consider the grammar rule $E \to E_1 - E_2$ for arithmetic expressions. The code generated is targetted to a CPU having a single user register. The subtraction operation requires the first operand to be in the register. If E_1 and E_2 do not have any common sub-expression, in order to get the shortest possible code :

(a) E_1 should be evaluated first

(b) E_2 should be evaluated first

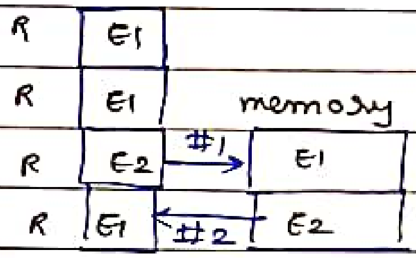
__/__/____

(c) Evaluation of E1 and E2 should necessarily be interleaved.

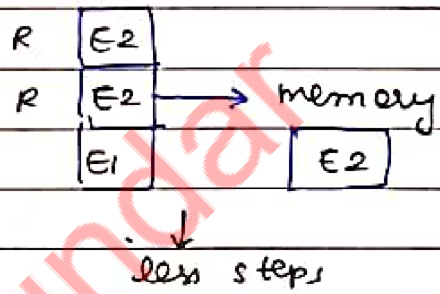
(d) order of evaluation of E1 and E2 is of no consequence.

(GATE 2004)

If E1 is evaluated first



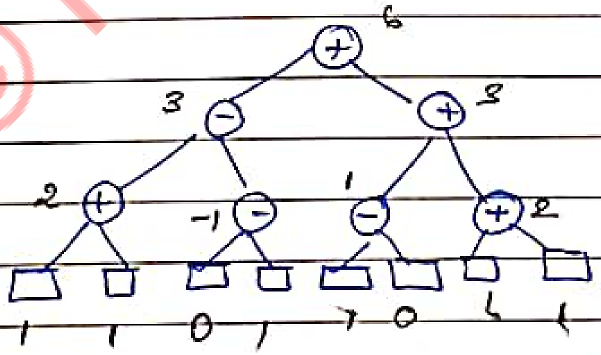
E2 first



(b) E2 should be evaluated first

Q: consider the expression tree shown. Each leaf represents a numerical value which can either be 0 or 1. over all possible choices of values at the leaves, the maximum possible value of the expression represented by the tree is ___.

(GATE 2014)



$0+0=0$ $\quad$ $0-0=0$

$0+1=1$ $\quad$ $0-1=-1$

$1+0=1$ $\quad$ $1-0=1$

$1+1=2$ $\quad$ $1-1=0$

max $(a+b)$ when a max and b max

max $(a-b)$ when a max and b min

$\therefore$ maximum possible value = $\underline{6}$

Summary:

✓ 2 solved problems on TAC.