UNIT-4

PART-C

1. What is three Address code? what is its type? how is it implemented?

(c) Intermediate Code Representation:

In the analysis - Synthesis model of a Comp. the front-end of a compiler translates a source program into an independent intermediate code. then the back-end of the compiler uses this I.C to generate the target code.

Three Address code: (Seq. of 3 Address Statement)

→ Statement involving more than 3 references (2 operands, 1 results)

⇒ $x = y$ op. z x, y, z → will have address (memory location)

→ Sometimes less than 3 references called as 3-Address code.

Representation of 3 Add. code.

- Quadruples
- Triples
- indirect triples

Eg: a + b * c + d → 3. Add. code form of Ic

temporary variables
$T_1 = b * c$
$T_2 = a + T_1$
$T_3 = T_2 + d$ → 3 Ac to represent any statement

→ generated by compiles for implementing code optimization

Quadruples

each instructions into 4 diff fields

→ op. arg1, arg2, result.
- op → internal code of the operator
- arg1, arg2 → Storing 2 operands
- result → Storing result.

Triples

Representation

→ References to instructions are made

→ Temporary variables are not used.

Indirect triples.

→ enhancement over triples Representation

→ Additional instructions, to point the pointer to triples

→ uses optimizers to easily re-position to optimized code.

Eg: a + b × c / e ↑ f + b × a.

$T_1 = e \uparrow f$
$T_2 = b \times c$
$T_3 = T_2 / T_1$
$T_4 = b \times a$
$T_5 = a + T_3$
$T_6 = T_5 + T_4$

Chas:

→ generated by compiles for implementing code optimization

→ 3 Addresses to represent

→ implemented as a record with Add. field.

2. Using Backpatching, generate an intermediate code for A < B OR C < D AND P < Q

→ Backpatching process of fulfilling unspecified info.

→ It uses semantic actions during the process of code generation

Steps: 1) generation of the production table

2) we have to find the TAC (Three. A. C) for the gu by backpatching

A < B OR C < D AND P < Q

100 | if A < B goto 106
101 | goto 102
102 | if C < D goto 104
103 | goto 107
104 | if P < Q goto 106
105 | goto 107

OR

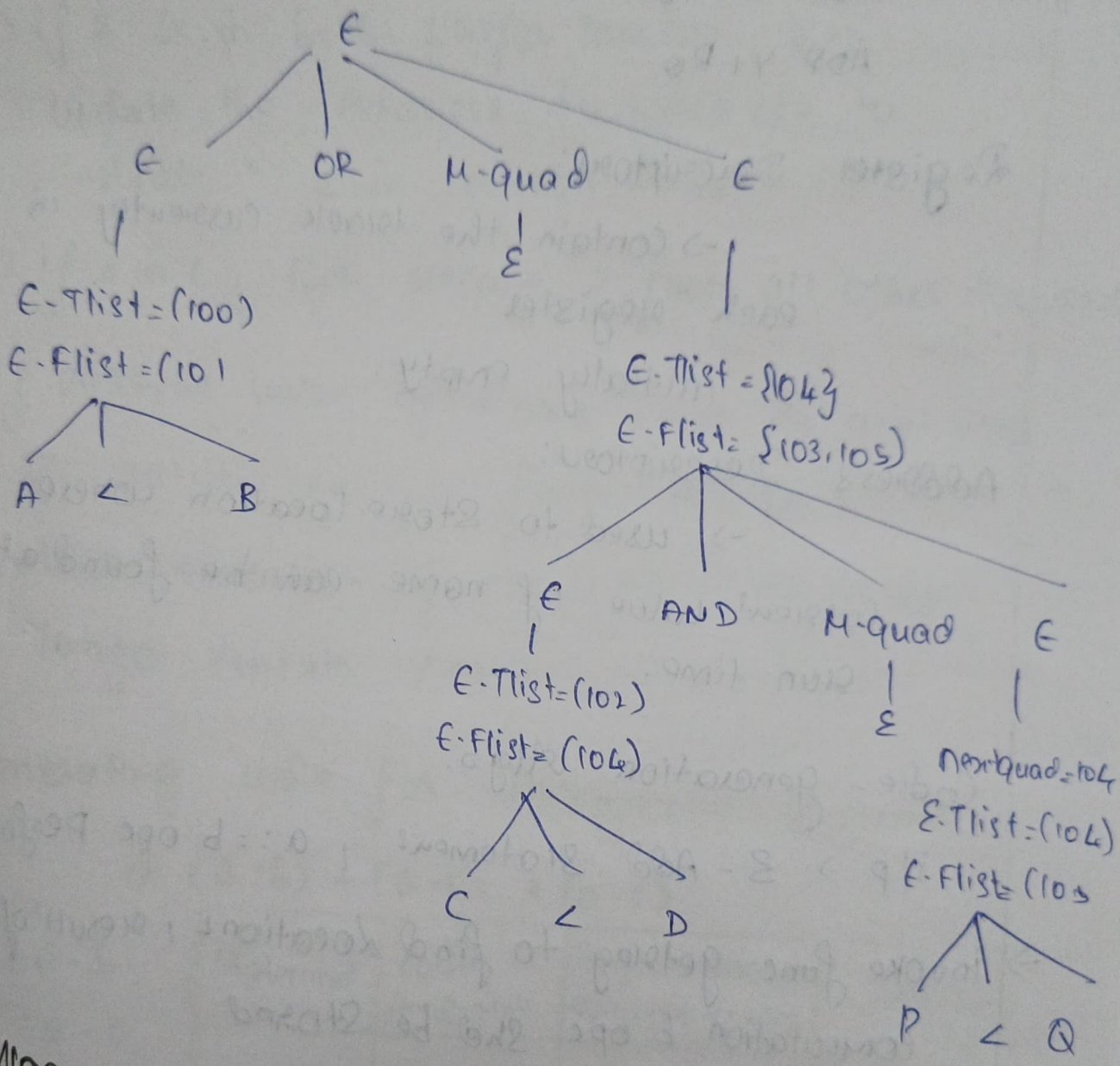
106: (true)
107: (false)

Backpatch (E1.Flist, 102)
E.Tlist = {100, 104}
E.Flist = {103, 105}

Backpatch (E1.Tlist, 104)
M.quad = next quad
E.Tlist := {104}
E.Flist = {103, 105}

AND

3) Make parse tree for ~~intermediate~~ exp.



3. ~~Algo~~ ~~issues~~ in code generation with eg:

Code generator: Converts the intermediate representation of source code into a form that can be readily executed by machine

Eg:

```
MOV x, R0
ADD y, R0
```

Register Descriptors:

→ Contain the track currently in each register

→ initially empty.

Address Descriptors:

→ used to store location where current value of name can be found at run time.

Code-generation Algo.

i/p → 3-Add statement, a := b opc performs Ad

⇒ invoke func getreg to find location L, result of computation b opc shd be stored

→ Address description for y to determine y'.

→ if value of y in memory & prefer register y'

→ if not, generate instruction MOV y', L to place a copy of y in L.

→ generate OP z', L, z' → current location.

→ if z is in both prefer location, then update the Address descriptor of x to indicate x is in location L.

→ if x in L, then remove x from all other descriptors

→ if current value of y or z have no next uses or not live on exit from the block

→ After execution of x := y op z those no longer contain y or z

generating code for Assignment Statement

d: (a-b) + (a-c) + (a-c)

3-A-Co

t := a-b
u := a-c
v := t+u
d := v+u

Statement	code generated	Register Discriptor Register empty	Address descriptor
t := a-b	MOV a, R0 SUB b, R0	R0 contain t	t in R0
u := a-c	MOV a, R1 SUB c, R1	R0 contains t R1 Contains u	t in R0 u in R1
v := t+u	ADD R1, R0	R0 contains u R1 contains u	v in R1 u in R1
d := v+u	ADD R1, R0 MOV R0, d	R0 contains d	d in R0 d in R0 & memory

D: (a - b) * (a - c) + (a - c)

```
t1 := a - b
t2 := a - c
t3 := t1 * t2
t4 := a - c
D := t3 + t4
```

```
t1 := a - b
t2 := a - c
t3 := t1 * t2
```

```
t1 := a - b
t2 := a - c
t3 := t1 * t2
ta := t2
D := t3 + t4
```

4. Various methods for translating Boolean exp.

→ Boolean exp have 2 primary purposes

→ Compute logical values (but conditional exp)

→ Composed of boolean operators.

Methods:

2 principal methods.

→ to encode T & F numerically & to evaluate a boolean exp to an arithmetic.

→ 1 - T , 0 - F

→ implementing boolean exp in flow-control statements

⇒ Numerical representation.

Expression evaluate from L to R. as with. exp.

Eg: a or b and not c → (TAC)

t1 := not c

t2 := b and t1

t3 := a or t2

Relational exp :- if a < b then 1 else 0.

100: if a < b goto 103 101: t := 0

102: goto 104

103: t := 1

104:

⇒ Short circuit code:

→ can convert boolean to TAC without generating code for any of the boolean op.

→ AKA short circuit / jumping code.

→ can evaluate bool. exp without code.

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

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 := t2 & t3
113: t5 := t1 OR t4

⇒ flow-of control statements:

Translation of bool. to TAC in context of if, if-then-else, while.

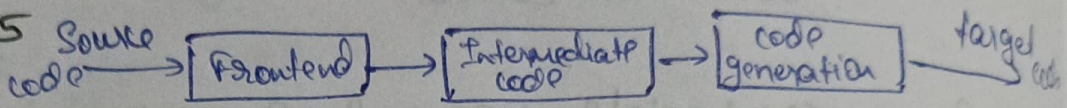
S → if E then S1 $
| if E then S2 else S2
| while E do S1

E → Boolean exp to be translated.

S → allows control to flow

S.next → translation after S. code.

PART-C UNIT 5



1. Optimization tech:

4m/12

→ Eliminating the unwanted code line

→ Rearranging the statement of code.

Ad:

Optimized code → faster execution speed, memory efficiency. better performance.

*Compile time evaluation: Shifting of computations from runtime to compile time

↳ folding: Computation of constant is done at compile time instead of execution time.

$A = (22/7) * d.$

↳ Const Propagation: value of the variable is replaced.

$Pi = 3.14$, $r = 15$ $Pi * r_1 * r_2$

*Loop-invariant Computation

→ Moving some amt of code outside the loop & placing it just before entering in loop.

Eg:
```
while (i<= max-1)
{
  sum= sum+ a[i];
}
```
=>
```
n= max-1;
while (i<=n)
{ sum= sum+ a[i];
}
```

* Strength reduction:
  -> the strength of certain operators is higher (i.e * than +) if higher replaced by lower

Eg.
```
for (i=1; i<=50; i++)
{...
 count= i * 7;
 ...
}
```
=>
```
temp= 7;
for(i=1; i<=50; i++)
{...
 count= temp;
 temp= temp+7;
}
```

* Dead code elimination:
  a value is said to be dead, if the value contained init is never been used

Ex:
```
i= j
...
x= i+10;
...
```
=>
```
i= 0;
if (i=1)
{a= x+5;
}
```

* Common Subexpression elimination:

→ appearing repeatedly in the prog. which is computed previously.

```
t1 = 4 * i                t1 = 4 * i
t2 = n[t1]                t2 = n[t1]
t3 = 4 * j        =>      t3 = 4 * j
t4 = 4 * i                t5 = n;
t5 = n;                   t6 = b[t1] + t5
t6 = b[t4] + t5
```

* Copy propagation:

→ used of one variable instead of another

```
Eg:  x = Pi                  -> area = Pi * r * r.
     ...
     area = x * r * r;
```

* Code Movement.

→ reduce the size of code & frequency of execution of code. to obtain time complexity.

```
Eg: for (i=0 ; i<=10 ; i++)          z = y * 5
    { x = y * 5;            =>       for (i=0; i<=10; i++)
      ...                            { x = z;
      k = (y * 5) + 50;                ...
    }                                  k = z + 50;
                                     }
```

* Loop Optimization:

→ code optimization performed on inner loop

→ code Motion, unrolling, fusion.

→ loop invariant method

→ induction variable & strength reduction.

Data flow Analysis: 4m

→ flow of Data in control flow graph.

→ with the help of Analysis, optimization can be done.

→ tech how Data flows through Prog.

Types:

→ Reaching Definitions Analysis:

→ the pt in prog where Definition reaches the particular use of variable / exp

→ Live variable Analysis:

→ value for some future computation

→ Available expression Analysis:

value has been computed & reused.

→ Const propagation Analysis:

→ tracks the particular value is const.

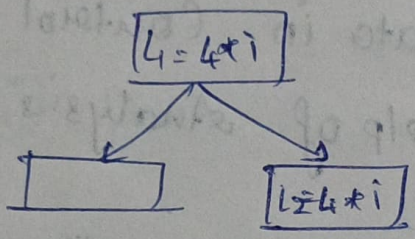
Ad: improved code quality.
Better errors Detection.
increased understanding of prog. Behaviour

Basic terminologies.

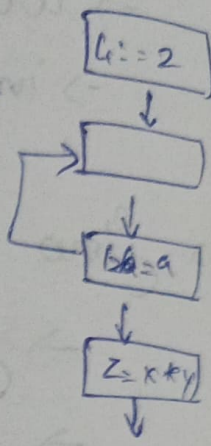
Definition pt, evaluation pt, reference pt.

Data flow properties

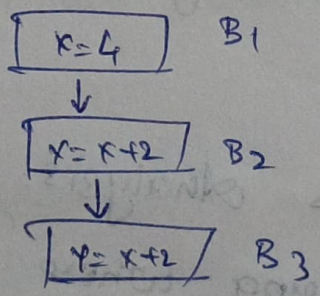
*Available exp:



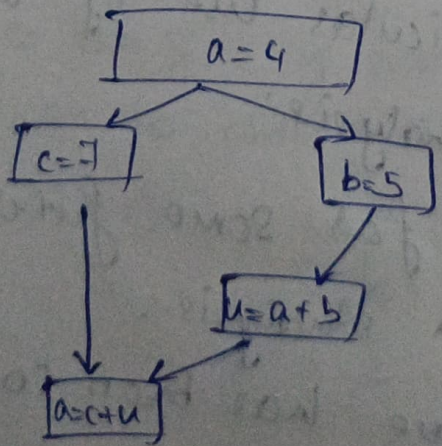
E.P



*Reaching Definition



*Live Variable



* Busy Exp:

busy along a path if its evaluation exist along the path.

Peephole optimization :- 4m

→ type of code optimization performed on a small part of code.

→ small set of instructions in a segment of code

⇒ Objectives

↳ to improve performance, memory footprint, code size

⇒ optimization tech.

* Redundant load & store elimination.
↳ redundancy elimination.

```
y = x + 5          y = x + 5;
i = y       =>     w = i * 3;
z = i
w = z * 3
```

* Const folding: → simplified by users itself.

x = 2 * 3; → x = 6.

* Strength Reduction: highes execution time Replaced by lower.

Y = X * 2

Y = X + X; or Y = X << 1;

* null sequences: useless op. Deleted.

```
a := a + 0;     a := a | 1;
a := a * 1;     a := a - 0;
```

* Combine operation : several op are replaced by a single equivalent op.
* Deadcode Elimination: A part of code which can never be executed