Chapter : 5 Intermediate Code generation

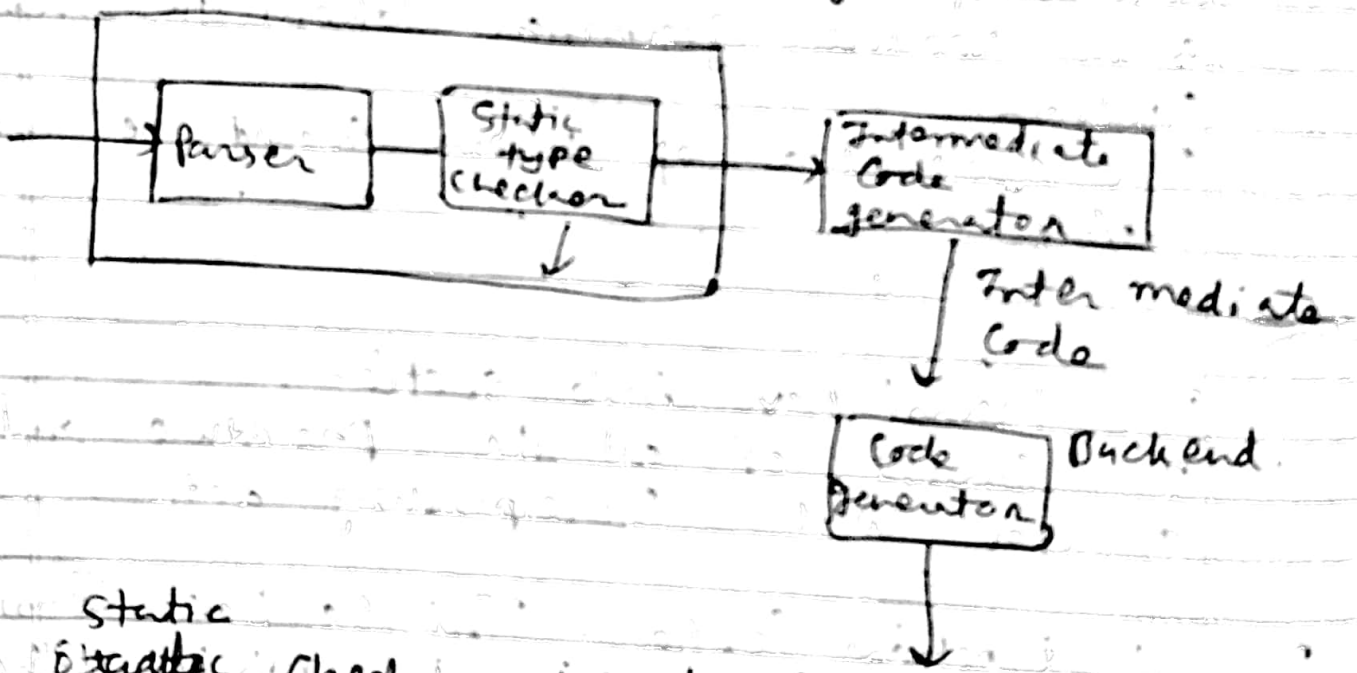
What is an Intermediate Code ?

→ The task of ~~Compiler~~ Compiler is to Convert the Source program into Machine Program. This activity can be done directly, but it is not always possible to generate such a Machine code directly in one pass.

→ Then, typically Compilers generate an easy to represent form of Source language which is called Intermediate language.

→ The generation of Intermediate language leads to Efficient Code generation.

→



Static checking includes type checking. Which ensures that operators are applied to Compatible operands. It also includes any Syntactic Checks that remain after parsing. for example. Static checking assures that break statement in C is enclosed within a while-, for- or Switch-Statement an error is generated if such an Enclosing statement does not exist.

DATE

Benefits of Intermediate code generation:

→ 1 A Compiler for different machines can be created by attaching different backend to the existing front ends of each machine.

2 A Compiler for different source languages (on the same machine) can be created by proving different front ends for corresponding source languages to existing back end.

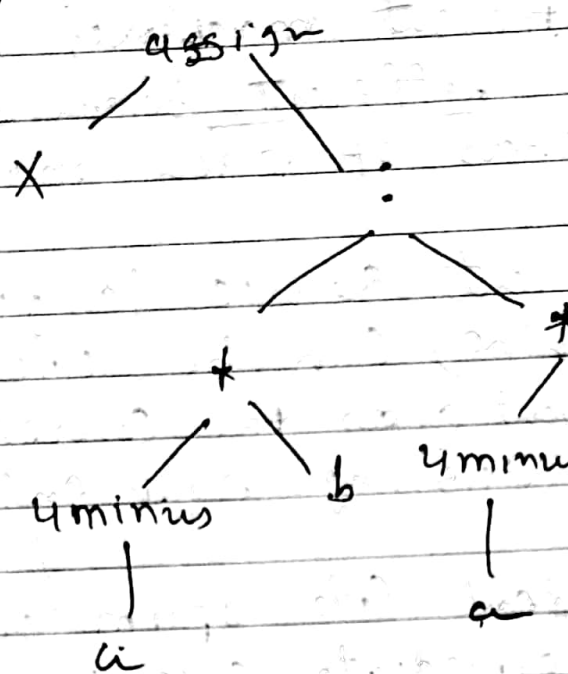
3 A machine Independent code optimizer can be applied to intermediate code in order to optimize the code generation.

Different Intermediate forms:
There are mainly three types of intermediate code representations:

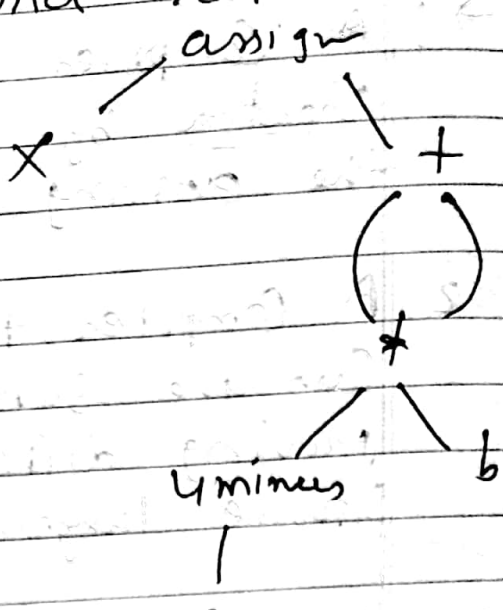
1 Abstract Syntax tree / DAGs:
2 polish notation
3 Three address code.

1 Abstract Syntax tree:
→ The natural hierarchical structure is represented by syntax trees. DAGs is very much similar to syntax tree but they are in more compact form.
→ The code being generated as intermediate should be such that the remaining processing of the subsequent phases should be easy.

Input String x := -a + b * + -a * b
for Syntax tree and DAG representation



Syntax tree



DAG

2 polish notation :

→ Basically the linearization of Syntax trees is polish notation. In this representation, operator can be easily associated with the corresponding operands. This is the most natural way of representation in expression evaluation.

→ The polish notation is also called as prefix notation in which the operator occurs first and then operands are arranged.

(a+b) * (c-d)

(+ab) * (-cd)

* +ab -cd.

There is Reverse polish notation which is used using postfix or posix representation.

$x := -a * b + -a * b$

$x := a - * b + a - * b$

$x := (a - b *) + (a - b *)$

$x := a - b * a - b * + :=$

3 Three address code:

→ In three address code form at the most three addresses are used to represent any statement.

The general form of three address code representation is:

$a := b \text{ op } c$

where a, b and c are the operands that can be names, constants, compiler generated temporaries and op represents the operator.

→ Only single operation at right side of the expression is allowed at a time.

for the $a = b + c + d$ expression:

$t_1 := b + c$

$t_2 := t_1 + d$

$a := t_2$.

→ Here t_1 and t_2 are the temporary names generated by the compiler.

Translate the arithmetic expression

→ a * -(b+c) into

1) Syntax tree
2) Postfix notation
3) Three address code.

→ 1) Syntax tree:

a * -(b+c)

Step: 1 Convert the expression into postfix notation.

a * -(bc+)

a * bc+-

a bc+-*

Postfix notation: **abc+-***

→ Step: 2

P1 = leaf(a, entry-a)
P2 = leaf(b, entry-b)
P3 = leaf(c, entry-c)
P4 = node('+', P2, P3)
P5 = node('uminus', P4)
P6 = node('*', P1, P5)

Step: 3

Syntax tree:

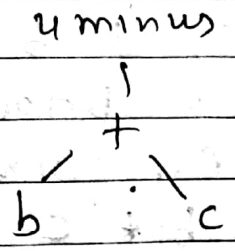
1)

a	b	c

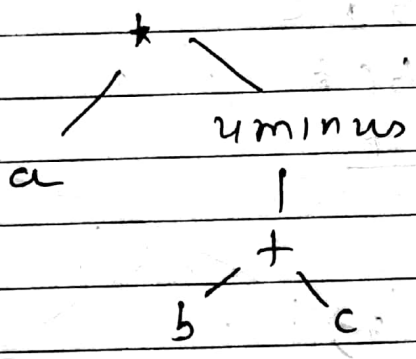
2)

a

3) [a]



4)



c) post fix notation:

a b c + - *

a * - T1 $\quad T_1 = bc+$

T2 $\quad T_2 = aT_1 -*$

Substituting in backward direction:

T2

a T1 - *

a b c + - *

3) Three address code:

T1 := b + c

T2 := uminus T1

T3 := a * T2

Step : 1 Convert the expression from infix to postfix.

Step : 2 Make use of functions mknode( ), mkleaf(id, ptr) and mkleaf(num, val).

→ mknode (op, left, right) :

→ This function creates a node with the field operator having operator as label, and the two pointers to the left and right.

mkleaf (id, entry) :

→ This function creates an identifier node with label id and an entry to the symbol table is given by entry.

mkleaf (num, val).

→ This function creates a node for number with label num and val is for value of that number.

Step : 3 Draw Syntax tree or DAG :

QB-12 (a + b * c) ↑ (b * c) + b * c ↑ a.
Draw syntax tree and DAG for following statement.

Step - 1 : Convert the expression into postfix.

(a b c * +) ↑ (b c *) + b * c ↑ a

(a b c * +) ↑ (b c *) + b * c a ↑

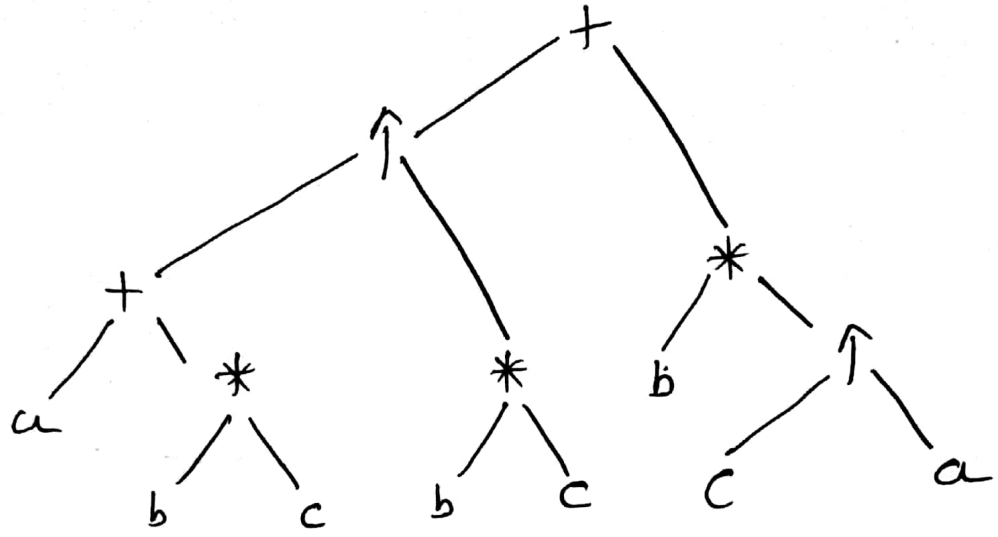
(a b c * + b c * ↑) + b c a ↑ *

a b c * + b c * ↑ b c a ↑ * +

step : 2 :

P1 : leaf (id, entry - a)
P2 : leaf (id, entry - b)
P3 : leaf (id, entry - c)
P4 : node ('*', P2, P3)
P5 : node ('+', P1, P4)
P6 : leaf (id, entry - b)
P7 : leaf (id, entry - c)
P8 : node ('*', P6, P7)
P9 : node ('↑', P5, P8)
P10 : leaf (id, entry - b)
P11 : leaf (id, entry - c)
P12 : leaf (id, entry - a)
P13 : node (↑, P11, P12)
P14 : node (*, P10, P13)
P15 : node (+, P9, P14)

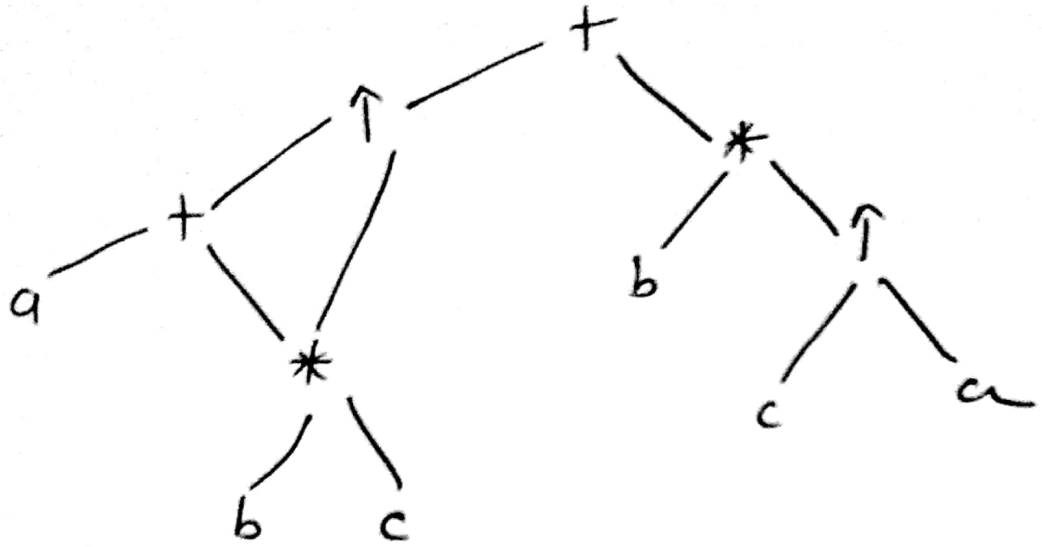
Step : 3 Syntax tree :



Step : 4 :

p_1 : leaf (id, entry - a)
p_2 : leaf (id, entry - b)
p_3 : leaf (id, entry - c)
p_4 : node ('*', p_2, p_3)
p_5 : node ('+', p_1, p_4)
p_6 : leaf (id, entry - b) = p_2
p_7 : leaf (id, entry - c) = p_3
p_8 : node ('*', p_2, p_3) = p_4
p_9 : node ('↑', p_5, p_4)
p_{10} : leaf (id, entry - b)
p_{11} : leaf (id, entry - c)
p_{12} : leaf (id, entry - a)
p_{13} : node ('↑', p_{11}, p_{12})
p_{14} : node ('*', p_{10}, p_{13})
p_{15} : node ('+', p_9, p_{14})

Step : 5 DAG :



prepared by kinjal patel

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

Draw syntax tree and DAG for the following statement.

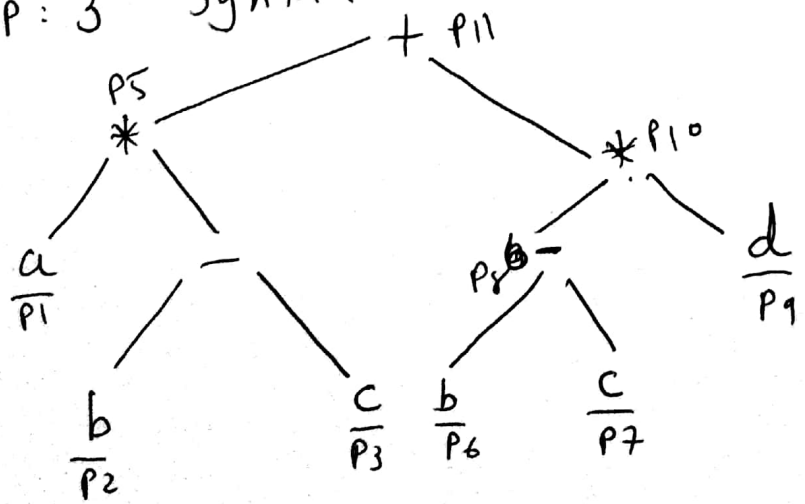
Step: 1 : Convert the expression into postfix notation:

postfix : $abc - * bc - d * +$

Step - 2 :

P_1 : leaf (id, entry - a)
P_2 : leaf (id, entry - b)
P_3 : leaf (id, entry - c)
P_4 : node ('-', P_2, P_3)
P_5 : node ('*', P_1, P_4)
P_6 : leaf (id, entry - b)
P_7 : leaf (id, entry - c)
P_8 : node ('-', P_6, P_7)
P_9 : leaf (id, entry - d)
P_{10} : node ('*', P_8, P_9)
P_{11} : node ('+', P_5, P_{10})

Step: 3 Syntax tree:



Three address code:

$t_1 := b - c$
$t_2 := a * t_1$
$t_3 := b - c$
$t_4 := t_3 * d$
$t_5 := t_2 * t_4$

Step : 4

P_1 = leaf (id, entry - a)

P_2 : leaf (id, entry - b)

P_3 : leaf (id, entry - c)

P_4 : node ('-', P_2, P_3)

P_5 : node ('*', P_1, P_4)

P_6 : leaf (id, entry - b) = P_2

P_7 : leaf (id, entry - c) = P_3

P_8 : node ('-', P_2, P_3) = P_4

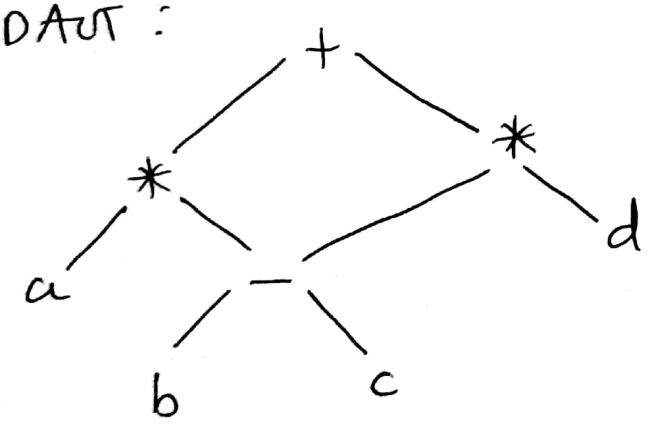
P_9 : leaf (id, entry - d)

P_{10} : node ('*', P_4, P_9)

P_{11} : node ('+', P_5, P_{10})

step : 5 :

DAG :



Three address Code :

$t_1 := b - c$

$t_2 := a * t_1$

$t_3 := t_1 * d$

$t_4 := t_2 + t_3$

ex: a = (a + b * c) ↑ (b * c) + b * c

Draw Syntax tree and DAG.

Step 1: Convert the expression into postfix notation

(a + b * c) ↑ b * c + b * c

a b c * + ↑ b c * + b * c

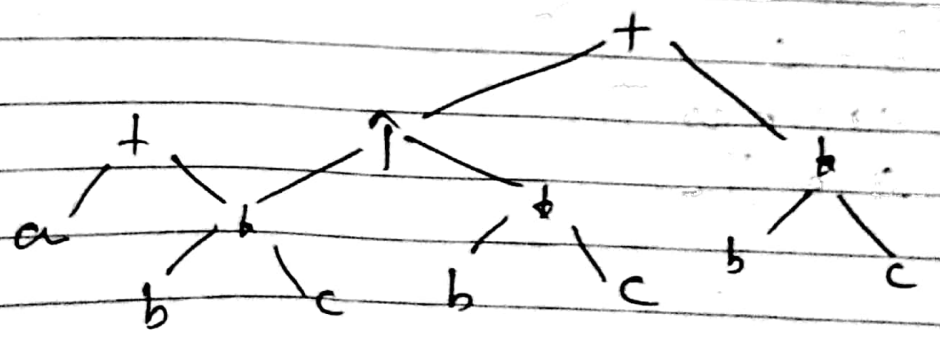
a b c * + b c * ↑ + b * c

a b c * + b c * ↑ + b c * +

postfix : a b c * + b c * ↑ b c * +

Step 2:
P_1 : leaf (id, entry - a)
P_2 : leaf (id, entry - b)
P_3 : leaf (id, entry - c)
P_4 : node ('*', P_2, P_3)
P_5 : node ('+', P_1, P_4)
P_6 : leaf (id, entry - b)
P_7 : leaf (id, entry - c)
P_8 : node ('*', P_6, P_7)
P_9 : node ('↑', P_5, P_8)
P_{10} : leaf (id, entry - b)
P_{11} : leaf (id, entry - c)
P_{12} : node ('*', P_{10}, P_{11})
P_{13} : node ('+', P_9, P_{12})

Step 3: Syntax tree:



P_1 : leaf (id, entry-a)

P_2 : leaf (id, entry-b)

P_3 : leaf (id, entry-c)

P_4 : node ('+', P_2, P_3)

P_5 : node ('+', P_1, P_4)

P_6 : leaf (id, entry-b) = P_2

P_7 : leaf (id, entry-c) = P_3

P_8 : node ('+', P_2, P_3) = P_4

P_9 : node ('↑', P_5, P_4)

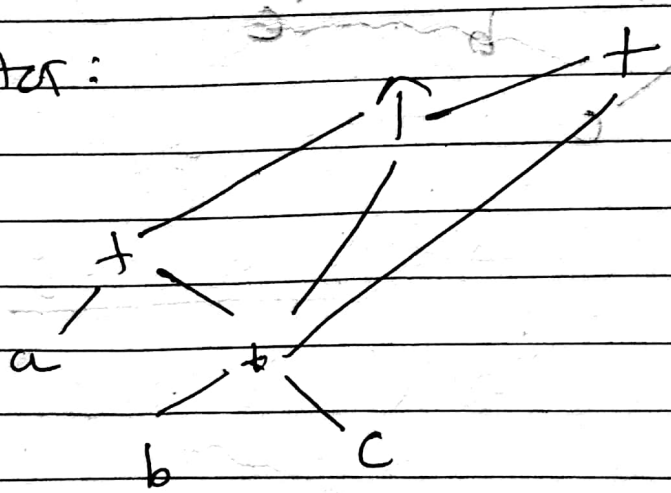
P_{10} : leaf (id, entry-b) = P_2

P_{11} : leaf (id, entry-c) = P_3

P_{12} : leaf ('+', P_2, P_3) = P_4

P_{13} : node ('+', P_9, P_4)

DAG:



→ Implementation of three Address Code :

→ Quadruple representation :

→ The quadruple is a structure with at the most four fields such as op, arg1, arg2, result.

op field is used to represent the internal code for operator, the arg1 and arg2 represent the two operands used and result field is used to store the result of an expression.

Consider the input statement

$$X := -a * b + -a + b$$

Three address code :		op	Arg1	Arg2	result
$t_1 := uminus\ a$	(0)	uminus	a		t_1
$t_2 := t_1 * b$	(1)	*	t_1	b	t_2
$t_3 := -a$	(2)	uminus	a		t_3
$t_4 := t_3 * b$	(3)	*	t_3	b	t_4
$t_5 := t_2 + t_4$	(4)	+	t_2	t_4	t_5
$x := t_5$	(5)	=	t_5		x.

Triples:

→ In the triple representation the use of temporary variables is avoided by referring the pointers in the symbol table. its position.

→ $x := -a * b + -a * b$

Triple representation is as given below:

→

Number	op	Arg1	Arg2
(0)	uminus	a	
(1)	*	(0)	b
(2)	uminus	a	
(3)	*	(2)	b
(4)	+	(1)	(3)
(5)	=	x	(4)

Indirect triples:

→ In the Indirect triple representation the listing of triples is been done. And listing pointers are used instead of using statements.

Number	op	Arg1	Arg2	Statement	
(0)	uminus	a		(0)	(11)
(1)	*	(11)	b	(1)	(12)
(2)	uminus	a		(2)	(13)
(3)	*	(13)	b	(3)	(14)
(4)	+	(12)	(14)	(4)	(15)
(5)	=	x	(15)	(5)	(16)

Triple: Note that the temporary, t does not actually appear in a triple, Since temporary values are referred to by their position in triple structure.

1

→ With quadruples, if we move an instruction that computes a temporary t, then the instructions that use t require no change.

optimization ✓

→ With triples, the result of an operation is referred to by its position. So moving an instruction may require us to change all references to that result.

optimization ×

→ Indirect triples consist of a listing of pointers to triples, rather than a listing of triples themselves.

→ With indirect triples, an optimizing compiler can move an instruction by reordering the instruction list, without affecting the triples themselves.

optimizer ✓

2 The format of quadruple is (op, operand1, operand2, result).

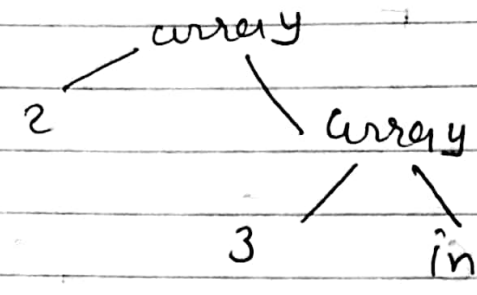
→ The format of triple is (op, operand1, operand2)

→ The format of indirect triple is (op, operand1, operand2) but it makes use of two tables.

Type checking :

→ To do type checking a Compiler needs to assign a type expression to each Component of the Source program.

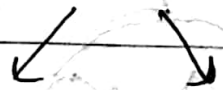
type expression :
int[2][3] :



→ Type checking has the potential for catching errors in programs.

→ An implementation of a language is strongly typed if a Compiler guarantees that the programs it accepts it will run without type errors.

Rules for type checking :
Type checking can take two forms :



Type Synthesis builds up the type of an expression from the types of its sub expressions. It requires names to be declared before they are used.

The type of $E_1 + E_2$ is defined in terms of the types of E_1 and E_2.

A typical rule for type synthesis has the form:
if f has type $s \to t$ and x has type s,
then expression f(x) has type t

f and x denote expressions.
$s \to t$ denotes a function from s to t

Type inference determines the type of language construct from the way it is used.

The typical rule for type Inference has the form:
If f(x) is an expression
then some α and β, f has type $\alpha \to \beta$,
and x has type α.

The term is often applied to the problem of inferring the type of function from its body.

List out motivations for back patching.

1) In boolean expression suppose there are two expressions E_1 && E_2. If the expression E_1 is false then there is no need to compute rest of the expression. But if we generate the code for the expression E_1 we must know where to jump on encountering the false E_1 expression. But this can be accomplished by the back patching technique. ex: if a<b then t=1 else t=0

```
100 if (a < b) goto 103
101 t = 0
102 goto 104
103 t = 1
104
```

three address code.

leaving the labels as empty and filling them later is called Backpatching.

PAGE No.
DATE

?) for the flow of control statement. (Such as if, else, while) the jumping location can be identified using back patching.

→ Back patching usually refers to the process of resolving forward branches that have been planted in the code e.g. as if statements when the value of the target becomes known. e.g when the closing brace or matching else is encountered.

Three address Code :

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

quadruple :

$t_1 : a * b$
$t_2 : uminus\ t_1$
$t_3 : c * d.$
$t_4 : t_2 + t_3.$

Location	op	Arg_1	Arg_2	Result
(0)	*	a	b	t_1
(1)	uminus	t_1		t_2
(2)	*	c	d	t_3
(3)	+	t_2	t_3	t_4

Triple :

Location	op	Arg_1	Arg_2
(0)	*	a	b
(1)	uminus	(0)	
(2)	*	c	d
(3)	+	(1)	(2)

Indirect triple :

Location	op	Arg_1	Arg_2
(0)	*	a	b
(1)	uminus	(11)	
(2)	*	c	d
(3)	+	(12)	(13)

Statement	
(0)	(11)
(1)	(12)
(2)	(13)
(3)	(14)