B.E.

Seventh Semester Examination, May-2009

Compiler Design (CSE-405-E)

Note : Attempt any ***FIVE*** questions. All questions carry equal marks.

Q. 1. (a) What is a compiler? Write Short note on LEX tool.

Ans. Compiler : Compiler is a software which takes as input a source program in high level language & produces the sequence of machine code instructions as output.

Source Program
Compiler
Machine code

Lex Tool : LEX is a parser generator tool. It is basically a program that generates the lexers LEX is commonly used with the yacc parser generator. LEX, originally written by mike lesk, is the standard lexical analyzer on UNIX systems. Lex reads an input file specifying the lexical analyzer and outputs code implementing the lexer in the C programming language.

Q. 1. (b) Define a non-deterministic finite state automata (nDFA). Write an algorithm to simulate nDFA.

Ans. Non-Deterministic Finite Automata : NFA is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ where,

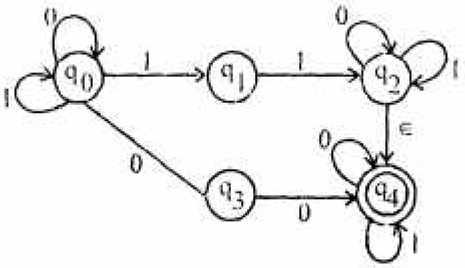
(i) Q is finite non-empty set of states.

(ii) Σ is finite non-empty set of inputs.

(iii) δ is a transition function mapping from $Q \times \Sigma$ to 2^{ϕ}, which is power set of Q, the set of all subsets of Q.

(iv) $q_0 \in Q$ is the initial state and

(v) $F \subseteq Q$ is the set of final states.



Algorithm for simulating NDPA

```
        P = S;
        While (input not empty)
            {
            scan (c);
            T = $\phi$ ;
            For (each $\Delta(q,c)$ where $q \in p$ )
            $T = T \cup \Delta(q, c)$ ;
        if $(T \subseteq \phi)$
            reject:
            $P = T$
        }
        accept iff $(P \cap F \neq \phi)$:
```

Algorithm for simulating NFA with $\in$ – transitions :

```
    $P = S$;
    while (true) {
    do {
    $T = P$;
    For (each $\Delta(q, \in)$ where $q \in p$ )
        $P = P \cup \Delta(q, \in)$ ;
        }
        while $(T \neq P)$ ;
        if (input empty)
        break;
```

Q. 2. (a) What is the role of lexical analyzer in compilation process. Define regular expression and give steps to convert a regular expression into a non-deterministic finite state automata.

Ans. Role of Lexical Analyzer : Lexical analyzer acts as an interface between the input source program to be compiled and the later stages of the compiler. The input program can be considered to be a sequence of characters. Lexical analyzer converts this character stream into a stream of valid words of the language, better known as tokens.

Regular Expression :

(i) Any terminal symbol in Σ, including $\in$ and ϕ are regular expressions.

(ii) Union of two regular expressions R_1 and R_2 written $R_1 + R_2$ is also a regular expression.

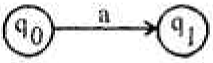
(iii) The concatenation of two regular expressions R_1 & R_2, written as R_1R_2 is also a regular expression.

(iv) The closure of a regular expression R is written as R* is also a regular expression.

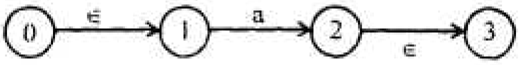
(v) If R is a regular expression then (R) is also a regular expression.

Steps to Convert Regular Expression to NFA :

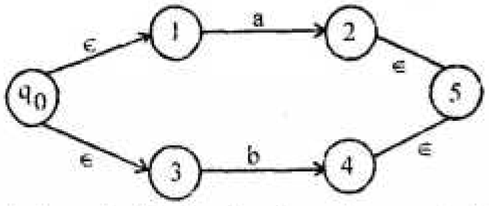
(i) Convert a state transition between two states for each input of i.e.,



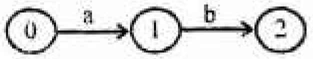
or



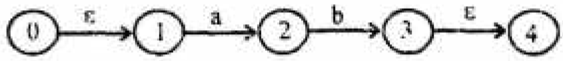
(ii) a+b or a/b can be thought of OR or a parallel circuit can be converted as follows :



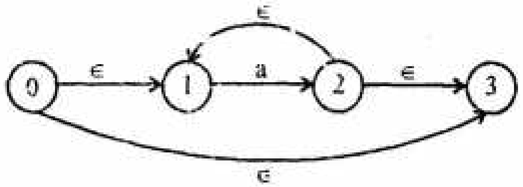
(iii) Intersection a.b can be thought of series circuit can be converted to NFA as below :



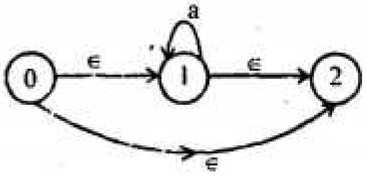
OR



(iv) Closure can be converted to NFA as a*



OR

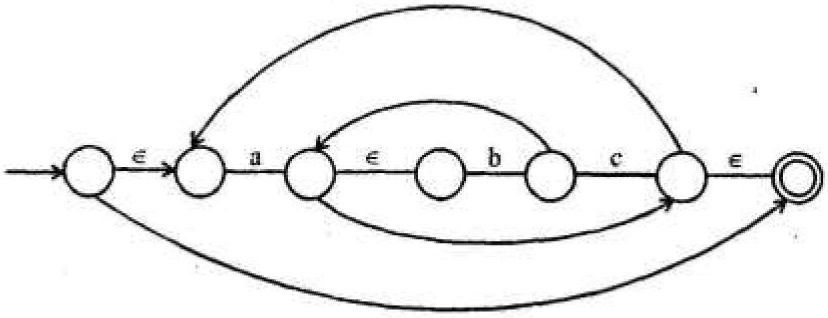


Q. 2. (b) Construct the transition diagram for the following regular expressions.

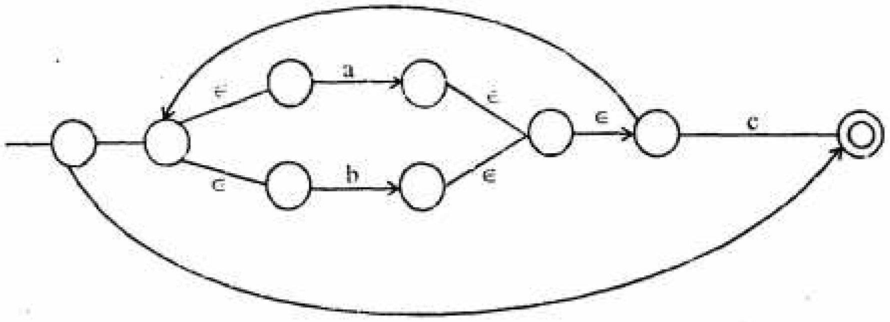
(i) $(ab^*)^*$ **(ii)** $((a|b)c^*)^*$

(iii) $(a|b)^*abb$

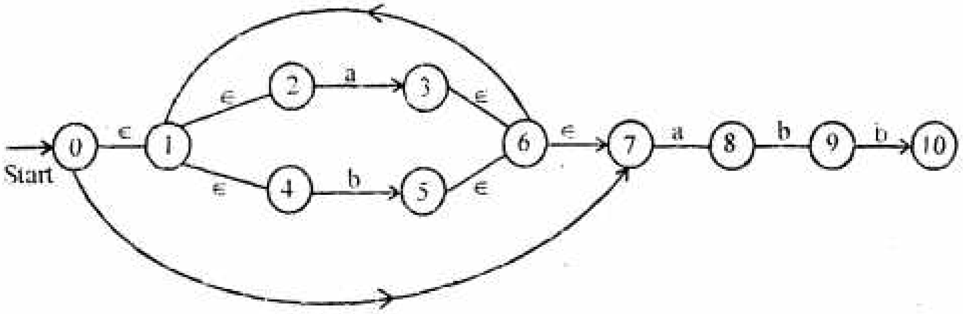
Ans. (i) $(ab^*)^*$ **:**



(ii) ((a|b)c*)* :



(iii) (a|b)*abb :



Q. 3. (a) Explain the algorithm for the Recursive Predictive Parser.

Ans.

```
Procedure match (t : token);
begin
if lookhead = t then
lookahead : = next token
else error
end;
procedure type;
begin
if lookahead is in {integer, char, num} then
simple
else if lookahead = '↑' then begin
match ('↑'): match(id)
end
else if lookahead =array then begin
match(array); match ('['); simple; match (']'); match (of); type
end
```

```
else error
end;
procedure simple;
begin
if lookahead = integer then
match (integer)
else if lookahead = char then
match (char)
else if lookanead = num then begin
match (num); match (dot dot); match (num)
end
else error
end;
```

Q. 3. (b) (i) Give the Chomsky classification of grammars.

Ans. Chomsky Classification of Grammar : Chomsky classified grammar into four categories (type 0– type 3)

Type 0 : Phrase structured or unrestricted grammar production form

$$\alpha \to \beta$$

Where $\alpha, \beta \in (V_N \cup T)^+$

Type 1 : Context sensitive grammar,

If production is of the form

$$\phi A \psi \to \phi \alpha \psi$$

Where, $\phi, \psi \in (V_N \cup T)^*$

$$A \in V_N$$

$$\alpha \in (V_N \cup T)^+$$

$$\alpha \neq \in$$

Type 2 : Context free grammar also sometimes called backus Naur From (BNF). In this grammar there is no left or right context restrictions & productions are of the form

$$A \to \alpha$$

$$A \in V_N$$

$$\alpha \in (T \cup V_N)^*$$

Type 3 : Regular Grammar : A grammar in which production is of the form

$$A \to a$$

$$A \to aB$$

Where, $A, B \in V_N$

$a \in T$

Q. 3. (b) (ii) Give the highest type of following grammar, with explanation :

$S \rightarrow a$, $S \rightarrow bB$, $BA \rightarrow bA$, $CA \rightarrow AB$.

Ans. $S \rightarrow a / bB$

$BA \rightarrow bA$

$CA \rightarrow AB$

(i) $S \rightarrow a$

It is of type 0

(ii) $S \rightarrow bB$

It is of type 3

(iii) $BA \rightarrow bA$

It is of type 1

(iv) $CA \rightarrow AB$

It is of type 0

So, highest type is zero.

Q. 4. (a) Make left and right derivation using top down and bottom up strategy to derive a statement $w = id + (id + id) * id$ using following grammar :

$E \rightarrow E + E$

$E \rightarrow E * E$

$E \rightarrow (E)$

$E \rightarrow id$

Check whether the grammar is ambiguous.

Ans. $E \rightarrow E + E$

$E \rightarrow E \times E$

$E \rightarrow (E)$

$E \rightarrow id$

Using Top-Down :

$E \rightarrow E + E$

$E \rightarrow id + E$

$E \rightarrow id + E * E$

$E \rightarrow id + (E) * E$

$E \rightarrow id + (E + E) * E$

$E \rightarrow id + (id + E) * E$

$E \rightarrow id + (id + id) * E$

$E \rightarrow id + (id + id) * id$

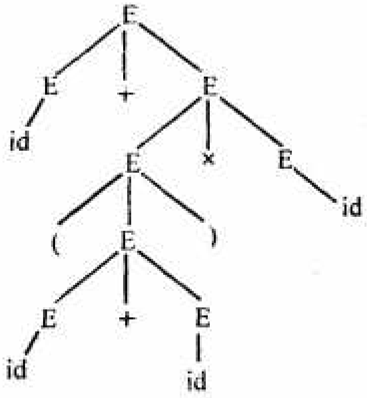


Fig. Left Most Derivation

Right Most Derivation :

$E \rightarrow E * E$

$E \rightarrow E * id$

$E \rightarrow E + E * id$

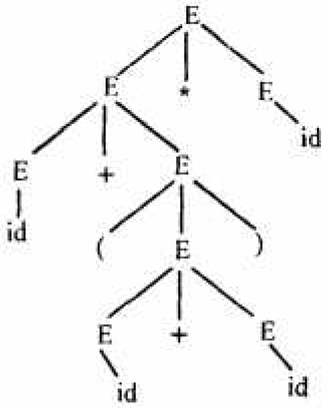
$E \rightarrow E + (E) * id$

$E \rightarrow E + (E + E) * id$

$E \rightarrow E + (id + id) * id$

$E \rightarrow id + (id + id) * id$

$E \rightarrow id + (id + id) * id$



Rightmost Derivation

The Grammar is Ambigous

Bottom-up :

$$W = id + (id + id) * id$$

$$W = E + (id + id) * id$$

$$W = E + (E + id) * id$$

$$W = E + (E + E) * id$$

$$W = E + (E + E) * E$$

$$W = E + (E) * E$$

$$W = E + E * E$$

$$W = E * E$$

$$W = E$$

Left Most Derivation :

$$W = id + (id + id) * id$$

$$W = id + (id + id) * E$$

$$W = id + (id + E) * E$$

$$W = id + (E + E) * E$$

$$W = E + (E + E) * E$$

$$W = E + (E) * E$$

$$W = E + E * E$$

$$W = E + E$$

$$W = E$$

Right Most Derivation

The grammar is ambiguous

Q. 4. (b) Remove the left recursion from grammar given in Q4.(a).

Ans.

$$E \rightarrow E + E$$

$$E \rightarrow E * E$$

$$E \rightarrow (E)$$

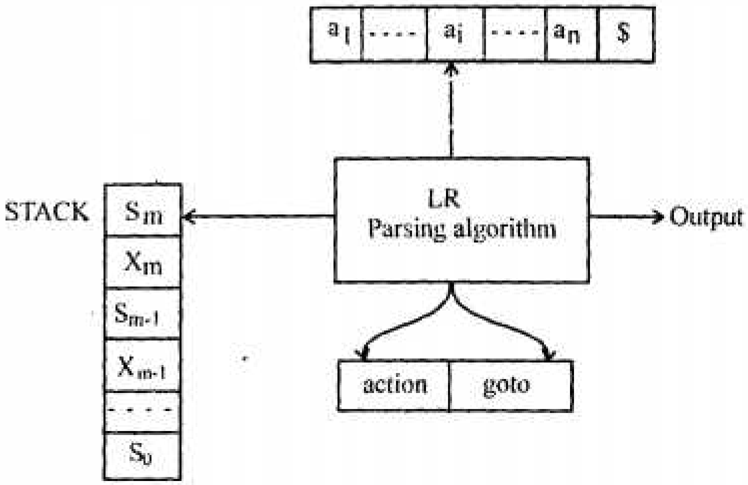
$$E \rightarrow id$$

Now, after removing left recursion the grammar is

$E \rightarrow (E)E'/idE'$
$E' \rightarrow +EE'/ * EE'/E$

Q. 5. (a) Explain the working and algorithm of LR parsers.

Ans. The schematic form of an LR parser is shown in fig. 1.



It consists of an input, an output, a stack, a driver program, & a parsing table that has two parts (action & goto). The driver program is the same for all the LR parsers, only the parsing table changes from one parser to another. The parsing program reads characters from an input buffer one at a time. The program uses a stack to store a string of the form $S_0X_1S_1X_2S_2......X_mS_m$, where S_m is on top. Each X_i is a grammar symbol and each S_i is a symbol called a state. Each state symbol summarizes the information contained in the stack below it and the combination of the state symbol on top of the stack.

Algorithm :

Input : An input string W and an LR parsing table with functions action and goto for a grammar G.

Output : If W is in L(G), a bottom-up parse for W; otherwise, an error indication.

Method : Initially, the parser has S_0 on its stack, where S_0 is the initial state and W$ in the input buffer. The parser then executes the program in fig. 2, until an accept or error action is encountered.

Fig. 2

```
Set ip to point to the first symbol of W$;
repeat forever begin
Let S be the state on top of the stack and a the symbol pointed to by ip;
if action [s, a] = shift s' then begin
push a then s' on top of the stack;
advance ip to the next input symbol
end
else if action [s, a] = reduce $A \rightarrow B$ then begin
pop 2*|B| symbols off the stack;
```

```
Let s' be the state now on top of the stack;
push A them goto [s', A] on top of the stack;
output the production $A \rightarrow \beta$
end
else if action [s, a] = accept then
return
else error ( )
end.
```

Q. 5. (b) Check whether following grammar is LR(0) grammar or not.

$S \rightarrow L = R$

$S \rightarrow R$

$L \rightarrow {}^*R$

$L \rightarrow id$

$R \rightarrow L$

Ans. (0) $S' \rightarrow S$

(1) $S \rightarrow R$

(2) $L \rightarrow {}^*R$

(3) $L \rightarrow id$

(4) $R \rightarrow L$

Starting with the closure $(S' \rightarrow S)$, we get I_0

I_0 : $S' \rightarrow S$

$S \rightarrow L = R$

$S \rightarrow .R$

$L \rightarrow .{}^*R$

$L \rightarrow .id$

$R \rightarrow .L$

I_1: $S' \rightarrow S.$

I_2 : $S \rightarrow L. = R$

$R \rightarrow L.$

I_3 : $S \rightarrow R.$

I_4 : $L \rightarrow {}^*.R$

$R \rightarrow L.$

$L \rightarrow .{}^*R$

$L \rightarrow .id$

I_5 : $L \rightarrow id.$

$I_6 : S \rightarrow L = .R$

$R \rightarrow .L$

$L \rightarrow .{*}R$

$L \rightarrow .{*}d$

$I_7 : L \rightarrow {*}R$

$I_8 : R \rightarrow .L$

$I_9 : S \rightarrow L. = R$

Canonical sets of LR(0) items.

Q. 6. (a) What is syntax directed definition, why are they important?

Ans. Syntax Directed Definition : A syntax directed definition is a generalization of a context-free grammar in which each grammar symbol has an associated set of attributes, partitioned into two subsets called the synthesized and inherited attributes of that grammar symbol.

Syntax-directed definitions are high level specifications for translations. They hide many implementation details & free the user from having to specify explicitly the order in which translation takes place.

Form of a Syntax-Directed Definition : In a syntax-directed definition, each grammar production $A \rightarrow \alpha$ has associated with it a set of semantic rules of the form $b := f(c_1, c_2, \ldots\ldots, c_k)$ where f is a function, and either.

(i) b is a synthesized attribute of A and $c_1, c_2, \ldots\ldots, c_k$ are attributes belonging to the grammar symbols of the production, or

(ii) b is an inherited attribute of one of the grammar symbols on the right side of the production and $c_1, c_2, \ldots\ldots, c_k$ are attribut belonging to the grammar symbols of the production.

Q. 6. (b) Create 3-address code for following expression :

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

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

$t_1 := b - c$

$t_2 := a * t_1$

$t_3 := t_1 * d$

$t_2 := t_2 + t_3$

$t_2 := a + t_2$.

Q. 7. (a) What are different types of errors in compilation process. Explain a typical error detection and 'recovery mechanism.'

Ans. Error can occur at every stage of compiler.

→ Something missed while typing.

→ Some misspelling occurred.

→ Incorrect use of logical operators.

→ Violation of rules of language.

- → Type incompatibility of variables.
- → Incorrect algorithm may produce erroneous result.
- → Division by zero.
- → Exceeding the memory size of a variable.
- → Exceeding array limit.
- → Incorrect use of keywords.
- → Syntactic violation.
- → Function with improper name or return type.
- → Transposition of two adjacent characters or tokens.
- → Deletion of required character or token.

Syntax errors are detected by lexical or syntactic phase of compiler and all other are termed as semantic errors.

Syntax Errors :

1. int a, b, → extraneous comma
2. ca*b + c/d → missing parenthesis
3. Column in place of semicolumn

   A = 20 :

   B = 30 :
4. Misspelled keyword

   mains( ) in place of main ( )

   Flot in place of float

   While in place of while
5. Extra blank

Error detection & its reporting is important function of compiler. Errors can be found at almost all the phases of the compiler.

Lexical Phase : Misspelled token

Syntax Phase : Missing parenthesis or erroneous arithmetic expressions.

Intermediate Code : Incompatible operand types.

Code Optimizer : Certain statements are not reachable due to wrongly written function calls.

Code Generator : Some of the created words are too large to fit into registers.

Whenever of the phase detects error, it must call the error handler, which issues appropriate diagnostic message. If errors are not handled properly it may result into chaos and system may crash.

Error-Recovery Strategies :

(i) Panic mode

(ii) Phrase level

(iii) Error productions

(iv) Global correction

Panic-Mode Recovery : This is the simplest method to implement and can be used by most parsing methods. On discovering an error, the parser discards input symbols one at a time until one of a designated set of synchronizing tokens is found. The synchronizing tokens are usually delimiters, such as semicolon or end, whose role in the source program is clear.

Phrase-Level Recovery : On discovering an error, a parser may perform local correction on the remaining input; that is, it may replace a prefix of the remaining input by some string that allows the parser to continue.

Error Productions : If we have a good idea of the common errors that might be encountered, we can augment the grammar for the language at hand with productions that generate the erroneous constructs. We then use the grammar augmented by these error productions to construct a parser.

Global Correction : Ideally, we would like a compiler to make as few changes as possible in processing an incorrect input string.

Q. 7. (b) Explain various target for the intermediate code optimization.

Ans. Intermediate Code Optimization : After syntax and semantic analysis, some compilers generate an explicit intermediate representation of the source program. This intermediate representation should have two important properties; it should be easy to produce and easy to translate into the target program. We can represent an intermediate form called "three address code", which is likely the assembly language for a machine in which every memory location can act like a register. Three address code consists of a sequence of instructions, each of which has almost three operands. Code optimization is an optional phase designed to improve the intermediate code so that the ultimate object program runs faster &/or takes less space. In code optimization we generally try to remove the necessary variables.

Q. 8. (a) How registers are allocated while creating machine code?

Ans. Instructions involving register operands are usually shorter & faster than those involving operands in memory. Therefore efficient utilization of registers is particularly important in generating good code.

The use of registers is often subdivided into two subproblems.

(i) During register allocation, we select the set of variables that will reside in registers at a point in the program.

(ii) During a subsequent register assignment phase, we pick the specific register that a variable will reside in.

Global Register Allocation : The code generation algorithm used registers to hold values for the duration of a single basic block. However, all live variables were stored at the end of each block.

One strategy for global register allocation is to assign some fixed number of registers to hold the most active values in each inner loop.

Q. 8. (b) What are basic blocks and flow diagram, explain PEEPHOLE optimization technique.

Ans. Basic Blocks : Functions transfer control from one place (the callar) to another (the called function).

A basic block is a sequence of statements that enters at the start and ends with a branch at the end.

Remaining task of code generation is to create code for basic blocks and branch them together.

```
main( )
{
int a = 0; int b = 0;
{
```

```
            int b = 1;
            {
                  int a = 2;
                  printf("%d%d\n", a, b);
            }
            {
                  int b = 3;
                  printf ("%d%d\n", a, b);
            }
            printf ("%d%d\n", a, b);
      }
            printf ("%d%d\n", a, b);
      }
```

Partition Into Basic Blocks :

* Input : sequence of TAC instructions :

(i) Determine set of leaders, the Ist statement of each basic block :

(a) The Ist statement is a leader.

(b) Any statement that is the target of a conditional jump or goto is a leader.

(c) Any statement immediately following a conditional jump or goto is a leader.

(ii) For each leader, the basic block contains all statements upto the next leader.

The following sequence of three address statements forms a basic block :

$t_1 : a * a$

$t_2 : a*b$

$t_3 : 2 * t_2$

$t_4 : t_1 + t_3$

$t_5 : b * b$

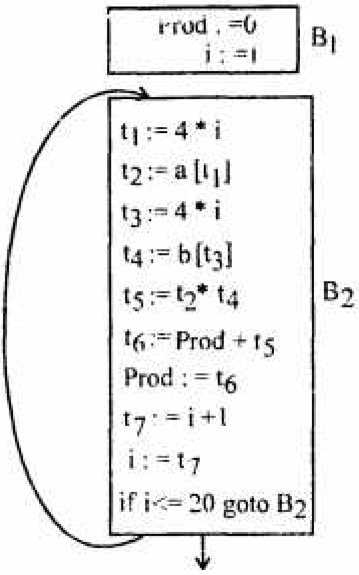
$t_6 \; t_4 + t_5$

Flow Graph : We can add the flow-to-control information to the set of the basic-blocks making up a program by constructing a directed graph called a flow graph. The modes of the flow graph are the basic blocks.

One node is distinguished as initial; it is the block whose leader is the first statement. There is a directed edge from block B_1 to block B_2 if B_2 can immediately follow B_1 in some execution sequence; that is, if

(i) There is a conditional or unconditional jump from the last statement of B_1 to the first statement of B_2 or

(ii) B_2 immediately follows B_1 in the order of the program and B_1 does not end in an unconditional jump.



Flow Graph

Peephole Optimization Technique : A simple but effective technique for locally improving the target code is peephole optimization, a method for trying to improve the performance of the target program by examining a short sequence of target instructions (called the peephole) & replacing these instructions by a shorter or faster sequence, whenever possible. Peephole technique can also be applied directly after intermediate code generation to improve the intermediate representation.

The peephole is a small, moving window on the target program. It is the characteristic of the peephole optimization that each improvement may spawn opportunities for additional improvements.