B.E.

Seventh Semester Examination, 2010-2011

Compiler Design (CSE-405-E)

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

Q. 1. (a) Differentiate between system software and an application software.

Ans. System Software and Application Software : System software is a computer software, that is designed to operate the computer hardware and to give and maintain a platform for running the application software. One of the most important and widely used system software are computer operating system. It is with the operating systems, that parts of a computer are able to work together. This system software performs tasks such as transferring data between memory and disks or rendering the output onto display device.

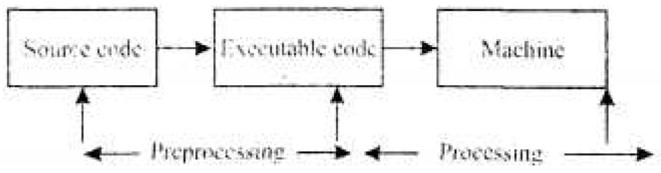
An application software is a computer software, which is designed to help the user to perform single or multiple related tasks. In other words, application software is actually a subclass of computer software, which employs the capabilities of a computer directly and thoroughly to a task, which the user wishes to perform. e.g., ERP software, media player, Tally etc.

Q. 1. (b) What is a translator ? Differentiate between compilation and interpretation.

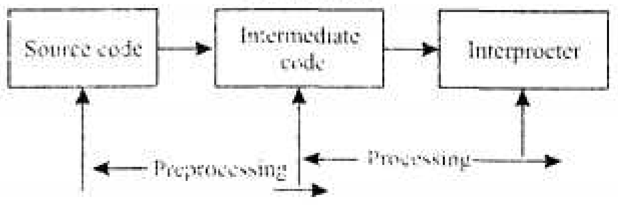
Ans. Translator : Compiler is a program that reads a program written in one language—source language and translate it into an equivalent program in another language (target language).

Basically translator translate one language code to other (as per requirement) language code.

Compiler Vs Interpreter : An interpreter translate some form of code into a target representation that it can immediately execute and evaluate. The structure of interpreter is similar to that of a compiler, but the amount of time it takes to produce the executable representation will vary as will the amount of optimization.



Compilation



Interpretation

Q. 1. (c) Explain how different phases of compilation will operates and converts following statement

position = initial + rate * 60

assuming data type of rate is float.

Ans. Three address code consists of a sequence of instructions, each of which has at most three operands

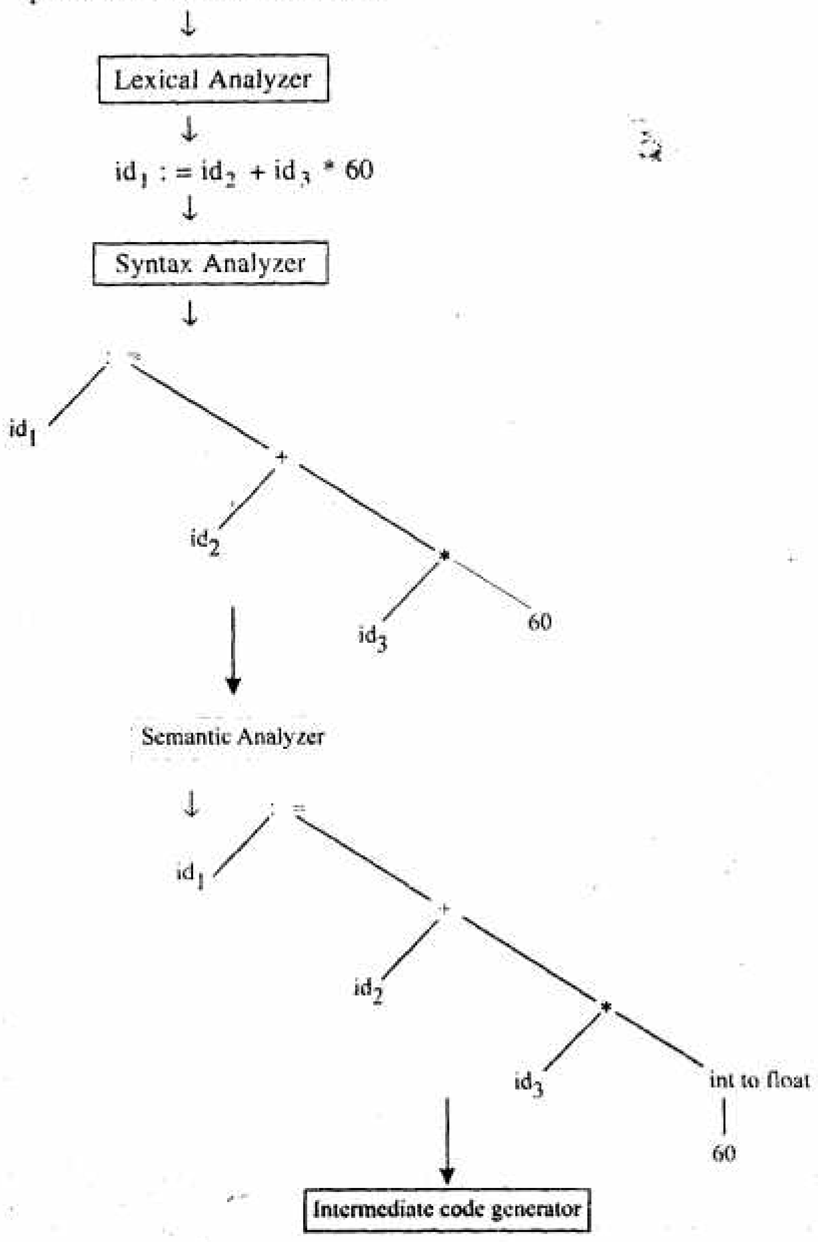
temp 1 : = int to float (60)

temp 2 : = id_3 * temp1

temp 3 : = id_2 & temp2

id 1 : = temp 3

position : = initial + rate * 60



↓

temp 1 : = int to float (60)
temp 2 : = id_3 * temp 1
temp 3 : = id_2 * temp 2
id_1 : = temp 3

↓

$\boxed{\text{code optimizer}}$

↓

temp 1 : = id_3 * 60.0
id_1 : = id_2 + temp1

↓

$\boxed{\text{code generater}}$

↓

```
MOV F        id3, R2
MUL F        ≠ 60.0, R2
MOV F        id2, R1
ADD F        R2, R1
MOV F        R1, id1
```

Symbol Table

1	position	....
2	initial	....
3	rate	....
4		

Q. 2. (a) Define finite state automata (DFA). Write an algorithm to simulate to DFA.

Ans. DFA (Deterministic Finite Automation) : A DFA consists of following three things :

(i) A finite set of states, one of which is designed as the initial state, called the starting state and some of which are designed as final states.

(ii) An input alphabet Σ.

(iii) A transition system (i.e., transition graph, transition table that tells for each state and each letter of input alphabet, the state to which to go next.

Mathematically a DFA is given $M = (Q, \Sigma, \delta, q_0, F)$.

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

(ii) Σ is a finite non-empty set of input symbols.

(iii) δ is a transition system and $\delta \in Q \times \Sigma \rightarrow Q$

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

(v) F is a set of accepting states (or final states) and $F \subseteq Q$.

Let $M = (Q, \Sigma, \delta\ q_0, F)$ be a NFA accept. We construct a DFA M′ that can also accept the same language L.

$$M' = (Q', \Sigma, \delta'\ q_0', F')$$

$Q' = 2^Q$ (power set of Q) and all states of Q' are denoted by $[q_1, q_2, q_3 \ldots q_n]$ where

$q_1, q_2 \ldots q_n \in Q$

Σ = input alphabet

$q_0' = [q_0]$

F' = set of all subsets of Q' containing an element of F

S' = transition system

Step 1 : Draw an initial vertex (state) with label $[q_0]$ of transition diagram S'.

Step 2 : Take this state $[q_0]$ and identify all next states from Q' on all different given input symbols.

Step 3 : Take any state of step 2 and find new next state from Q' on all different given input symbols.

Step 4 : Repeat step 3 until no new states are generated.

Step 5 : Draw all states as a final states of S' if it contain the final state of NFA.

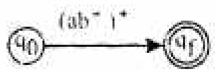
Q. 2. (b) Construct the transition diagram for the following regular expressions :

(i) (ab*)*

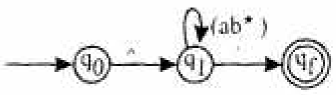
(ii) ((a | b)c*)*

(iii) (a | b)*abb

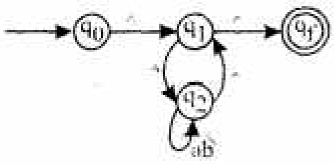
Ans. (i) Step 1 :



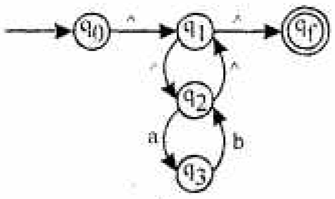
Step 2 :



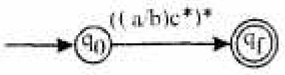
Step 3 :



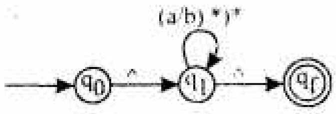
Step 4 :



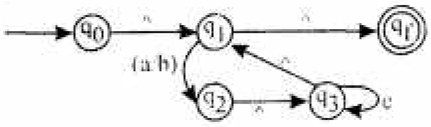
(ii) Step 1 :



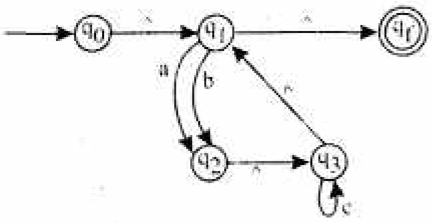
Step 2 :



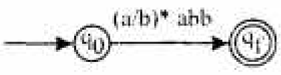
Step 3 :



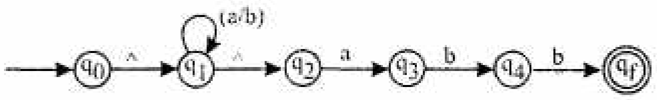
Step 4 :



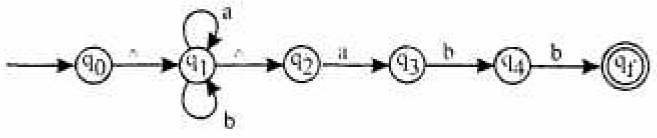
(iii) Step 1 :



Step 2 :



Step 3 :



Q. 3. (a) Check whether the following grammar is LL(1) grammar or not

S → iEtSS′ | a

S′ → eS | ε

E → b

Ans. S → iEtSS′ | a

S′ → eS | ε

E → b

Non-terminal	Input Symbol						
	a	b	e	i	t	$	
S	$S \to a$			$S \to iE$ tss'			
S'			$S' \to E$ $S' \to eS$			$S' \to \epsilon$	
E		$E \to b$					

Find first and follow, make the parsing table. If table contain multiple entries in a block, then grammar is not LL(1).

So above grammar is not LL(1).

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

Ans. Chomsky Classification of Grammars : There are 4 types of grammars :

(i) (Type 0) Unrestricted Grammar

(ii) (Type 1) Context Sensitive Grammar

(iii) (Type 2) Context Free Grammar

(iv) (Type 3) Regular Grammar

Type 0 : A grammar is called an unrestricted grammar if it contains a contracting production rules. It is also called recursive enumerable language.

Contracting Production Rule : A production rule, which is in the form of $w_{i+1} \to w_i$ where

$$|w_{i+1}| \to w_i$$

e.g., $bc \to b$

$$|bc| = 2 \qquad |b| = 1$$

Type 1 : A context sensitive grammar is a grammar in which each production contain a non-contracting production rules.

Non-contracting Production :

Production $\alpha \to \beta$

$$|\alpha| \le |\beta|$$

Example : $a = (\{S, A\}, \{a\}, P, S)$

$$S \to aA$$
$$aA \to aaA$$
$$aA \to aa$$

Context Free Grammar (Type 2) : In a production $\phi A \psi \to \phi w \psi$, string ϕ and ψ both are required to be empty. Thus, the possibility of replacing a non-terminal latter in a sequential form is independent of adjacent symbols.

If grammar contain only one non-terminal symbol in left hand side, Type 2 grammar.

Example : $S \to A$

$$A \to qaA$$

$A \to a$

Regular Grammar (Type 3) : If grammar contain production either in right linear form or left linear form.

Right Linear Form :

$A \to aB$ {terminal variable

$A \to a$

Left Linear Form :

$A \to Ba$ {variable terminal

$A \to a$

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

S → a, S → bB, BA → bA, CA → A

Ans.

$S \to a$	type 3	so Type 1 Grammar
$S \to bB$	type 3	
$BA \to bA$	type 1	
$CA \to A$	type 1	

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 → E + E

E → E * E

E → (E)

E → id

Check whether the grammar is ambiguous for above statement.

Ans. w = id + (id) + id * id

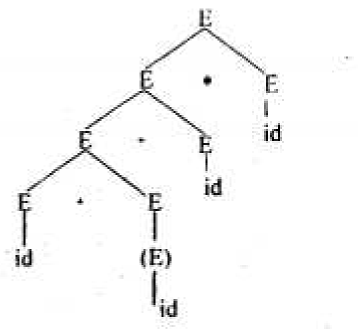
E → E + E

E → E * E

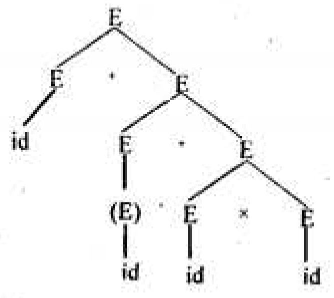
E → (E)

E → id

Left Derivation Tree



Right Derivation Tree



If any grammar has more than one parse tree then, grammar is ambiguous grammar so above grammar is ambiguous.

Q. 4. (b) Explain the problem of left factoring and left recursion. How these problems are removed.

Ans. Left Factoring & Left Recursion :

Left Factoring : Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive parsing.

Ex. : $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$

A contain two production, we cannot identified from first letter of given production, for this confusion we need left factoring

$$A \rightarrow \alpha A'$$
$$A' \rightarrow \beta_1 \mid \beta_2$$

Left Recursion : It is possible for a recursive descent parser to loop forever. A problem arises with left recursive productions like

$$\text{expr} \rightarrow \text{expr} + \text{term}$$

in which the leftmost symbol on right side is the same as non-terminal on the left side of production

$A = \text{expr},\quad \delta = +\text{term},\quad \beta = \text{term}$

so $\quad A \rightarrow A\,\alpha \mid \beta$

Left Recursion is recognized by same symbol is present both side (left and right side).

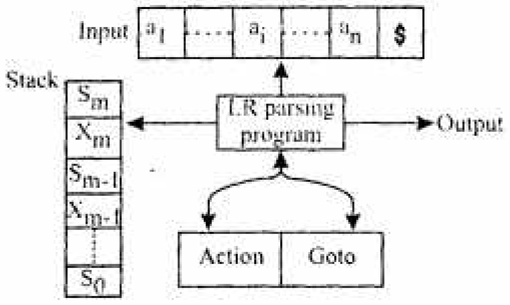
For elimination of left recursion is

$$A \rightarrow \beta R$$
$$R \rightarrow \alpha R \mid \epsilon$$

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

Ans. The schematic form of an LR parser is shown in fig. given following. It consists of an input, an output, a stack, a driver program, and a parsing table that has two parts (action and goto).

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 $S_0\, X_1\, S_1\, X_2\, S_2\, \ldots .. X_m\, S_m . S_m$ is top of stack. Each X_i is a grammar symbol and each S_i is a symbol called a state.



Model of an LR Parser

The parsing table consists of two parts, a parsing action function **action** and goto function **goto.** The program driving the LR parser behaves as follows. It determines S_m, the state currently on top of stack and a_i the current input symbol. It then consults action $[S_m, a_i]$, the parsing action table entry for state S_m and input a_i, which can have one of 4 values :

(i) shif s , where s is a state

(ii) reduce by a grammar $A \to \beta$

(iii) accept and

(iv) error.

LR Parsing 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 until an accept or error action is encountered set i_p to point to the first symbol of w $;

```
repeat forever begin
  let s be the state on top of stack and
    a the symbol pointed to by $i_p$;
if action [s, a] = shift s′ then begin
     push a then s′ on top of stack;
  advance $i_p$ to the next input symbol
end
else if action [s, a] = reduce A → β then begin
     pop 2 * |β| symbols off the stack;
     let s′ be the state now on top of stack;
  push A then goto (s′, A] on top of stack;
  output the production A → β
end
else if action [s, a] = accept then
     return
     else error( )
end
```

Q. 5. (b) Explain how LALR parsing table is constructed. Explain with example, reduce-reduce conflict.

Ans. Constructing LALR Parsing Table :

LALR → Lookahead LR parser

Input : An augmented grammar G′.

Output : The LALR parsing table functions action and goto for G′.

Method :

(i) Construct C = $\{I_0, I_1, ... I_n\}$, the collection of sets of LR(1) items.

(ii) For each core present among the set of LR(1) items find all sets having that core, and replace these sets by their union.

(iii) Let $C' = \{J_0, J_1, ...J_m\}$ be the resulting sets of LR(1) items. The parsing actions for state i are constructed from J_i in the same manner, if there is a parsing action conflict, the algorithm fails to produce a parser and grammar is said not to be LALR(1).

(iv) The goto table is constructed as follows if J is the union of one or more sets of LR(1) items; that is $J = I_1 \cup I_2 \cup .... \cup I_K$, then cores of goto (I_1, X), goto (I_2, X),... goto (I_k, X) are the same, since $I_1, ...I_k$ all have same core. Let K be union of all sets of items having the same core as goto (I_1, X) then goto $(J, X) = K$.

Q. 6. (a) What is syntax directed translation schemes, explain how they are used to make syntax Trees.

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

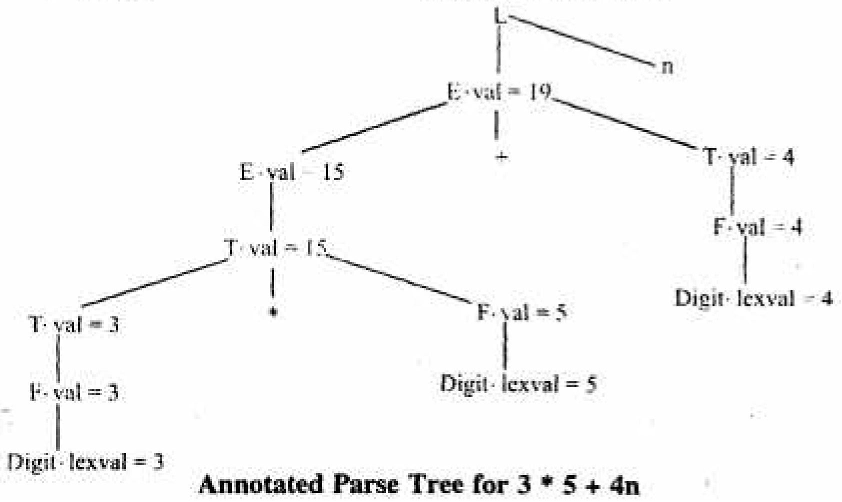
In a syntax directed definition each grammar production $A \to \alpha$ has associated with it set of semantic rules of the form $b := f(c_1, c_2 ... c_k)$ where f is a function and either.

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

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

Example.

Production	Semantic Rules
$L \to E_n$	print (E.val)
$E \to E_1 + T$	$E.val := E_1.val = T.val$
$E \to T$	E.val := T.val
$T \to T_1 * F$	$T.val := T_1.val * F.val$
$T \to F$	T.val = F.val
$F \to (E)$	F.val = F.val
$F \to$ digit	F.val := digit.lexval



Annotated Parse Tree for 3 * 5 + 4n

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

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

Ans. Three Address Code :

Quadruples :

a + a * (b–c) + (b–c) * d

op	arg1	arg2	result
–	b	c	t_1
*	t_1	d	t_2
*	t_1	a	t_3
+	t_2	t_3	t_4
+	t_4	a	t_5

Triple :

op	arg1	arg2
–	b	c
*	(0)	d
*	(0)	a
+	(1)	(2)
+	(3)	a

Q. 7. (a) What are the typical entries in a symbol table, what are various data structures used to implement the table.

Ans. Symbol Table : An essential function of a compiler is to record the identifiers used in the source program and collect information about various attributes of each identifier. These attributes may provide information about the storage allocated for an identifier, its type, its scope (where in the program it is valid) and in case of procedure names, such things as number and types of its arguments, the method of passing each argument and type returned if any.

A symbol table is a data structure containing a record for each identifier, with fields for attributes of identifier. The data structure allows us to find the record for each identifier quickly and to store or retrieve data from that record quickly. When an identifier in the source program is detected by the lexical analyzer, the identifier is entered into symbol table.

Example, Pascal declaration

var position, initial, rate : real;

The type real is not known when position, initial and rate are seen by the lexical analyzer. The remaining phases enter information about identifiers into symbol table and then use this information in various ways.

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

Ans. Various Target for Code Optimization : When we optimize the code, must see the following property should preserve :

(i) A transformation must preserve the meaning of programs. That is, an "optimization" must not change the output produced by a program for a given input, or cause an error, such as a division by zero, that was not present in original version of source program.

(ii) A transformation must, on the average, speed up programs by a measurable amount.

(iii) A transformation must be worth the effort.

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

Ans. Register Allocation : Instruction involving only register operands are shorter and faster than those involving memory operands, efficient utilization of register is important in generating good code. There are various strategies for deciding what values in a program should reside in registers (register allocation) and in which register each value should reside (register assignment).

One approach to register allocation and assignment is to assign specific values in an object program to certain registers. This approach has the advantage that is simplifies the design of a camplier. Its disadvantage is that applied too strictly, it uses registers inefficiently certain registers may go unused over substantial portions of code, while unnecessary loads and stores are generated.

Nevertheless, it is reasonable in most computing environments to reserve a few registers for base registers, stack pointers and like and to allow the remaining registers to be used by compiler as it sees fit.

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. To save some of these stores and corresponding loads, we might arrange to assign registers to frequently used variables and keep these registers consistent across block boundaries (globally).

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

Ans. Basic Blocks : Basic blocks can be represented by a variety data structures. Each basic block can be represented by a record consisting of a count of the no. of three address code in block, followed by a pointer to the leader (first three address code) of the block and by the lists of predecessors and successors of block.

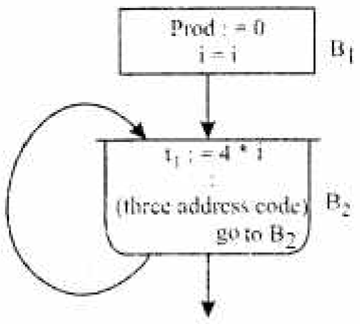
Flow Graph : We can add the flow of control information to set of basic blocks making up a program by constructing a directed graph called a flow graph. The nodes of flow graph are basic blocks. One node is initial; it is block whose leader is first statement.

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, in the order of program, and B_1 does not end in an unconditional jump.

B_1 is a predecessor of B_2
B_2 is a successor of B_1



Flow graph

B_1 & B_2 are basic blocks.

Peephole Optimization : A statement by statement code generation strategy often process target code that contains redundant instructions and suboptimal construct. The quality of such target code can be improved by applying "optimizing" transformation to the target program. The term optimizing is somewhat misleading because there is no guarantee that the resulting code is optimal under any mathematical measure.

A simple but effective technique for locally improving the target code is peephole technique, a method for trying to improve the performance of target program by examining a short sequence of target instructions (called peephole) and replacing these instructions by a shorter or faster sequence, whenever possible.

The peephole is a small, moving window on the target program. The code in peephole need not be contiguous, although some implementations do require this peephole characteristic :

(i) redundant instruction elimination

(ii) flow of control optimization

(iii) algebraic simplification

(iv) use of machine idioms.