SOLVED QUESTION BANK

[Sequence given as per syllabus]

INTRODUCTION

This unit develops the theme of the translation of languages guided by context free grammars we associate information with a programming language construct by attaching attributes to the grammar. Symbols representing the construct values for attributes are computed by semantic rules associated with the grammar productions.

There are notational framework for intermediate code generation that is an extension of context-free grammars. This framework is called as syntax-directed translation scheme. It allows subroutines or 'semantic actions' to be attached to the productions of context free grammar. These subroutines generate intermediate code when called at appropriate times by a parser for that grammar.

We will see what are inherited, synthesized attribute, evaluation of attributes, L and S attributed definition, implementation of SDTS, evaluation of expression using semantic actions and finally the postfix, syntax tree and three address code used for intermediate code representation.

SPECIFICATION OF TRANSLATIONS

Q.1. Write short note on specification of translation.

Ans. Specification translation :

- It involves specifying what the construct is, as well as specifying the translating rules for construct.
- Whenever a compiler encounters that construct in a program, it will translate the construct according to the rules of translation.
- Translation does not necessarily mean generating either intermediate code or object code.
- Translation also involves adding information into the symbol table as well as performing construct specific computations.
- For example, if a construct is a declarative about the constructs type attribute into symbol table.
- If construct is an expression then its translation generates the code for evaluating the expression.
- Translation of a construct involves manipulating the values of various quantities.

When translating the declarative statement int a, b, c, the compiler need to extract the type int and add it to the symbol table records of a, b, c.

- This require compiler to keep the track of the type int as well as the pointers to the symbol records containing a, b and c.
- Syntax directed definitions use CFG to specify the syntactic structure of the construct.
- It associates a set of attributes with each grammar symbol and with each production, it associates a set of semantic rules for computing the values of the attributes of the grammar symbols appearing in that production.
- Therefore the grammar and the set of semantic rules constitute syntax directed definitions.

SYNTAX DIRECTED DEFINITION

Q.2. What is syntax-directed definitions?

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.
- If we think of a node for the grammar symbol in a parse tree as a record with fields for holding information, then an attribute corresponds to the name of a field.

Q.3. Explain form of a Syntax-directed definition.

Ans.

- 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.

(1) 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

(2) b is a 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 attributes belonging to the grammar symbol of the production.

- An attribute grammar is a syntax-directed definition in which the function in semantic rules cannot have side effects.

As per Indian Copyright Act 1957, photocopying of a book is punishable under law. VBD

Q.4. Explain semantic action. **CS : S-10(2M), W-10(1M)**

Ans. Semantic action :

- A compiler has to do more than just recognize if a sequence of characters forms a valid sentence in the language.
- The semantic action of a parser perform useful operation.
  (1) Build an abstract parse tree.
  (2) Type checking
  (3) Evaluation in the case of an interpreter.
  (4) Begin the translation process in the case of a compiler.
- In recursive descent parser the semantic action are mixed in with the control flow of parsing.
- In some compiler constructors such as Java CC and Yacc the semantic actions are attached to the production rules.
- In other compiler constructions sable CC for example the syntax tree automatically generated.
- Each symbol, terminal or non terminal, may have its own type of semantic value.

  exp → INT

  exp → exp PLUS exp.

  exp → exp MINUS exp.

  exp → exp MUL exp.

  exp → exp DIV exp.

  exp → MINUS exp.

SYNTAX DIRECTED TRANSLATION SCHEME

Q.5. What is Syntax directed translation scheme? Justify the necessity for intermediate code and give some example.

Ans. Syntax directed translation scheme :

- A syntax directed translation scheme is a notational framework for intermediate code generation which is an extension of context-free grammars.
- It allows subroutines or "Semantic action" to be attached to the productions of a context-free grammar.
- These subroutines generate intermediate code when called at appropriate times by a parser for that grammar.
- Enable the compiler designer to express the generation of code directly in terms of the syntactic of the source language.

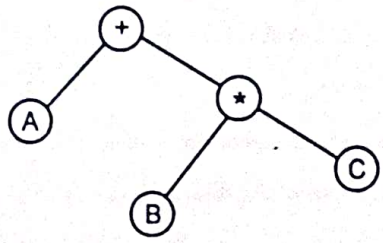
Necessity for the intermediate code. :

- Syntax directed translation scheme can be used for translation of source code to target code, which generally speeds up the compiler, the optimization of machine or assembly code is harder. Hence some form of intermediate code is generated on which optimization can be performed easily.
- The reason why efficient machine or assembly code is hard to generate is that one is immediately forced to choose a particular register to hold result difficult. Usually the intermediate code is choosen in such a way like assembly language each statement involves at most one operation, but unlike assembly code, the register in which each operation occurs is left unspecified.

Example of intermediate code :

(1) **Postfix notation :** abc* - is a postfix notation for a + b*c.

(2) **Syntax tree :** For a + b*c syntax tree is shown below.



(3) **Three address code :** The three address code for a + b*c will be

T_1 : b * c

T_2 : a + T_1 , where T1 and T2 are compiler generated temporaries.

INHERITED AND SYNTHESIZED ATTRIBUTES

Q.6. What are the implementations of the translations specified by syntax-directed definitions?

OR What are synthesized and inherited attributes?

OR Define synthesized and inherited attribute. **CS : W-10(2M)**

Ans.

- Attributes are associated with the grammar symbols that are the labels of the parse tree node.
- When a semantic rule is evaluated, the parser computes the value of an attribute at a parse tree node.
- To evaluate the semantic rules and carry out translation, the parse tree must be traversed and the values of the attributes at the nodes computed.
- The order in which we traverse the parse tree nodes depends on the dependencies of the attributes at the parse tree node.

- Carrying out the translation specified by the syntax-directed definition involves :

(1) Generating the parse tree for the input string w.

(2) Finding out the traversal order of the parse tree nodes by generating a dependency graph and doing a topological sort of that graph.

(3) Traversing the parse tree in the proper order and getting the semantic rules evaluated.

Synthesized and inherited attributes :

- The attributes associated with a grammar symbols are classified into two types :

(A) Synthesized attributes.

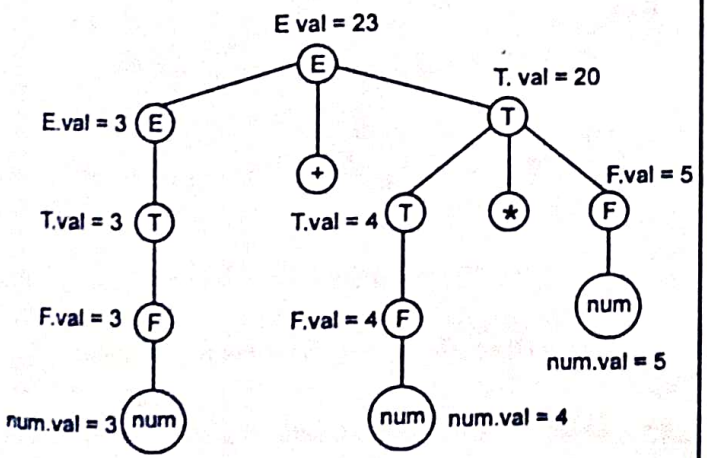
(B) Inherited attributes.

(A) Synthesized attributes :

- If value of attribute at parse tree node is determined from attribute values of the children node, then attribute is defined as synthesized attribute.
- They are evaluated during a single bottom up traversal of parse tree and are as follows :

$E \rightarrow E_1 + T$	E. val : = E_1 . val + T. val
$E \rightarrow T$	E. val : = T. val
$E \rightarrow T_1 * F$	T. val : = T_1 . val * F. val
$T \rightarrow F$	T. val : = F. val
$F \rightarrow id$	F. val : = num. lexval

- The above shown SDTS using synthesized attribute specify the translations, that are needed to be carried by expression evaluator.
- For example, parse tree for an expression 3 + 4 * 5 along with values of attribute at the nodes of parse tree is shown below.



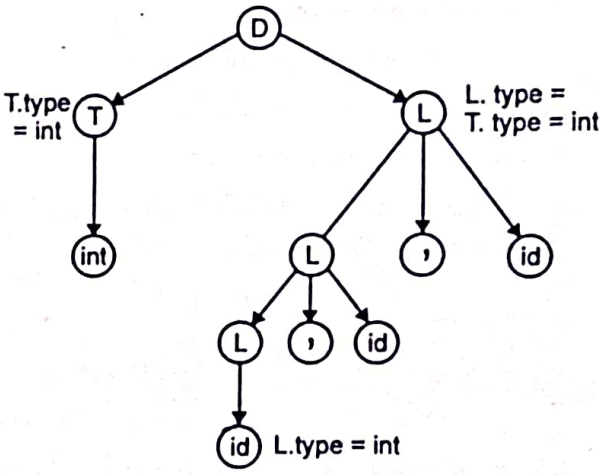
- Syntax directed definitions that use synthesized attributes only are known as "S-attributed" definitions.

(B) Inherited attributes :

- They are those initial values at a node in the parse tree which are defined in terms of attributes of the parent and / or siblings of that node.
- For example,

$D \rightarrow TL$	L. type = T. type
$T \rightarrow$ int	T. type = int
$T \rightarrow$ real	T. type = real
$L \rightarrow L_1$, id	L_1 . type = L. type enter (id . prt, L . type)
$L \rightarrow$ id	enter (id . prt, L . type)

- A parse tree for an input string int id_1, id_2, id_3 along with values of the attributes at the nodes of parse tree is as shown below.



- Inherited attributes are convenient for expressing the dependency of a programming language construct on the context in which it appears when inherited attributes are used, then the inter dependencies among the attributes at the nodes of the parse tree must be taken into account when evaluating their semantic rules.
- The interdependencies among attributes are depicted by a directed graph called a "dependency graph".
- If the semantic rule is in the form of a procedure call fun $(a_1, a_2, a_3, ..., a_k)$, then it can be transformed into form b = fun $(a_1, a_2, a_3, ..., a_k)$, where b is a dummy synthesized attribute.

DEPENDENCY GRAPHS

Q.7. Write short note on dependency graphs.

Ans. Dependency graph :

- If an attribute b at a node in a parse tree depends on an attribute c, then the semantic rule for b at that node must be evaluated after the semantic rule that defines c.
- The interdependencies among the inherited and synthesized attributes at the nodes in a parse tree can be depicted by a directed graph, called a dependency graph.
- Before constructing a dependency graph for a parse tree, we put each semantic rule into the form $b := f(c_1, c_2, \ldots, c_k)$ by introducing a dummy synthesized attribute b for each semantic rule that consists of a procedure call.
- The graph has a node for each attribute and an edge to the node for b from the node for c if attribute b depends on attribute c.
- The algorithm for construction of dependency graph for a given parse tree is as follows :

  for each node n in the parse tree do

  for each attribute a of the grammar symbol at n do

  construct a node in the dependency graph for a :

  for each node n in the parse tree do

  for each semantic rule $b := f(c_1, c_2, \ldots, c_k)$ associated with the production used at n do

  for $i = 1$ to k do

  construct an edge from the node for c_i to the node for b :
- For example, fig. (c) shows the dependency graph for the parse tree in fig. (b), nodes in the dependency graphs are marked by numbers.
- The parse tree in fig (b) is constructed from fig (a).

Production	Semantic rules
$D \rightarrow TL$	L. in := T. type
$T \rightarrow$ int	T. type := integer
$T \rightarrow$ real	T. type := real
$L \rightarrow L_1$, id	L_1. in := L. in addtype (id. Entry, L. in)
$L \rightarrow$ id	addtype (id. Entry, L. in)

Fig. (a) Syntax-directed definition with inherited attribute L. in.

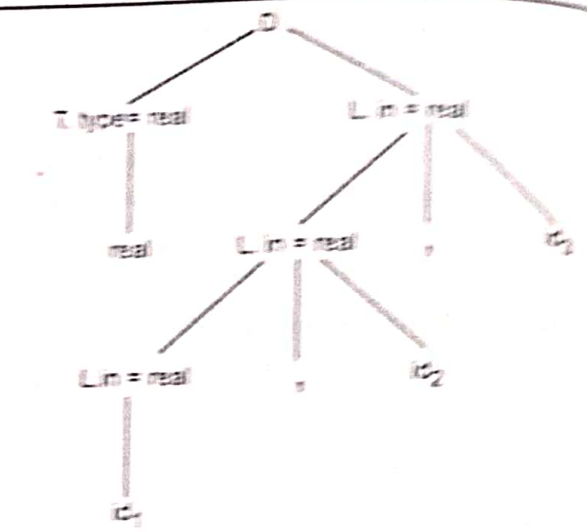


Fig. (b) Parse tree with inherited attribute in at each node labelled L.

- In fig. (c), there is an edge to node 5 for L. in from node 4 for T. type because the inherited attribute L. in depends on the attribute T. type according to the semantic rule L. in := T. type for the production D → TL.
- The two downward edges into nodes 7 and 9 arise, because L. in depends on L. in according to the semantic rule L_1. in := L. in for the production $L \rightarrow L_1$, id. Each of the semantic rules addtype (id. entry, L. in) associated with the L-production leads to the creation of a dummy attribute.
- Nodes 6, 8 and 10 are constructed for these dummy attributes.

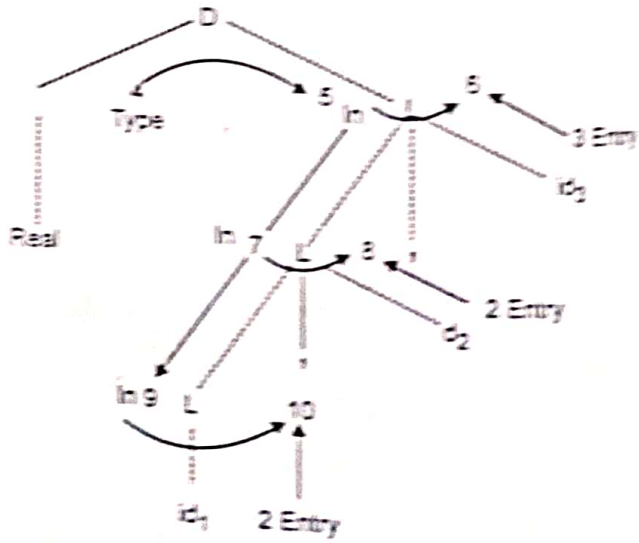


Fig. (c) Dependency graph for parse tree of fig. (b)

Q.8. Explain the term 'topological sort'.

Ans. Topological sort :

- A topological sort of a directed cyclic graph is any ordering $m_1, m_2, \ldots, m_k$ of the nodes of the graph such that edges go from nodes earlier in the ordering to later nodes; that is, if $m_i \rightarrow m_j$ is an edge from m_i to m_j, then m_i appears before m_j in the ordering.
- Any topological sort of a dependency graph gives a valid order in which the semantic rules associated with the nodes in a parse tree can be evaluated.
- That is, in the topological sort, the dependent attributes $c_1, c_2, \ldots, c_k$ in a semantic rule $b := f(c_1, c_2, \ldots, c_k)$ are available at a node before f is evaluated.
- For example, each of the edges in the dependency graph in figure goes from a lower-numbered node to a higher-numbered node.
- Hence a topological sort of the dependency graph is obtained by writing down the nodes in the order of their numbers.
- From this topological sort, we obtain the following program. We write a_n for the attribute associated with the node numbered n in the dependency graph.

$a_4 :=$ real;

$a_5 := a_4$;

addtype (id_3.entry, a_5);

$a_7 := a_5$

$a_9 := a_7$;

addtype (id_1. Entry, a_9);

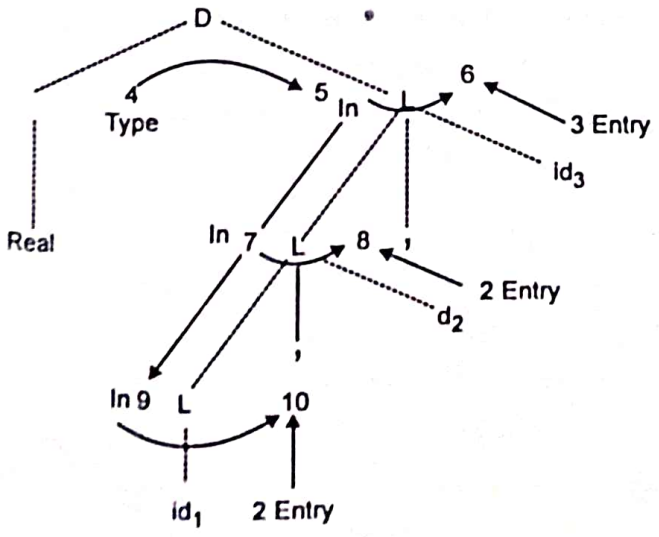


Fig. (d) Dependency graph

EVALUATION ORDER

Q.9. Write a short note on evaluation order.

Ans. Evaluation order :

- A topological sort of a directed acycle graph is any ordering $m_1, m_2, \ldots, m_k$ of the nodes of the graph such that edges go from nodes earlier in the ordering to later nodes.
- If $m_i \rightarrow m_j$ is an edge from m_i to m_j, then m_i appears before m_j in the ordering.
- Any topological sort of a dependency graph gives a valid order in which the semantic rules associated with the nodes in a parse tree can be evaluated.
- That is, in the topological sort, the dependent attributes $c_1, c_2, \ldots, c_k$ in a semantic rule $b := f(c_1, c_2, \ldots, c_k)$ are available at a node before f is evaluated.
- From a topological sort of the dependency graph we obtain an evaluation order for the semantic rules.
- Evaluation of the semantic rules in this order yield the translation of the input string.
- The several methods for evaluating semantic rules are as follows :

(1) Parse-tree method :

- At compile time, these method obtain an evaluation order from a topological sort of the dependency graph constructed from the parse tree for each input.
- These methods will fail to find an evaluation order only if the dependency graph for the particular parse tree under consideration has a cycle.

(2) Rule-based method :

- At compiler-construction time, the semantic rules associated with productions are analyzed, either by hand or by a specialized tool.
- For each production, the order in which the attributes associated with that production are evaluated is predetermined at compiler construction time.

(3) Oblivious method :

- An evaluation order is chosen without considering the semantic rule.
- For example, if translation takes place during parsing than the order of evaluation is forced by the parsing method, independent of the semantic rules.
- An oblivious evaluation order restricts the class of syntax-directed definitions that can be implemented.

- Rule based an oblivious methods need not explicitly construct the dependency graph at compile time, so they can be more efficient in their use of compile time and space.
- A syntax-directed definition is said to be circular if the dependency graph for some parse tree generated by its grammar as a cycle.

TOP DOWN AND BOTTOM UP EVALUATION

Q.10. What is top down and bottom up evaluation of attributes?

Ans.

- A syntax analyzer is a program that perform syntax analysis.
- A parser obtain a string of tokens from lexical analyzer and verifies whether or not string is a valid construct of the source language i.e. whether or not it can be generated by the grammar for the source language.
- The parser either attempts to derive the string of tokens w from the start symbol S or it attempts to reduce w to the start symbol of grammar by tracing the derivations of w in reverse.
- An attempt to derive w from the grammar's start symbol S is equivalent to an attempt to construct the top-down parse tree i.e. it starts from the root node and proceeds towards the leaves.
- Similarly an attempt to reduce w to the grammar's start symbol S is equivalent to an attempt to construct bottom up tree.

Alphabet : An alphabet is a finite set of symbols denoted by symbol Σ.

Language :

- A language is a set of strings formed by using the symbols belonging to some previously chosen alphabet.
- For example, if $\Sigma = \{0, 1\}$ then one of the languages that can be defined over this Σ will be L : { $\in$, 0, 00, 000, 1, 11, 111,...}.

TOP DOWN PARSING

Q.11. What is top down parsing?

Ans. Top down parsing :

- Top down parsing attempts to find the left most derivation for an input string w, which is equivalent to constructing a parse tree for the input string w that starts from the root and creates the nodes of the parse tree in predefined order.
- The reason that top-down parsing seeks the left most derivations for an input string w and not the right most derivations is that input string w is scanned by the parser from left to right.
- Since top down parsing attempts to find the leftmost derivations for an input string w, a top down parser may require backtracking because in the attempt to obtain the left most derivation of input string w, a parser may encounter a situation in which a non-terminal A is required to be derived next and there are multiple A productions, such as $A \rightarrow \alpha_1 | \alpha_2 | .... | \alpha_n$.
- In such a situation deciding which A production to use for derivation of A is a problem.
- So, the parser will select one of the A-production to derive A and if this derivation finally leads to the derivation of w, then parser announces the successful completion of parsing.
- For example, consider top-down parser for following grammar

  $S \rightarrow aAb$

  $A \rightarrow cd / c$
- Let input string be w = acb the parser initially creates a tree consisting of a single node, labeled S and input pointer points to a, the first symbol of input string w.
- The parser then user the S-production $S \rightarrow aAb$ to expand tree shown in Fig. (a).

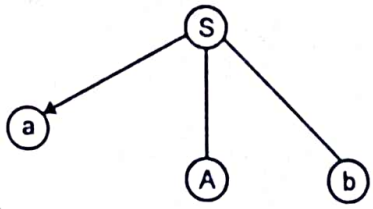


Fig. (a)

- The leftmost leaf labeled a, matches the first input symbol of w. Hence parser will now advance the input pointer to c, the second symbol of string w and consider the next leaf labeled A.
- It will then expand A using first alternative for A in order to obtain the tree as shown in Fig. (b)

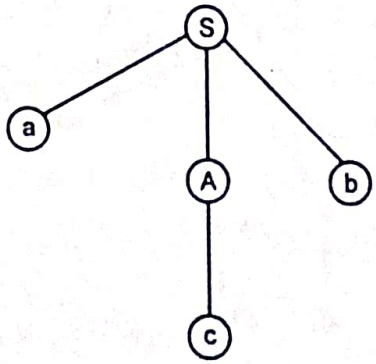


Fig.(b)

- The parser now has the match for the second input symbol, so it advances the pointer to b, the third symbol of w and compare it to the label of the next leaf
- If the label does not match d, it reports failure and goes back (backtracks) to A as shown in Fig. (c)

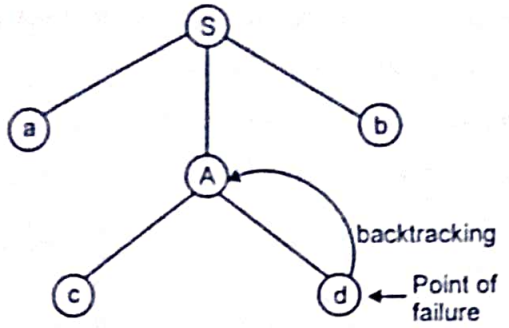


Fig.(c)

- The parser now reset the input pointer to the second input symbol - the position it had when the parser encountered A and it will try a second alternative for A in order to obtain the tree.
- If the leaf c matches the second symbol, if next leaf b matches the third symbol of w, then the parser will halt and announce the successful completion of parsing.

BOTTOM UP PARSING

Q.12. What is bottom up parsing?

Ans. Bottom up parsing :

- This parsing can be defined as an attempt to reduce the input string w to the start symbol of a grammar by tracing out the rightmost derivations of w in reverse.
- This is equivalent to constructing a parse tree for the input string w by starting with leaves and proceeding toward the root i.e. attempting to construct the parse tree from bottom up.
- This involves searching for the substring that matches the right side of any of the productions of the grammar.
- This substring is replaced by the lefthand side nonterminal of the production if this replacement leads to the generation of the sentential form that comes one step before in the right most derivation.
- This process of replacing the right side of the production by the left side nonterminal is called "reduction".
- Hence, reduction is nothing more than performing derivations in reverse.
- The reason why bottom up parsing tries to trace out the right most derivations of an input string w in reverse and not the left most derivation is because the parse scans the input string w from left to right, one symbol per token at a time.
- And to trace out right most derivations of an input string w in reverse, the token of w must be made available in a left to right order.
- For example, if the rightmost derivation sequence of some w is

  $S \to \alpha_1 \to \alpha_2 \to \alpha_3 \to \ldots\ldots \to \alpha_{n-1} \to w$

  then bottom up parse starts with w and searches for the occurrence of a substring of w that matches the right side of some production $A \to \beta$ such that the replacement of β by A will lead to the generation of α_{n-1}.
- The parser replaces β by A, then it searches for the occurrence of a substring at α_{n-1} that matches the right side of some productions $\beta \to r$ such that replacement of r by β will lead to the generation of α_{n-2}.
- This process continues until the entire w substring is reduced to S, or until the parser encounters an error.
- Bottom up parsing involves the selection of a substring that matches the right side of the production.
- These reduction to the nonterminal on the left side of the production represents one step along the reverse of a right most derivation.
- It leads to the generation of the previous right most derivation.
- This means that selecting a substring that matches the right side of production is not enough ; the positions of this substring in the sentential form is also important.

A handle of a right sentential form :

(1) A handle of a right sentential form r is a production $A \to \beta$ and a position of β in r.

(2) The string β will be found and replaced by A to produce the priors right sentential form in the right most derivation of r.

(3) If $S \rightarrow \alpha A\beta \rightarrow \alpha r \beta$, then $A \rightarrow r$ is a handle of $\alpha r \beta$, in the position following α.

(4) For example, consider the grammar :

$E \rightarrow E + E / E * E / id$

and the right most derivation :

$E \rightarrow E + E \rightarrow E + E * E \rightarrow E$

$+ E * id \rightarrow E + id * id \rightarrow id + id * id$

The handles of the sentential forms occurring in the above derivations are shown in table below.

Sentential form	Handle
id + id * id	E → id at position preceding +.
E + id * id	E → id at position following +
E → E * id	E → id at position following *
E + E * id	E → E * E at position following +
E + E	E → E + E at position preceding the end marker.

- So, the bottom up parsing is an attempt to detect the handle of a right sentential form.
- This is equivalent to performing right most derivation in reverse and is called "handle pruning".
- So if the right most derivation sequence of some w is $S \rightarrow \alpha_1 \rightarrow \alpha_2 \rightarrow \alpha_3 \rightarrow ..... \rightarrow \alpha_{n-1} \rightarrow w$, then handle pruning starts with w, the n^{th} right sentential form, the handle $A_n \rightarrow B_n$ of w is located and β_n is replaced by the left side of production $A_n \rightarrow B_n$ in order to obtain α_{n-1}.

L AND S ATTRIBUTE DEFINITIONS

Q.13. Explain the term S-attributed definition and L-attributed definition.

OR Explain L-attributed definitions S-attributed definition.

CS : S-10(2M)

Ans. S-attributed definition :

- A syntax-directed definition that uses synthesized attributes exclusively is said to be an S-attributed definition. A parse tree showing the values of attributes at each node is called an annotated parse tree.
- Synthesized attributes can be evaluated by a bottom-up parser as the input is being parsed.
- The parser can keep the values of the synthesized attributes associated with the grammar symbols on its stack.
- Whenever, a reduction is made, the values of the new synthesized attributes are computed from the attributes appearing on the stack for the grammar symbols on the right side of the reducing production.
- A translator for an S-attributed definition can often be implemented with the help of an LR-parser generator.
- From an S-attributed definition, the parser generator can construct a translator that evaluates attributes as it parses the input.
- A bottom-up parser uses a stack to hold information about sub-trees that have been parsed. We can use extra fields in the parser stack to hold the values of synthesized attributes.

L - attributed definition :

- An inherited attribute is one whose value at a node in a parser tree is defined in terms of attributes at the parent and / or siblings of that node.
- Inherited attributes are convenient for expressing the dependence of a programming language, constructed on the context in which it appears.
- A syntax-directed definition is L-attributed if each inherited attribute of X_j, $1 \le j \le n$, on the right side of $A \rightarrow X_1, X_2, ...X_3$ depends only on

(1) The attributes of the symbols $X_1, X, ...X_{1-1}$ to the left of X_1 in the production, and

(2) The inherited attributes of A.

$D \rightarrow TL$	L . Type = T . Type
$T \rightarrow int$	T . type = int
$T \rightarrow real$	T . type = real
$L \rightarrow L_1$. id	L_1 . type = L . type ; enter (id . prt, L . type) ;
$L \rightarrow id$	enter (id . ptr, L . type)

- The syntax directed definition given above is an example of the L – attributed definition, because the inherited attribute L . type depends on T. type, and T is to the left of L in the production $D \rightarrow TL$.
- Similarly the inherited attribute L_1 . type depends on the inherited attribute L. type, and L is parent of L_1 in the production $L \rightarrow L_1$, id .
- Note that every S-attributed definition is L-attributed.

IMPLEMENTATION OF SDTS

Q.14. Write short note on implementation of SDTS.

Ans. Implementation of SDTS :

- A syntax-directed translation scheme is a convenient description of what we would like done.
- The output defined is independent of the kind of parser used to construct the parse tree or the kind of mechanism used to compute the translations.
- Thus, a syntax-directed translation scheme provides a method for describing an input-output mapping and that description is independent of any implementation.
- Another convenience of the approach is that it is easy to modify.
- New productions and semantic actions can often be added without disturbing the existing translations being computed.
- Having written a syntax-directed translation scheme, our next task is to convert it into a program that implements the input-output mapping described.

Mechanism to implement syntax directed translation :

- A useful requirement is that we have a bottom-up parser for the grammar. Some type of LR (1) parser would be quite adequate for most schemes.
- However, we must augment the parser with some mechanism for computing the translations.
- To compute the translation at a node A associated with a production $A \rightarrow XYZ$, we need only the values of the translations associated with nodes labeled X, Y and Z.
- These nodes will be roots of subtrees in the forest representing the partially constructed parse tree.
- The nodes X, Y and Z will become children of node A after reduction by $A \rightarrow XYZ$. Once the reduction has occurred we do not need the translations of X, Y and Z any longer.
- One way to implement a syntax-directed translator is to use extra fields in the parser stack entries corresponding to the grammar symbols.
- These extra fields hold the values of the corresponding translations.
- Let us suppose the stack is implemented by a pair of arrays STATE and VAL, as shown in Fig. (a). Each STATE entry is a pointer (or index) to the LR (1) parsing table.
- If the i-th STATE symbol is E, then VAL [i] will hold the value of the translation E. VAL associated with the parse tree node corresponding to this E.
- TOP is a pointer to the current top of the stack. We assume semantic routines are executed just before each reduction.
- Before XYZ is reduced to A, the value of the translation of Z is in VAL [TOP], that of Y in VAL [TOP + 1] and that of X in VAL [TOP + 2].
- After the reduction, TOP is incremented by 2 and the value of A. VAL appears in VAL [TOP].

Example :

- Consider an example of how a syntax-directed translation scheme can be used to specify a "desk calculator" program and how that translation scheme can be implemented by a bottom-up parser that invokes program fragment to compute the semantic actions.
- The desk calculator is to evaluate arithmetic expressions involving integer operands and the operators + and *. We assume that an input expression is terminated by $. The output is to be the numerical value of the input expression. For example, for the input expression 23 * 5 + 4$, the program is to produce the value 119.
- We use the nonterminals S (for complete sentence), E (for expression) and I (for integer). The productions are

$S \rightarrow E\$$

$S \rightarrow E + E$

$S \rightarrow E * E$

$S \rightarrow (E)$

$S \rightarrow I$

S → I digit

I → digit

- The terminals are $, +, *, parentheses and digit, which we assume stands for any of the digits 0, 1, ...9.

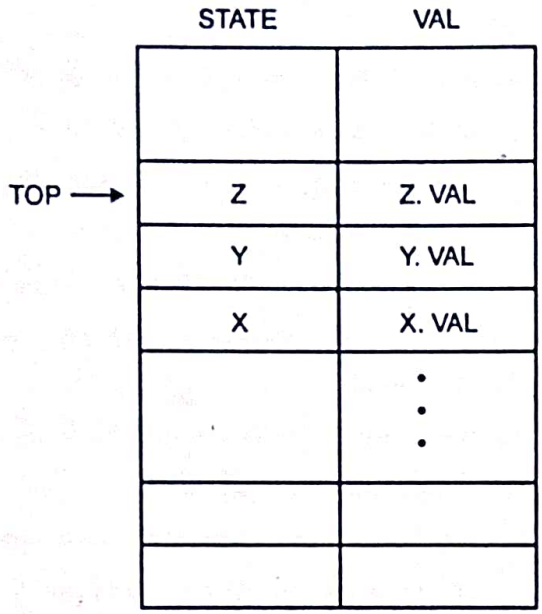


Fig. (a) Stack before reduction

- We must now add the semantic actions to the productions. With each of the nonterminal E and I we associate one integer-valued translation, called E. VAL and I. VAL, respectively, which denotes the numerical value of the expression or integer represented by a node of the parse tree labeled E or I.
- With the terminal digit we associate the translation LEXVAL, which we assume is the second component of the pair (digit, LEXVAL) returned by the lexical analyzer when a token of type digit is found.
- One possible set of semantic actions for the desk calculator grammar is shown in table (1). Using this syntax-directed translation scheme, the input 23 * 5 + 4$ would have the parse tree and translations shown in Fig. (b).
- To implement this syntax-directed translation scheme we need to construct a lexical analyzer and a bottom-up parser and we must make the parser invoke a program fragment to implement a semantic action just before making each reduction.
- A compiler-complier would tie the parser and the semantic action program fragments together, producing one module.

Table (1) Syntax-directed translation scheme for desk calculator

Sr. No.	Production	Semantic action
(1)	S → E$	{print E. VAL}
(2)	$E \rightarrow E^{(1)} + E^{(2)}$	{E. VAL := E.(1) VAL + E.(2) VAL }.
(3)	$E \rightarrow E^{(1)} * E^{(2)}$	{E. VAL := E.(1) VAL * E.(2) VAL }.
(4)	$E \rightarrow (E^{(1)})$	{E. VAL := E.(1) VAL}.
(5)	E → I	{E. VAL := I.VAL}.
(6)	$I \rightarrow I^{(1)}$ digit	{E. VAL := 10 * I.(1) VAL + LEXVAL}.
(7)	I → digit	{I. VAL := LEXVAL}.

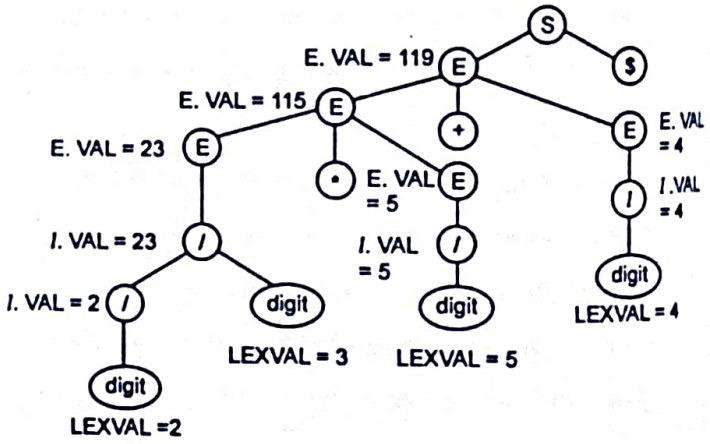


Fig. (b) Parse tree with translation

- The construction of the lexical analyzer is easy. We assume the lexical analyzer scans the input and partitions it into tokens, which here are just the terminals of the grammar.
- Every time the parser calls for a shift and needs the next token, the lexical analyzer skips over blanks to find the next nonblank input symbol, which will be one of the terminals.
- In the case that one of 0, 1, ... 9 is the next nonblank, the lexical analyzer returns the token digit and a value, denoted LEXVAL which will be the numerical value of the digit found.
- To implement the semantic actions we cause the parser to execute the program fragments of table (2) just before making the appropriate reduction.

Table (2) Implementation of desk calculator

Sr. No.	Production	Semantic action
(1)	S → E$	print VAL [TOP]
(2)	E → E + E	VAL [TOP] : = VAL [TOP] + VAL [TOP – 2]
(3)	E → E * E	VAL [TOP] : = VAL [TOP] * VAL [TOP – 2]
(4)	E → (E)	VAL [TOP] : = VAL [TOP – 1]
(5)	E → I	none
(6)	I → I digit	VAL [TOP] : = 10 VAL [TOP] * VAL [TOP] + LEXVAL
(7)	I → digit	VAL [TOP] : = LEXVAL

- Fig. e shows the sequence of moves made by the parser on input 23 * 5 + 4$.
- The contents of the STATE and VAL fields of the parsing stack are shown after each move.
- We have again taken the liberty of replacing stack states by their corresponding grammar symbols.
- We take the further liberty of using, instead of digit on the stack, its associated LEXVAL.

Table (3) Sequence of moves

Sr. No.	Input	STATE	VAL	Production used
(1)	23 * 5 + 4$	-	-	
(2)	3 * 5 + 4$	2	-	
(3)	3 * 5 + 4$	I	2	I → digit
(4)	* 5 + 4$	I3	2_	
(5)	* 5 + 4$	I	(23)	I → I digit
(6)	* 5 + 4$	E	(23)	E → I
(7)	5 + 4$	E*	(23)_	
(8)	+ 4$	E * 5	(23)_ _	
(9)	+ 4$	E * I	(23)_ 5	I → digit
(10)	+ 4$	E * E	(23)_ 5	E → I
(11)	+ 4$	E	(115)	E → E * E
(12)	4$	E +	(115)_	
(13)	$	E + 4	(115)_ _	
(14)	$	E + I	(115)_ 4	I → digit
(15)	$	E + E	(115)_ 4	E → I
(16)	$	E	(119)	E → E + E
(17)	-	E$	(119)_	
(18)	-	S	-	S → E$

- Consider the sequence of events on seeing the input symbol 2. In the first move the parser shifts the state corresponding to the token digit (Whose LEXVAL is 2) onto the stack (The state is represented by LEXVAL which is 2).
- On the second move the parser reduces by the production I → digit and then invokes the semantic action I. VAL = LEXVAL. The program fragment implementing this semantic action causes the VAL of the stack entry for digit to acquire the value 2.
- Note that after each reduction and semantic action the top of the VAL stack contains the value of translation associated with the left side of the reducing production.

EVALUATION OF EXPRESSIONS USING SEMANTIC ACTIONS

Q.15. Write in brief about evaluation of expression using semantic actions.

Ans. Semantic actions :

- A syntax-directed translation scheme is merely a context-free grammar in which a program fragment called an output action (or sometimes a semantic action or semantic rule) is associated with production.
- For example, suppose output action α is associated with production $A \rightarrow XYZ$. The action α is executed whenever the syntax analyzer recognizes in its input a substring w which has a derivation of the form $A \Rightarrow XYZ \overset{*}{\Rightarrow} w$.
- In a bottom-up parser, the action is taken when XYZ is reduced to A.

- In a top-down parser the action is taken when A, X, Y or Z is expanded, whichever is appropriate.
- The output action may involve the computation of values for variables belonging to the compiler, the generation of intermediate code, the printing of an error diagnostic, or the placement of some value in a table.
- For example, the values computed by action α quite frequently are associated with the parse tree nodes corresponding to the instance of A to which α is reduced.
- A value associated with a grammar symbol is called a translation of that symbol. The translation may be a structure consisting of fields of various types. The rules for computing the value of a translation can be as involved as we wish.
- We shall usually denote the translation fields of a grammar symbol X with names such as X. VAL, X. TRUE, and so forth.
- If we have a production with several instances of the same symbol on the right, we shall distinguish the symbols with superscripts. For example, suppose we have the production and semantic action

$E \rightarrow E^{(1)} + E^{(2)}$

{E. VAL : = $E^{(1)}$. VAL + $E^{(2)}$. VAL}

- The semantic action is enclosed in braces and it appears after the production.
- Here the semantic action is a formula which states that the translation E.VAL associated with the E on the left side of the production is determined by adding together the translations associated with the E's on the right side of the production.
- Note that the terminal symbol + in the productions is "translated" into its usual meaning by the semantic rule.
- This translation is suitable not for a compiler, but for a "desk calculator" program that actually evaluates expressions rather than generating code for them.
- In most compilers we need an action that generates code to perform the addition.
- It defines the value of the translation of the nonterminal on the left side of the production as a function of the translations of the nonterminals in the right side.
- Such a translation is called a synthesized translation.

Consider the following production and action

$A \rightarrow XYZ$ {Y.VAL : = 2 * A.VAL}

- Here the translation of a nonterminal on the right side of the production is defined in terms of a translation of the nonterminal on the left. Such a translation is called an inherited translation.
- Consider the following syntax-directed translation scheme suitable for a "desk calculator" program, in which E.VAL is an integer-valued translation.

Production	Semantic action
$S \rightarrow E^{(1)} + E^{(2)}$	{E. VAL : = E. $^{(1)}$ VAL + E. $^{(2)}$ VAL }.
E → digit	{E. VAL : = digit }.

- Here digit stands for any digit between 0 and 9.
- The values of the translations are determined by constructing a parse tree for an input string and then computing the values the translations have at each node.
- For example, suppose we have the input string 1 + 2 + 3. A parse tree for this string is shown in Fig. (a).

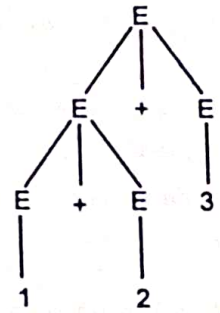


Fig.(a) Parse tree for expression 1 + 2 + 3.

- Consider the bottom leftmost E. This node corresponds to a use of the production $E \rightarrow 1$.
- The corresponding semantic actions sets E.VAL = 1.
- Thus we can associate the value 1 with the translation E.VAL at the bottom leftmost E.
- Similarly, we can associate the value 2 with the translation E.VAL at the right sibling of this node.

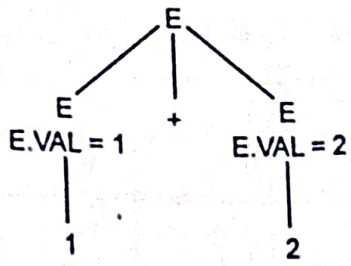


Fig.(b) Subtree with previously computed translations.

- Now consider the subtree shown in Fig. (b). The value of E.VAL at the root of this subtree is 3, which we calculate using the semantic rule

$$E.VAL := E^{(1)}.VAL + E^{(2)}.VAL$$

- In applying this rule we substitute the value of E.VAL of the bottom left most E for $E^{(1)}.VAL$ and the value of E.VAL at its right sibling E for $E^{(2)}.VAL$.
- Continuing in this manner we derive the values shown in Fig. (c) for the translations at each node of the complete parse tree.

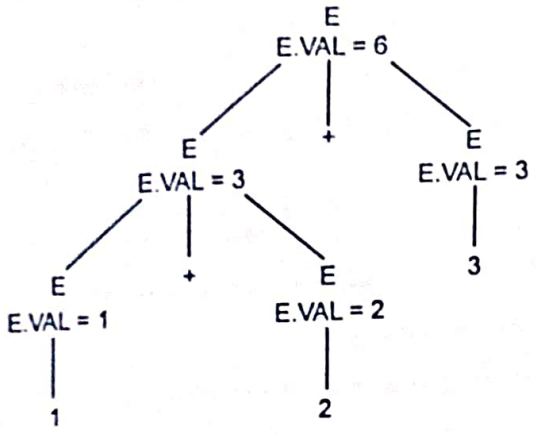


Fig. (c) Complete parse tree

- We see from this example that whenever we apply the rule

$$E.VAL := E^{(1)}.VAL + E^{(2)}.VAL$$

the values of $E^{(1)}.VAL$ and $E^{(2)}.VAL$ have been computed by a previous application of this rule or the rule E.VAL : = digit. That is to say, if we have a translation A.VAL, then the formulas for all productions with an A on the left side must produce a value that can be used for every occurrence of A. VAL in a semantic rule associated with a production having A on the right.

INTERMEDIATE CODE REPRESENTATION (POSTFIX, SYNTAX TREE, TAC)

Q.16. What are different intermediate forms of source program used in compilers? Discuss the relative advantages and disadvantages.

OR Write short note on intermediate code.

Ans. Intermediate code :

- Many compilers translate the source code into a language which is intermediate in complexity between programming language (high level) and machine code which is called as intermediate code.
- The reason for this is, if source is directly translated into machine code or assembly code, generation of optimal or relatively good code is difficult task.
- Four kinds of intermediate code are often used in compilers, they are:

(1) Postfix notation

(2) Syntax trees

(3) Quadruples

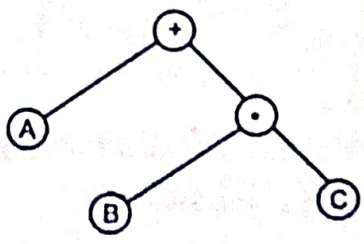
(4) Triples

(1) Postfix notation :

- In this notation, the operator is placed at the right end in the expression example ab+. If e_1 and e_2 are any two postfix expressions and O is a binary operator, the result of applying O to the values denoted by e_1 and e_2 is indicated in the postfix notation by $e_1\ e_2$.
- No parentheses are needed, because the position and priority of the operations permit only one way to decode a postfix expression.
- If O is k operators and $e_1, e_2, e_3, \ldots\ldots e_k$ are postfix expressions, then the results of applying O to these expressions, is denoted as $e_1e_2e_3\ldots e_k$.

(2) Syntax trees :

- This is a representation of source program, especially in optimizing compilers when the intermediate code needs to be extensively restructured.
- A syntax tree is variant of parse tree in which each leaf represents an operand and each interior node an operator, e.g., a syntax tree A+B*C is shown below.



(3) **Three-Address Code (Quadruples and Triples) :**

- This intermediate code is preferred in many compilers, especially those doing extensive code optimization, because it allows convenient rearrangement of code.
- It consists of sequence of statements, typically of the general form A = B Op C, where A, B, C are either programmer defined names, constants or compiler generated temporary names, Op stands for operator.
- Since each statement usually contains three addresses two for operands and one for results, such is called as three address code.
- In actual compiler, these statements can be implemented in one of the following ways :

(i) **Quadruple :**

- It is a list of records with each record consisting of four fields; this representation is known as quadruples. The fields can be labelled as OP, ARG1, ARG2, RESULT respectively.
- The OP fields contain an internal code for operator. The contents of the rest of the fields are normally pointers to the symbol table entries of names.
- A typical quadruple representation is shown below :

	OP	ARG1	ARG2	RESULT
(0)	*	A	B	T1
(1)				
(2)				

(ii) **Triples :**

- This is a list of records, each consisting of only three fields, which is labelled as OP, ARG1, ARG2 respectively.
- The contents to the records of list itself, because RESULT field is absent.
- Typical triple representation is shown below :

	OP	ARG1	ARG2
(0)	*	A	B
(1)	*	(0)	C

(iii) **Indirect triple :**

In this, a list pointer to the three address statement represented as triple maintained, as shown below.

	LIST	OP	ARG1	ARG2
(0)	(14)	(14) *	A	B
(1)	(15)	(15) *	(14)	C

Comparison :

- The postfix notation is useful if the language is mostly expressive.
- The problem with this notation is in handling flow control.
- One of the solutions is to introduce labels and conditional and unconditional jumps into postfix code.
- The postfix code can then stored in single dimensional array (in which each being either opened).
- Operands are represented by pointer to symbol table and operators by integer codes. Therefore, the table will be just an index of the array.
- Using quadruple, the location for each temporary can be immediately accessed via symbol table. In triple we have no idea unless we scan code.
- In optimizing compiler quadruple permits convenient movement of statement around, whereas in Triples the movement of a statement that defines temporarily value requires us to change all pointer to that statement in ARG1 and ARG2 fields.
- Indirect triples present no such problems because a separate list of pointer into the triple structure is maintained, hence to move the statements recording of this list is required to be done, and no change in the triple structure.
- Hence the utility of the indirect triples is almost same that of quadruples.

Q.17. Explain the term 'Three-address code'.

Ans. Three-address code :

- Syntax trees and postfix notations are of two kinds of intermediate representations.
- Three-address code is a sequence of statements of the general form

  x := y op z

  where x, y and z are names, constants or compiler-generated temporaries, op stands for an operator, such as fixed or floating point arithmetic operator, or a logical operator on Boolean valued data.

- Note that there should be only one operator on the right side of a statement.
- Thus a same language expression like x + y + z might be treated into a sequence

$t_1 := y * z$

$t_2 := x + t_1$

- The reason for the term "three-address code" is that each statement usually contains three addresses, two for the operands and one for the result.

Q.18. Explain the data structures used for representation of three address code and compare them. **CS : W-14(6M)**

Ans. Three address codes can be represented by using one of the following data structures :

(1) Quadruple :

- This is a list of record, with each record consisting of four fields. The fields can be labeled as OP, ARG1, ARG2, RESULT respectively.
- The OP fields contain an internal code for operator. The representation is shown below.

	OP	ARG1	ARG2	RESULT
(0)	+	A	B	T1
(1)				

(2) Triples :

- This is a list of records, each consisting of only three fields which can be tabled as OP, ARG1, ARG2 respectively.
- The contents of ARG1 and ARG2 fields are either pointer to the symbol table entries or pointer to the records of list itself, because RESULT fields are absent.
- Typical triple representation is shown below.

	OP	ARG1	ARG2
(0)	+	A	B
(1)	*	(0)	C

3) Indirect triple :

In this, a list of pointer three address statements represented as triples is maintained, as shown below.

	LIST
(0)	(14)
(1)	(15)

	OP	ARG1	ARG2
(14)	+	A	B
(15)	*	(14)	C

Comparison :

- Using quadruple, the location for each temporary can be immediates accessed via symbol table.
- In triple we have no idea unless we scan code in optimizing compiler.
- Quadruple permits convenient movement of statements around, whereas in Triples. The movement of a statement in ARG1 Indirect Triple and Quadruples require about same amount of space Indirect Triple can save some space, compared to Quadruples if the same temporary value is used more than once.

Q.19. What are the different types of three-address statements?

Ans. The common three-address statements which we have used are as follows :

(1) Assignment statements of the form x : = y op z, where op is a binary arithmetic or logical operation.

(2) Assignment instructions of the form x : = op y, where op is a unary operation.

(3) Copy statements of the form x : = y, where the value of y is assigned to x.

(4) The unconditional jump goto L. The three-address statement with label L is the next to be executed.

(5) Conditional jumps such as if x relop y goto y.

(6) Param x and call p, n for procedure calls and return y, where y representing a returned value is optional. Their typical use is as the sequence of the three-address statements :

param x_1

param x_2

.....

param x_n

call p, n

generated as part of a call of the procedure p ($x_1, x_2, ...x_n$)

(7) Indexed assignments of the form x : y [i] and x [i] = y.

(8) Address and pointer assignments of the form x : = & y, x : = * y and * x : =y.

Q.20. Give the scheme to produce three-address code for assignments.

Ans.

PRODUCTION	SEMANTIC RULES
S → id : E	S. code : = E. code ‖ gen (id. Place ': =' E. place)
$E \rightarrow E_1 + E_2$	E. place : = newtemp; E.code : = E_1 .code ‖ E_2 . code ‖ Gen(E.place ': =' E_1 . Place '+' E_2 . Place)
$E \rightarrow E_1 * E_2$	E.place : = newtemp; E.code : = E_1 .code ‖ E_2 . code ‖ Gen(E.place : =' E_1 place '*' E_2 place)
$E \rightarrow -E_1$	E. place : = newtemp; E.code:= E_1 code ‖ Gen (E.place ': =' 'Uminus' E_1 place).
$E \rightarrow (E_1)$	E. place : = E_1 . place; E.code := E_1 . code
E → id	E.place : = id. place; E. code : = ''

Now, the scheme to produce three-address code for assignment is

S → id : "E {p : = lookup (id. name);

if p ≠ nil, then

emit (p ': = ' E. place)

else error}

$E \rightarrow E_1 + E_2$ {E. place : = newtemp;

emit (E. place ': = ' E_1 place '+' E_2 . place)}

$E \rightarrow E_1 * E_2$ {E.place : = newtemp;

emit (E.place ': = ' E_1 . Place ': = ' '*' E_2 . place)}

$E \rightarrow -E_1$ {E.place : = newtemp;

emit (E.place ': = ' 'Uminus ' E_1 . place)}

$E \rightarrow (E_1)$ {E. place : = E_1 . place)

E → id {p : = lookup (id.name);

if p ≠ nil, then

E. place : = p

else error}

Q.21. Translate following expression

– (a * b) * (c + d) / (a * b + c) into

(i) Quadruples

(ii) Triples

(iii) Indirect triple CT : S-09(6M)

Ans.

(i) Quadruple :

Sr. No.	Operator	operand 1	operand 2	result
(1)	*	a	b	t_1
(2)	uniminus	t_1		t_2
(3)	+	c	d	t_3
(4)	*	t_2	t_3	t_4
(5)	+	t_1	c	t_5
(6)	/	t_4	t_5	t_6

(ii) Triples :

Sr. No.	Operator	operand 1	operand 2
(1)	*	a	b
(2)	uniminus	(1)	
(3)	+	c	d
(4)	*	(2)	(3)
(5)	+	(1)	c
(6)	+	(4)	(5)

(iii) Indirect triples :

Sr. No.	Statement	Operator	operand 1	operand 2
(1)	(14)	+	a	b
(2)	15	–	(14)	
(3)	16	+	c	d
(4)	17	*	15	16
(5)	(18)	+	14	c
(6)	19	+	17	18

Q.22. Translate the following expression :

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

(i) Quadruple

(ii) Triples

(iii) Indirect triples.

Describe various TAC and translate the expression

- (a + b) * (c + d) + (a + b + c) into quadruple and triples.

CT : W-11(6M), S-13,W-13(8M)

Ans.

(i) Quadruple :

Sr. No.	Operator	Operand 1	Operand 2	Result
(1)	+	a	b	t2
(2)	–	t1		t2
(3)	+	c	d	t3
(4)	*	t2	t3	t4
(5)	+	t1	c	t5
(6)	+	t4	t5	t6

(ii) Triples :

Sr. No.	Operator	Operand 1	Operand 2
(1)	+	a	b
(2)	–	1)	
(3)	+	c	d
(4)	*	2)	3)
(5)	+	1)	c
(6)	+	4)	5)

(iii) Indirect triples :

Sr. No.	Operator	Operand 1	Operand 2
(1)	+	a	b
(2)	–	1)	
(3)	+	c	d
(4)	*	2)	3)
(5)	+	1)	c
(6)	+	4)	5)

Q.23. Generate TAC for following statement using SDTS.

A : = – B * (C + D). **CS : S-11,12(8M)**

Ans. The three address code will be

$t_1 = -B_1$

$t_2 = C + D$

$t_3 = t_1 * t_2$

$a = t_3$

POINTS TO REMEMBER :

(1) *The attributes associated with a grammar symbol are classified into two categories :*

(i) Synthesized attributes.

(ii) Inherited attributes.

(2) *An attribute is said to be synthesized if its value at a parse tree node is determined by the attribute values at the child nodes.*

(3) *Inherited attributes are those whose initial value at a node in the parse tree is defined in terms of the attributes of the parsing or siblings of that node.*

(4) *While translating a source program into a functionally equivalent object code representation, a parser may first generate an intermediate representation that is called as intermediate code generation.*

VBD

shown in following Fig. (b) :

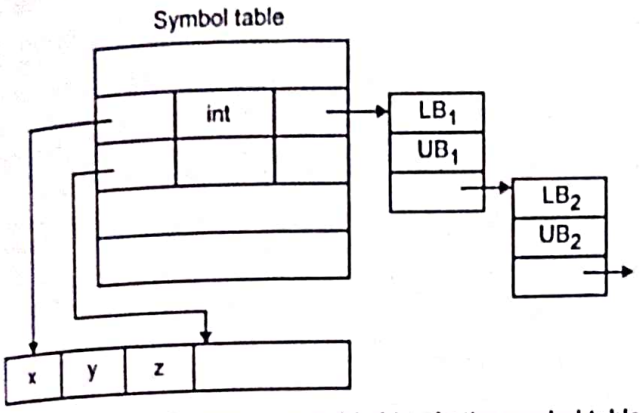


Fig.(b) Symbol table names are held either in the symbol table record or in a separate string table.

Q.14. What are the factors which should be considered while designing a symbol table?

Ans. While designing a symbol table for a practical situation, the following factors should be decisive in terms of the overall implications of the decision.

(1) Entry format :

- While deciding the entry format, one should examine the information which is to be stored in the symbol table to find out whether fixed length entries can be used without excessive memory sacrifices for unused fields in the entry.
- If certain fields in the entry are very rarely required, then it is better to consider variable length entries with all the attendant implications regarding the organization and access speeds.

(2) Memory availability :

- This has important bearing on the matter of the entry format and on the table organization.
- If availability is not a critical factor, then fixed length entries with a hash variant would be preferable.

(3) Choice of the organization :

- Considerations (1) and (2) would decide the entry format. Depending on the format, a feasible organization should be selected. For variable entry lengths, the choice is from sequential, linked list or binary tree organization.
- While binary tree organization is superior in terms of access speeds and a priori memory commitments, the loss of positional determinacy is an important factor.
- Since it implies that we must keep track of the entire address of the entry, rather than merely its number in the table, in order to refer to it.
- In sequential organizations the memory commitments are a one-time decision and the search is slow but additions are fast.
- Linked list symbol table have been used in practice even for fixed length entries.
- The major advantage of this is the fact that no commitment needs to be made to support a fixed size symbol table irrespective of a program's requirement.

Q.15. What is an ordered and unordered symbol table? Explain.

CT : W-11, S-13(3M)

Ans. Ordered symbol table :

- When symbol table is implemented with ordered set of link list then the symbol table is called as ordered symbol table.
- It require O(log n) time for binary search.
- It require O(n) time for insertion.

Advantages :

(1) It is simple and good method if table is known in advance.

(2) Implementation is relatively easy.

Disadvantage :

(1) It have insertion very expensive as it required more time for insertion.

Unordered symbol table :

- When symbol table is implemented using an array or linked list then that symbol table is called as unordered list symbol table.
- It require O(n) time for searching.
- O(1) time for insertion of data in symbol table.
- It is very simple and compact in its advantage.

Advantages :

(1) It requires less space.

(2) It is easy to implement.

Disadvantages :

(1) It requires more access time.

(2) It is too slow in practice.

CODE OPTIMIZATION

Q.16. What is optimization? What are its types?

Ans. Optimization :

- Optimization is the process of transforming a piece of code to make more efficient without changing its output or side-effects.

- Optimization technique can be classified into two types :

  (1) Machine dependent optimization.

  (2) Machine independent optimization.

(1) Machine dependent optimization :

It refers to those optimization which cannot be performed without the knowledge of the target machine for which code is to be generated.

Example : Attempt to generate the object code, which will utilize the registers of the target machine move efficiently.

(2) Machine independent optimization :

It refers to those optimization which can be performed independent of target machine for which compiler is supposed to generate code.

Example : Elimination of loop invariant computation.

Q.17. What is code optimization?

Ans. Code optimization :

- The technique used by the compiler to improve the execution efficiency of the generated object code is called the code optimization.
- This process involves complex analysis of the intermediate code performing various transformations but it is also required that the compiler should not attempt any transformation which may lead to change in the semantic of the program.
- Actually optimization is a misnomer, because even after performing various optimizing transformations by the compiler, there is no guarantee that the generated object code will be optimal, hence actually it is the code improvement.

Q.18. Explain different code optimization techniques in brief.

CT : S-13(6M)

Ans. The code optimization techniques are as follows :

(1) Common sub-expression elimination

(2) Copy propagation

(3) Dead code elimination

(4) Loop optimization

(1) Common sub-expression elimination :

- An occurrence of an expression E is called a common sub-expression if E was previously computed and the values of variables in E have not changed since the previous computation.
- We can avoid recomputing the expression if we can use the previously computed value.
- This can be eliminated by using DAG representation of basic block.
- For example, block B_5 shown in figure (a) recalculates $4 * i$ and $4 * j$. The assignment to t_7

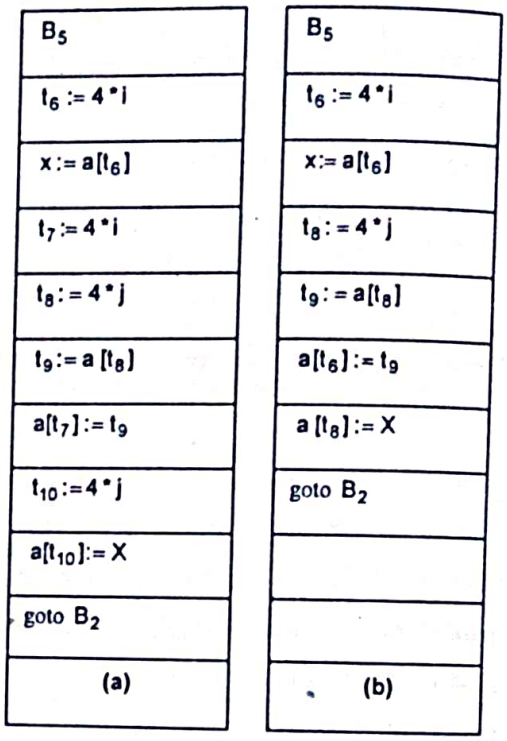


Fig. Local common sub-expression elimination

and t_{10} have the common sub-expression $4 * i$ and $4 * j$ respectively, on the right side in Fig. (a). They have been eliminated in Fig. (b) by using t_6 instead of t_7 and t_8 instead of t_{10}.

(2) Copy propagation :

- The assignment statements of the form $f := g$ called copy statement, or copies for short Consider a basic block

  $x := t_3$

  $a[t_2] := t_5$

  $a[t_4] := x$

  goto B_2

- This can be further improved by eliminating x using two new transformations.
- The idea behind the copy-propagation transformation is to use g for f, wherever possible after the copy statement $f := g$.
- The assignment $x := t_3$ in block B_5 is a copy. Copy propagation applied to block yields.

  $x := t_3$

  $a[t_2] := t_5$

$a[t_4] := t_3$

goto B_2

(3) **Dead code elimination :**

- A variable is live at a point in a program if its value can be used subsequently, otherwise, it is dead at that point.
- A related idea is dead or useless code, statements that compute values that never get used.
- While the programmer is unlikely to introduce any dead code intentionally, it may appear as the result of previous transformations.
- If we use following statement at various points in the program, the statement is like

  If (debug) print .....
- By a data-flow analysis, it may be possible to deduce that each time the program reaches this statement.
- Deducing at compile time that the value of an expression is constant and using the constant instead is known as constant flooding.
- One advantage of copy propagation is that it often turns the copy statement into dead code.

Consider the basic block which is considered previously.

$x := t_3$

$a[t_2] := t_5$

$a[t_4] := t_3$

goto B_2

- The above basic block is after copy propagation. The dead code elimination removes the assignment to x and transforms into

$a[t_2] := t_5$

$a[t_4] := t_3$

goto B_2 .

(4) **Loop optimization :**

- Major source of optimization is the loop optimization since majority of execution time is spent on performing the computation that are inside the inner loops. Loop invariant computation elimination and elimination of induction variables are the important loop optimization.
- Loop invariant computation means the computations, computing a value which is not dependent on how many times loop gets executed. By data flow analysis we can eliminate the loop invariant computations.

Q.19. Explain machine independent and machine dependent code generation. What are their relative advantages?

CT : W-11(3M), S-14(4M)

Ans. Machine independent code generation :

In this optimization the compiler takes in the intermediate code and transforms a part of the code that does not involve any CPU register and/or absolute memory locations.

Example :

```
do
{
   item = 10;
   value = value + item;
}
     while (value < 100);
```

This code involves repeated assignment of the identified item which if we put this way :

```
item = 10;
do
{
     value = value + item;
}
     while (value < 100);
```

This code should not same the CPU cycles but can be used on any processor.

Machine dependent code generation :

- It is done after the target code has been generated and when the code is transformed according to the target machine architecture.
- It involves CPU register and may have absolute memory references rather than relative references.
- Machine dependent optimization put effects to take minimum advantage of memory hierarchy.

Q.20. Discuss machine dependent optimization techniques.

CS : W-13(6M)

Ans. Machine dependent optimization techniques are as follows :

(1) Special instruction.

(2) Code recordering.

(1) **Special instruction :**

- Most assemblers provide special instructions that facilitate the generation of the code in assembly language
- For instance, in masm
  - loop, loope, loopz, loopne, loopnz are used
- It implement loops with a counter i.e decremented at each iteration, even though there is no corresponding instruction in machine code.

(2) **Code recordering :**

- The underlying idea is that instruction may influence the size of the final program.
- The following is an example of the final program

  a : = b/c;

  d : = b % c;
- It can produce the following code in assembly language :

```
Mov dword eax, [b]
cdq
idiv dword eax, [c]
mov dword [a], eax
mov dword eax, [b]
cdq
idiv dword eax,[c]
mov dword [d], edx
```

- That code not as efficient as it could be, as the instruction idiv leaves the remainder of the division in eax, it is not necessary to repeat it.
- The following code is equivalent :

```
mov dword eax, [b]
cdq
idiv dword eax, [c]
mov dword [a], eax
mov dword [d], edx
```

Q.21. With the help of an example prove that order of computation affect cost of object program. Give one algorithm to find an optimize order of computation. **CS : S-14(13M)**

Ans. Consider the expression

$x = (a + b) - ((c + d) - e)$

The TAC for this is as follow

$t_1 = a + b$

$t_2 = c + d$

$t_3 = t_2 - e$

$x = t_1 - t_3$

Applying the algorithm of code generation in above results for idx becomes as follows :

Statement	I.	Instruction generated	Register descriptor	Address descriptor
			All register empty	
$t_1 = a + b$	R_0	MOV a, R_0 ADD b, R_0	R_0 will hold t_1	t_1 in R_0
$t_2 = c + d$	R_1	MOV c, R_1 ADD d, R_1	R_1 will hold t_2	t_2 in R_1
$t_3 = t_2 - e$	R_1	SUB e, R_1	R_1 will hold t_3	t_3 in R_1
$x = t_1 - t_3$	R_0	SUB R_1, R_0	R_0 will hold x	x in R_0
		MOV R_0, x		x in R_0 and memory

- The algorithm makes use of next use information of each name x order to make more informed decision regarding register allocation.
- So it is required to compute the next use information if :

(i) A statement at the index i in a block assigns a value to name x.

(ii) And if a statement at the index j in the same block uses x as an operand,

(iii) And if the path from the statement at index i to the statement at index j is a path without any intervening assignment to name x, then we say that the value of x computed by the statement at index i is used in the statement at index j.

- Consider the basic block :

$t_1 = a + b$

$t_2 = c + d$

$t_3 = e - t_2$

$t_4 = t_1 - t_3$

When straightforward code generation is done using the following algorithm as follows :

for each statement i of the

form x = y op z do

{

attach information about the next uses of x, y and z to statement i

Set the information for x to no next use

Set the information for y to z to be the next use in statement i

}

- The code generation is done if and only if two register R_0 and R_1 are available, then the generated code is as shown in following table :

Statement	L	Instruction Generated	Cost	Register Descriptor	Address descriptor
				R_0 and R_1 are empty	
$t_1 = a + b$	R_0	MOV a, R_0 ADD b, R_0	2 words 2 words	R_0, will hold t_1	t_1 is in R_0
$t_2 = c + d$	R_1	MOV c, R_1 ADD d, R_1	2 words 2 words	R_1 will hold t_2	t_2 is in R_1
$t_3 = e - t_2$		MOV R_0, t_1 (generated by getreg( ))	2 words		t_1 is in R_0
	R_0	MOV e, R_0 SUB R_1, R_0	2 words	R_0 will hold t_3 R_1 will be empty because t_2 has no next use.	
$x = t_1 - t_3$	R_1	MOV t_1, R_1 SUB R_0, R_1	2 words 1 word	R_1 will hold x R_0 will be empty because t_3 has no next use.	x is in R_1
		MOV R_1, x	2 words		x is in R_1 and memory

- We see that total length of instruction sequence generated is 18 memory words.

- If rearrange the final computation as :

$t_2 = c + d$

$t_3 = e - t_2$

$t_1 = a + b$

$t_4 = t_1 - t_3$

and then generate the code, will get following table :

Statement	L	Instruction Generated	Cost	Register Descriptor	Address descriptor
$t_2 = c + d$	R_0	MOV c, R_0 ADD d, R_0	2 words 2 words	R_0 will hold t_2	t_2 is in R_0
$t_3 = c - t_2$	R_1	MOV c, R_1 SUB R_0, R_1	2 words 1 words	R_1 will hold t_3 R_0 will be empty because t_2 has no next use.	t_3 is in R_1
$t_1 = a + b$	R_0	MOV a, R_0 ADD b, R_0	2 words 2 words	R_0 will hold t_1.	t_1 is in R_0
$x = t_1 - t_3$	R_1	SUB R_1, R_0	1 word	R_0 will hold x R_1 will be empty because t_3 has no next use.	x is in R_0
		MOV R_0, x	2 word		x is in R_0 and memory.

- Hence the length of the instruction sequence generated is 14 memory words.
- This indicate that the order of computation is a deciding factor in the cost of code generated.
- In above example cost is reduced when order of $t_2 - t_3 - t_1 - t_4$ is used because t_1 adds left operand.
- Hence no intermediate store and load is require.

- Optimize order of computation is as follows :

  It list the nodes of a DAG and rearrange listing results of computation order.

```
{
    While there exists an unlisted interior node.
{
    Select an unlisted node n whose parent has been listed
    list n
  {
    Select unactive used
    User access part
    {
        list m
        m = n
    }
  }
Reverse of matrix after list of node
```

SOURCES OF OPTIMIZATION

Q.22. State the sources of optimization.

OR What are the sources of code optimization? **CT : W-12(2M)**

Ans. There are three sources of optimization as follows :

(1) Local optimization.

(2) Loop optimization.

(3) Data flow analysis.

Q.23. Explain local optimization and common sub expression.

CT : W-12(2M)

Ans. Local optimization :

- The transformation of a program is called local if it can be performed by looking only at the statements in a basic block, such as function preserving transformation.
- Common subexpression elimination, dead code elimination, constant folding are common examples.

  Example : Common sub-expression elimination
- An occurrence of expression $\in$ is called as common sub expression if $\in$ was previously computed and values of variables in $\in$ have not changed since the previous computation.
- Thus, the recomputation of expression can be avoided, if we can use the previously computed value.
- This elimination of common sub-expression can be done by using DAG (Directed acyclic graph) representation of basic block reminder a block,

  Consider a block ;

  $T_1 = 4 * I$

  $T_2 = addr\ (A) - 4$

  $T_3 = T_2\ [T_1]$

  $T_4 = 4 * I$

  $T_5 = addr\ (B) - 4$

  $T_6 = T_5\ [T_4]$

  $T_7 = T_3 * T_6$
- In the above block T_1 and T_4 are common sub expression, hence we can eliminate T_4 and can use T_1 everywhere in place of T_4.

  This optimized block will be,

  $T_1 = 4 + I$

  $T_2 = addr\ (A) - 4$

  $T_3 = T_2\ [T_1]$

  $T_5 = addr\ (B) - 4$

  $T_6 = T_5\ [T_1]$

  $T_7 = T_3 * T_6$
- Consider the basic block which is considered previously.

  $X := t_3$

  $a[t_2] := t_5$

  $a[t_4] = t_3$

  goto B_2
- The basic block is after copy propogation. The dead code elimination removes the assignment to X and transforms into ;

  $a[t_2] := t_5$

  $a[t_4] := t_3$

  goto B_2

Q.24. Explain how to remove local common sub expression with suitable example. **CT : S-11(2M)**

OR Write short note on elimination of local common subexpression.

CS : W-14(3M)

4.12 D

Ans. Local common subexpression has two steps as follows :

(1) First step is to detect the common sub expression in a basic block.

(2) The common sub expression can automatically detected if we construct a Directed Acyclic Graph (DAG).

DAG construction :

- For construction of DAG we make use of the function node (id) which returns the most recently created node associated with id.
- For every three address statements x = y op z, x = op y or x = y in the block,

do

{

(1) If node (y) is undefined create a leaf labeled y and let node (y) be this node.

(2) If node (x) is undefined create a leaf node z and let that leaf be node (z).

(3) If statements is of the form x = op y or x = y then if node (y) is undefined create a leaf labeled y and let node (y) be this node.

(4) If a node exists that is labeled op whose left child is node (y) and whose right child in node (y) then return this node.

(5) Otherwise create such node and return it. If the statement is of the form x = op y then check if a node exists that is labeled op whose only child is node (y). Return this node otherwise create such node and return. Let the returned node be n.

(6) Append x to the list of identifier for the node n returned in step (2). Delete x from the list of attached identifiers for node(x) and set node (x) to be node n.

}

Example :

Consider an example which shows the elimination of common sub expression.

(1) $S_1 := 4 * I$

(2) $S_2 := addr\ (A) - 4$

(3) $S_3 := S_2[S_1]$

(4) $S_4 := 4 * i$

(5) $S_5 := addr\ (B) - 4$

(6) $S_6 := S_5[S_4]$

(7) $S_7 := S_3 * S_6$

(8) $S_8 := PROD + S_7$

(9) $PROD := S_8$

(10) $S_9 := I + 1$

(11) $I := S_9$

(12) If $I \leq 20$ goto (1)

DAG representation diagram :

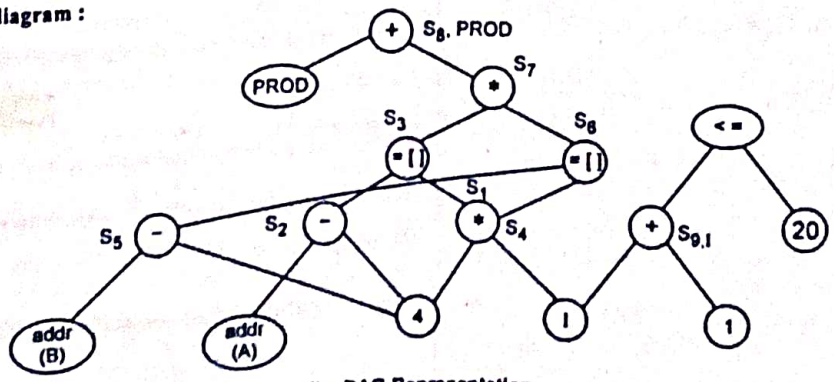


Fig. DAG Representation

Q.25. Explain method to eliminate global common sub expression.

CS : S-09(7M)

OR How global common subexpression can be eliminated with the help of "Available" information? Explain with the help of example.

CS : S-14(13M)

Ans.

- Global common subexpression computes value but in different basic blocks. For detection of such expression, we need to compute available expression.

Available expression :

- An expression x op y is available at point P if every path from the initial node of x op y and if after the last such evaluation and prior to reaching p there are no subsequent assignment, to x or y.
- To eliminate such global common subexpression are need to compute the set of all the expression available at the point just before the start of every block and end of every block. This is called as IN(B) and OUT(B).
- The computation of IN(B) and OUT(B) required the set of expression generated and killed by the basic block.
- To compute basic block solve the following expression :

$OUT(B) = IN(B) - KILL(B) \cup GEN(B)$

$IN(B) = OUT(B)$

Algorithm :

(1) $IN(B_1) = \phi$

$OUT(B_1 = GEN(B_1))$

(2) For $(i = 2; i \le n; i++)$

{

$IN(B) = U$

$OUT(B) = U - GEN(B)$

}

(3) Flag = True

(4) While (flag) do

{

flag = false

for $(i = 2; i \le n; i++)$

{

$IN_{new} = \phi$

For each predecessor P of B_i

$IN_{new}(B_i) = IN_{new}(b_i) \cup OUT(P)$

If $IN_{new}(B_i)$ is not equal to $IN_{(B_i)}$ then

{

flag = true

$IN(B_i) = IN_{new}(B_i)$

$OUT(B_i) = IN(B_i) - KILL(B_i) \cup GEN(b_i)$

}

}

}

- Now after computing IN and OUT we will start eliminating of global common subexpression.
- For every statements 'S' of the form x = y op z such that y op z is available at the beginning of the block containing S and neither y nor z is defined prior to the statement x = y op z in that block do.

(a) Find all the definition reaching up to the 'S' statement block that have y op z on the right.

(b) Create new temp.

(c) Replace each statement U = y op z found in the step (1) by :

temp = y op z

U = temp

(d) Replace the statement x = y op z in block by x = temp.

LOOP OPTIMIZATION

Q.26. Write short note on loop optimization.

CS : S-10,11(4M), CT : W-11(7M)

OR Explain loop-optimization with an example.

OR What is the role of loop-optimization in code optimization process?

CT : W-12(5M)

OR What are different loop optimizations? Explain in brief.

CS : W-11(7M)

Ans. Loop optimization :

- Loop optimization is the most valuable machine independent optimization because a program's inner loops are good candidates for improvement.
- A loop invariant computation is one that computes the same value every time a loop is executed. Therefore, moving such a computation outside the loop leads to a reduction in the execution time.
- Induction variables are those variables used in a loop, their values are in lock-step, and hence, it may be possible to eliminate all except one.
- The important techniques of loop optimization are, code motion, induction variable elimination and reduction in strength.

(1) Code Motion :

- An important modification that decreases the amount of a code in a loop is code motion.
- This transformation takes an expression that yields the same result independent of the number of times a loop is executed (a loop invariant computation) and places the expression before the loop.
- Note that the notion "before the loop" assumes the existence of an entry for the loop.
- For example, evaluation of limit – 2 is a loop-invariant computation in the following while statement.

while (i <= limit – 2) */* statement does not change limit */*

code motion will result in the equivalent of

t = limit –2;

while (i<= t) */* statement does not change limit or t */*

(2) Induction variables and reduction in strength :

While code motion is not applicable to the quick sort example we have been considering the other two transformations.

Loops are usually processed inside out. For example, consider the loop around B_3. Only the portion of the flow graph relevant to transformations on B_3 is shown in fig.

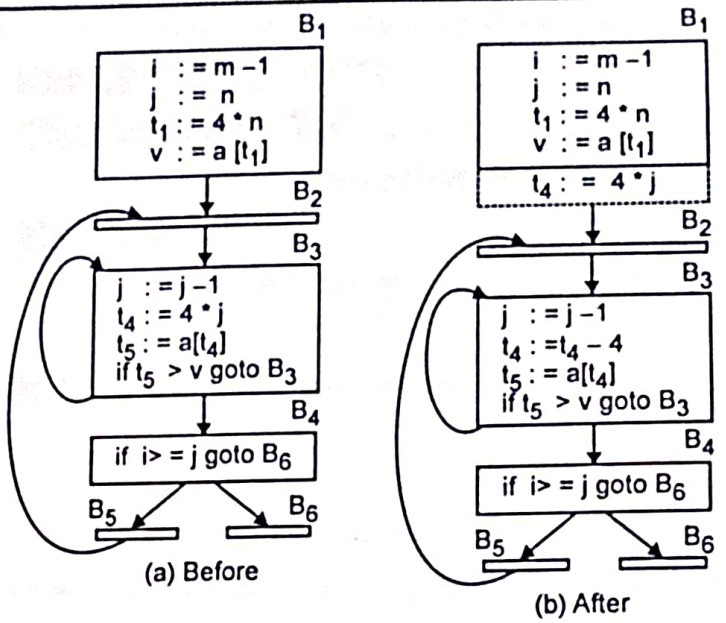


Fig. Strength reduction applied to 4 * j in block B_3

- Note that the values of j and t_4 remain in lock-step; every time the value of j decreases by 1, that of t_4 decreases by 4 because 4*j is assigned to t_4. Such identifiers are called induction variables.
- When there are two or more induction variables in a loop, it may be possible to get rid of all but one by the process of induction variable elimination. For the inner loop around B_3 in Fig. we cannot get rid of either j or t_4 completely; t_4 is used in B_3 and j in B_4.
- However, we can illustrate reduction in strength and illustrate a part of process of induction variable elimination.
- Eventually j will be eliminated when the outer loop of $B_2 - B_5$ is considered.

Example : As the relationship $t_4 = 4 * j$ surely holds after such an assignment to t_4 in Fig. and t_4 is not changed elsewhere in the inner loop around B_3, it follows that just after the statement j:= j – 1 the relationship $t_4 = 4 * j - 4$ must hold.

Q.27. Define dominator. **CS : S-09 (2M)**

OR Write short note on dominators and dominator computing algorithm. **CT : S-10 (4M)**

OR What is dominator? How is it used to detect a loop?

CT : S-14(3M), W-10(4M), S-12(5M).

CS : W-14(2M), S-11, W-11(3M)

OR How does dominators help to detect the loops in program?

CT : S-11(3M), W-12(2M)

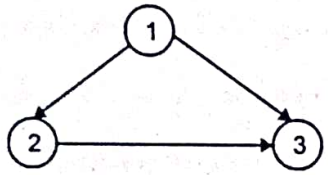
OR Illustrate steps to be carried out for loop detection with the help of an example. **CS : W-13(7M)**

OR How a loop can be detected in given TAC? **CT : W-09(7M)**

Ans. Dominator :

- If every path from the initial node of the flow graph to n goes through node d then we can say that node d dominates node n.
- Under this definition, every node dominates itself and the entry of the loop dominates all node in the loop.

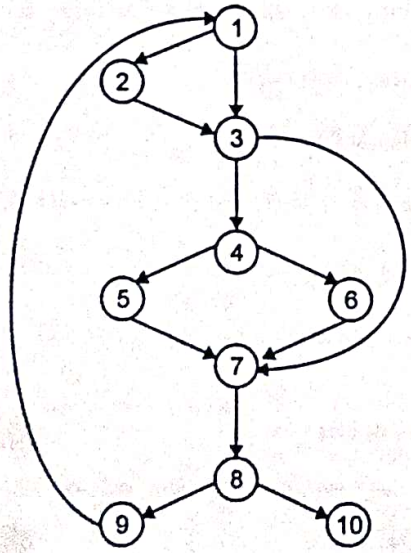
Example :



- In this flow graph, the dominator of node 3 is node 1 only, since all the paths reaching upto the node 3 from node 1 does not go through node 2.
- The properties of the Dominator (DOM) are as follows :

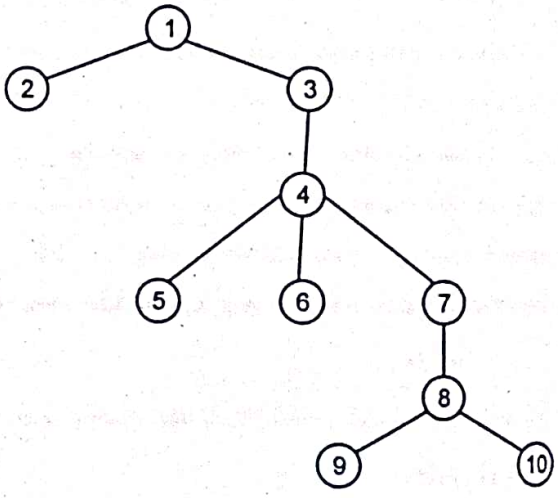
  (1) They are reflexive, i.e. every node dominates itself.

  (2) They are transitive, i.e. if a dom b and b dom c implies a dom c.
- For example, consider a flow graph ;



Node no.	Dominator set
1	1
2	1, 2.
3	1, 3.
4	1, 3, 4.
5	1, 3, 4, 5.
6	1, 3, 4, 5, 6.
7	1, 3, 4, 7.
8	1, 3, 4, 7, 8.
9	1, 3, 4, 7, 8, 9.
10	1, 3, 4, 7, 8, 10.

- For storing the set of dominator, static or dynamic dist are used.
- But this requires large amount of space. Thus an alternative representation is used i.e. tree representation.



- The existence of dominator tree is based on the property that each node n has unique immediate dominator m, such that m is the last dominator of n on any path from the initial node to n.
- In terms of dom relation, the immediate dominator M has property that if d ≠ n and d dom n, then d dom m.
- Thus, the immediate dominator in above flow graph will be,

Node no.	Immediate dominator
1	1
2	1
3	1

4	3
5	4
6	4
7	4
8	7
9	8
10	8

Application of dominator :

(1) One of the important application of dominator is to determine the loops of a flow graph. There are 2 properties of such loops :

(a) A loop must have a single entry point called the 'Header'.

(b) There must be at least one way to iterate the loop, i.e. at least one path back to the header.

(2) The best way of finding all loops in a flow graph is to search for edges in the flow graph always heads dominators their tails.

Loop detection :

- In a flow graph loop can be defined as a cycle that satisfies two properties :

(a) The loop should have a single node or header, so that it will be possible to move all of the loop invariant computation in a unique place i.e. in front of header.

(b) Loop should be strongly connected in other words it should be possible to go from any node of loop to any other node while staying within the loop. This is needed until at least some of the loops get executed repeatedly.

- When the flow graph contains one or more back edges, then only one or more loops / cycles, are there in flow graph.
- Thus for detection of loop identification of back edges is required.

Identification of the back edges :

- To detect loop, it is necessary to identify the back edges in the flow graph.
- If there is no back edge then there is no loop and if there is back edge then there may exist one or more loop formed by back edge.

Back edge detection :

- For back edge section, we do depth first search of the program flow graph and during the search when a node is visited a unique number associated with it is called as depth first number.
- This number will then be used to decodes whether a –→ b is back edge or not.

Algorithm for back edge detection :

```
Number = Total number of nodes in a program
                flow graph
search (m)
{
    if (m is not visited) then
    {  mark m = visited ;
    DFN (m) = Number ;
    Number = Number – 1;
    For every successor s of m do
    Search (s) :
    } // if end.
main( )
    {
      Search (n);              /*no. is initial node of graph */
    } // main end
}  // search end
```

Q.28. Find the dominator of each node of following flow graph. Also construct dominator tree. **CS : IV-14(5M)**

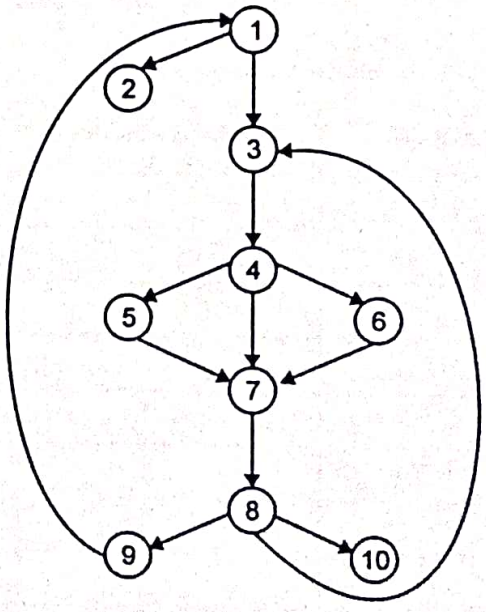
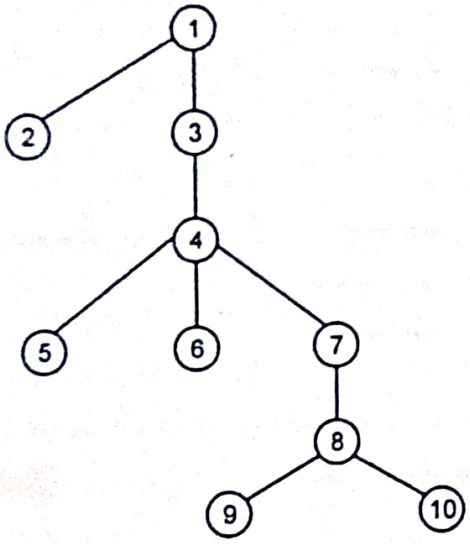


Fig. Flow graph

Ans. Dominator set are as follows :

Node no.	Dominator set
1	1
2	1, 2
3	1, 3
4	1, 3, 4
5	1, 3, 4, 5
6	1, 3, 4, 6
7	1, 3, 4, 7
8	1, 3, 4, 7, 8
9	1, 3, 4, 7, 8, 9
10	1, 3, 4, 7, 8, 10

Dominator tree is as follows :



Immediate dominator in flow graph is as follows :

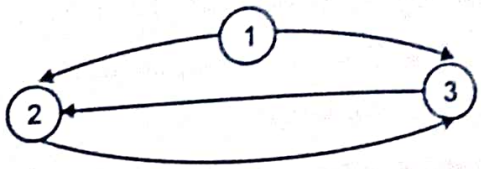
Node no.	Immediate dominator
1	1
2	1
3	1
4	3
5	4
6	4
7	4
8	7
9	8
10	8

Q.29. Define reducible flow graph. CS : S-09(2M)

OR What do you mean by reducible flow graph? Explain with suitable example. CS : W-10(6M), W-12(4M)

OR What is need of reducible flow graph? CT : S-12(2M)

OR What is meant by reducible flow graph? Is following graph reducible? Justify your answer. CT : W-12(5M), S-11(4M)



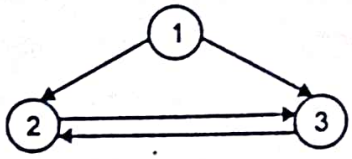
OR Write short note on reducible flow graph. CT : S-14(4M)

Ans. Reducible flow graph :

- A flow graph G is said to be reducible, if its edges can be partitioned into two disjoint groups, forward edges and backward edges.
- Reducible flow graph has two properties :

  (1) The forward edges form an acyclic graph in which every node can be reached from the initial node G.

  (2) The back edges are those edges whose heads dominates their tails.
- For example, consider the flow graph.



- In above flow graph, no edges head dominates the tail of that edge, thus there is no back edge.
- It could has been a reducible graph, if the entire graph had been acyclic.
- But that is not the case. Thus it is not a reducible flow graph.

Q.30. Define induction variable. CS : S-09(2M)

OR Write short note on elimination of induction variable. CS: W-14(3M)

Ans. Induction variable :

- We define induction variable of a loop as those name whose only assignments within the loop are of the form.

I = I ± C, where C is a constant or a name whose value does not change within the loop.

- A basic induction variable may or may not form an arithmetic progression at the loop header.
- For example, consider the flow graph shown in Fig. In the loop formed by B_2, I is a basic induction variable.

Algorithm for detecting and eliminating induction variables :

An algorithm exists that will detect and eliminate induction variables. Its method is as follows :

(1) Find all of the basic induction variables by scanning the statements of loop L.

(2) Find any additional induction variables and for each such additional induction variable A, find the family of some basic induction B to which A belongs. (If the value of A at the point of assignment is expressed as $C_1B + C_2$, then A is said to belong to the family of basic induction variable B). Specifically, we search for names A with single assignments to A within loop L, and which have one of the following forms :

A = B * C

A = C * B

A = B / C

A = B ± C

A = C ± B

Where, C is a loop constant, and B is an induction variable, basic or otherwise. If B is basic, then A is in the family of B. If B is not basic, let B be in the family of D, then the additional requirements to be satisfied are :

(a) There must be no assignment to D between the lone point of assignment to B in L and the assignment to A.

(b) There must be no definition of B outside of L reaches A.

(3) Consider each basic induction variable B in turn. For every induction variable A in the family of B :

(a) Create a new name, temp.

(b) Replace the assignment to A in the loop with A = temp.

(c) Set temp to $C_1B + C_2$ at the end of the preheader by adding the statements :

```
temp = C1 * B
temp = temp + C2   /* omit if C2 = 0 */
```

(d) Immediately after each assignment B = B + D, where D is a loop invariant, append :

```
temp = temp + C1 * D
```

If D is a loop invariant name, and if $C_1 \neq 1$, create a new loop invariant name for $C_1 * D$, and add the statements :

```
temp1 = C1 * D
temp = temp + temp1
```

(e) For each basic induction variable B whose only uses are to compute other induction variables in its family and in conditional branches, take some A in B's family, preferably one whose function expresses its value simply, and replace each test of the form B reloop X goto Y by :

```
temp2 = C1 * X
temp2 = temp2 + C2   /* omit if C2 = 0 */
if temp reloop temp2 goto Y
```

Delete all assignments to B from the loop, as they will now be useless.

(f) If there is no assignment to temp between the introduced statement A = temp (step 1) and the only use of A, then replace all uses of A by temp and delete the statement A = temp.

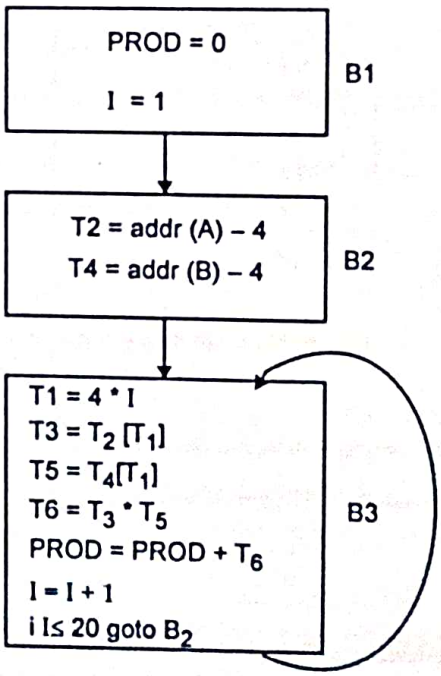


Fig. (a) Flow graph where I is a basic induction variable

In the flow graph shown in Fig. (a) we see that I is a basic induction variable, and T1 is the additional induction variable in the family of I, because the value of T1 at the point of assignment in the loop is expressed as T1 = 4 * I. Therefore, according to step 3b, we replace T1 = 4 * I by T1 = temp. And according to step 3c, we add temp = 4 * I to the preheader. We then append the statement temp to temp + 4 after statement (10), as shown in Fig. (a) as per step 3d. And according to step 3e, we replace the statement if $I \leq 20$ goto B2 by :

temp1 = 80

if (temp ≤ temp1) goto B2, and delete I = I + 1

The results of these modifications are shown in Fig. (b).

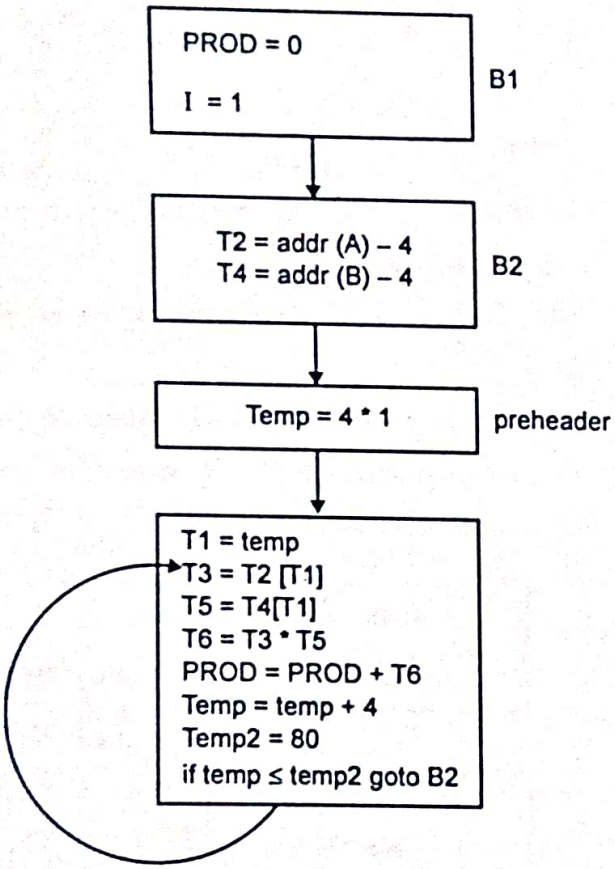


Fig. (b) Modified flow graph

Q.31. How does strength reduction and dead code elimination help for code optimization? Explain with suitable example.

CT : S-11(7M)

Ans.

(1) Strength reduction :

- Loops are usually processed inside-out. For example, consider the loop around B_3. Only the portion of the flow graph relevant to transformations on B_3 is shown in Fig. below.
- The values of j and t_4 remain in lock-step; every time the value of j decreases by 1, that t_4 decreases by 4 because 4*j is assigned to t_4 such identifiers are called induction variables.
- When there are two or more induction variables in a loop, it may be possible to get rid of all but one by the process of induction-variable elimination.

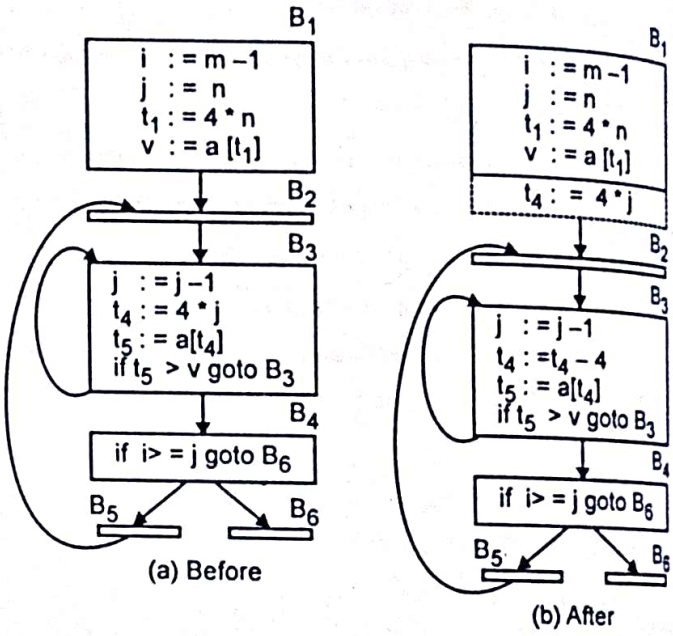


Fig. Strength reduction applied to 4 * j in block B_3

- For the inner loop around B_3 in Fig. (a), we cannot get rid of either j or t_4 completely; t_4 is used in B_3 and j in B_4.
- However, we can illustrate reduction in strength and illustrate a part of process of induction variable elimination. Eventually j will be eliminated when the outer loop of $B_2 - B_5$ is considered.

(2) Dead code elimination :

- A variable is live at a point in a program if its value can be used subsequently, otherwise, it is dead at that point.
- A related idea is dead or useless code, statements that compute values that never get used.
- While the programmer is unlikely to introduce any dead code intentionally, it may appear as the result of previous transformations.
- If we use following statement at various points in the program, the statement is like

If (debug) print .....

- By a data-flow analysis, it may be possible to deduce that each time the program reaches this statement.
- Deducing at compile time that the value of an expression is constant and using the constant instead is known as constant flooding.
- One advantage of copy propagation is that it often turns the copy statement into dead code.
- Consider the basic block which is considered previously.

$x := t_3$

$a[t_2] := t_5$

$a[t_4] := t_3$

goto B_2

- The above basic block is after copy propagation. The dead code elimination removes the assignment to x and transforms into

$a[t_2] := t_5$

$a[t_4] := t_3$

goto B_2.

Q.32. **Write short note on function preserving transformation in code optimization.** ***CT : S-09,12(4M)***

Ans.

- The transformation of a program is called local if it can be performed by looking only at the statements in a basic block, otherwise it is called global.
- Many transformations can be performed at both the local and global levels. Local transformations are usually performed first.

Function preserving transformations :

Common sub-expression elimination, copy propagation, dead-code elimination and constant folding are common examples of such function preserving transformations.

(1) **Common sub-expression elimination :**

- An occurrence of an expression E is called a common sub-expression if E was previously computed and the values of variables in E have not changed since the previous computation.
- We can avoid recomputing the expression if we can use the previously computed value. This can be eliminated by using DAG representation of basic block.
- For example, block B_5 shown in Fig. below recalculates 4 * i and 4 * j.
- The assignment to t_7 and t_{10} have the common subexpression 4 * i and 4 * j, respectively.
- There is elimination of common subexpression using t_6 instead of t_7 and t_8 instead of t_{10}.

B_5	B_5
$t_6 := 4 * i$	$t_6 := 4 * i$
$x := a[t_6]$	$x := a[t_6]$
$t_7 := 4 * i$	$t_8 := 4 * j$
$t_8 := 4 * j$	$t_9 := a[t_8]$
$t_9 := a[t_8]$	$a[t_6] := t_9$
$a[t_7] := t_9$	$a[t_8] := X$
$t_{10} := 4 * j$	goto B_2
$a[t_{10}] := X$	
goto B_2	

Fig. Local common sub-expression elimination.

(2) **Copy propagation :**

- The assignment statements of the form $f := g$ called copy statement, or copies for short.

Consider a basic block

$x := t_3$

$a[t_2] := t_5$

$a[t_4] := x$

goto B_2

This can be further improved by eliminating x using two new transformations.

- The idea behind the copy-propagation transformation is to use g for f, wherever possible after the copy statement $f := g$.
- The assignment $x := t_3$ in block B_5 is a copy.

- Copy propagation applied to block yields.

$x = t_3$

$a[t_2] = t_5$

$a[t_4] = t_3$

goto B_2

(3) Dead code elimination :

- A variable is live at a point in a program if its value can be used subsequently, otherwise, it is dead at that point.
- A related idea is dead or useless code, statements that compute values that never get used.
- While the propogation unlikely to introduce any dead code intentionally, it may appear as the result of previous transformations.
- If we use following statement at various points in the program, the statement is like

If (debug) print .....

- By a data-flow analysis, it may be possible to deduce that each time the program reaches this statement.
- Deducing at compile time that the value of an expression is constant and using the constant instead is known as constant flooding.

Advantage :

It turns the copy statement into dead code.

Q.33. Consider the following code :

```
for (j = 0 ; j < 20 ; j++)
{
    for (k = 0 ; k < = 20, k++)
    {
        x [j] [k] = y[j] [k] + z [j] [k];
    }
}
```

Perform the following loop optimizations :

(i) Elimination of common sub-expression

(ii) Code motion

(iii) Reduction in strength. **CT:W-13(9M)**

Ans. (i) Common sub-expression elimination :

- The offset for each of the arrays is the same. So, we can eliminate that common sub-expression (i.e. compute the value in t_2 only once and use it 3 times).
- C_1 is a constant representing W * N, W is the width of an element and N is the number of columns in each of the arrays

```
for (j = 0 ; j < = 30 ; j ++) {
for (k = 0 ; k < = 30 ; k++) {
t2 = C1 * j + W * k ;
X [t2] = y [t2] + z [t2] ;
}
}
```

(ii) Code Motion :

The expression C_1 * j is constant in the inner loop and can be take out :

```
for (j = 0 ; j < = 30 ; j ++) {
t1 = C1 * j ;
for (k = 0 ; k < = 30 ; k++) {
t2 = t1 * j + W * k ;
X [t2] = y [t2] + z [t2] ;
}
}
```

(iii) Loop reduction in strength :

We can replace multiplications by additions in the inner and outer loop since t_1 and t_2 are induction variables.

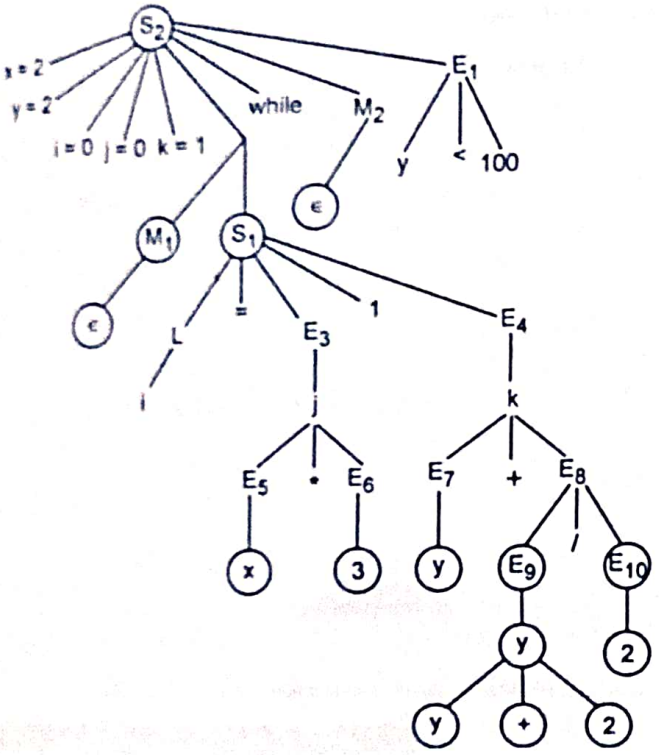
```
t1 = 0 ;
for (j = 0 ; j < = 30 ; j ++) {
t2 = t1 + 0 ;
for (k = 0 ; k < = 30 ; k++) {
x [t2] = y [t2] + z [t2] ;
t2 = t2 + W ;
}
t1 = t1 + C1 ;
}
```

Q.34 Optimize following code using appropriate techniques of code optimization :

```
x = 2; y = 2; i = 0; j = 0; k = 1;
while (y < 100)
{
    i = j / k;
    j = x * 3;
    k = j + y / 2;
    y = y + 2;
}
```

CS : S-14(13M)

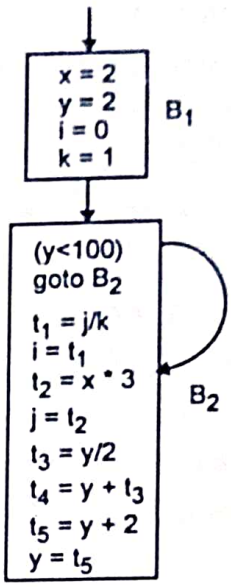
Ans. DAG representation of a code :



TAC is as follows :

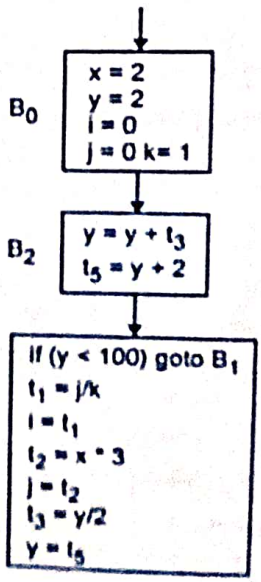
100	x = 2
101	y = 2
102	i = 0
103	j = 0
104	k = 1
105	if (y < 100) goto 117
106	go to 117
107	$t_1 = j/k$
108	$i = t_1$
109	$t_2 = x * 3$
110	$j = t_2$
111	$t_3 = y/2$
112	$t_4 = y + t_3$
113	$t_5 = y + 2$
114	$y = t_5$
117	exit

Basic block computation is as follows :



(i) **Elimination of loop invariant computation :**

- In block B_2, $yt_4 = y + t_3$ and $t_5 = y + 2$ are loop invariant computations.
- In this, moving these loop invariant computation outside the block would not change the meaning of code.



(ii) **Removal of common sub-expression :**

- A common sub-expression is an expression having same or common value.
- For identifying the common sub-expression, we use special type of graph called as directed acyclic graph
- Here there is no common sub-expression so cannot eliminate it.

(iii) **Elimination of induction variable :**

As there is no induction variable present, so can't eliminate it.

Q.35. Write short note on loop invariant computations.

CS : W-14(2M)

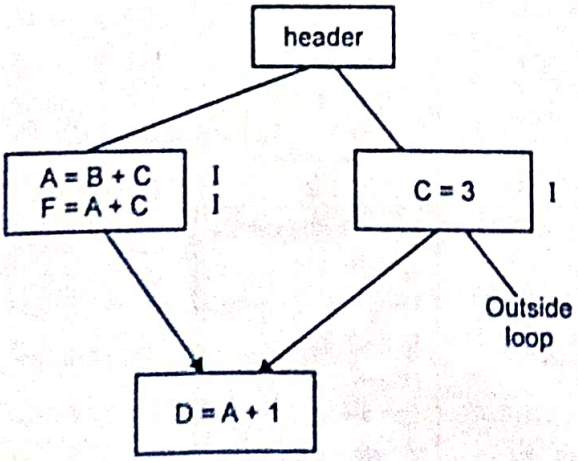
Ans. Loop invariant computations :

- Loop invariant computation means the computations computing a value which is not dependent on how many times loop gets executed.
- Identification of loop invariant computation requires the detection of loops in the program.
- Whether a loop exists in the program or not depends on the programs control flow.
- To eliminate loop invariant computations, we first identify the invariant computations and then move them outside loop if the move does not lead to a change in the program's meaning.

Q.36. Write short note on three conditions of code motion in loop invariant computation. **CT : S-09(4M)**

Ans.

- Loop invariant code consist of statements or expressions which can be moved outside of the body of a loop without affecting the semantic of program.
- Loop invariant computation and code motion (LICM) move a loop invariant statement within a loop to preheader of the loop.



Where, I - Labelled statements are looks like loop invariant.

- Observations are as follows :

(1) **Loop invariants :**

Operands are defined outside loop or are defined by loop invariants

(2) **Code motion :** Not all invariant statements can be moved to the preheader.

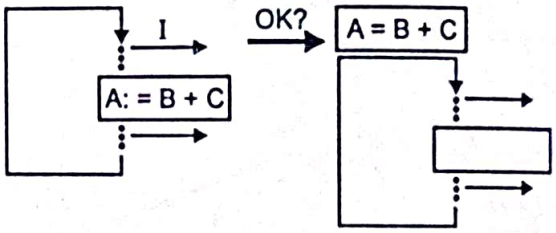
Algorithm is as follows :

- Detect loop invariant computations.
- Check condition for code motion.
- Code transformation conditions of code motion are as follows :

(i) **Correctness :** Movement does not change the program semantics.

(ii) **Performance :** Code should not be slowed down.

Example :



(3) **Need information :**

Control flow of the loop such that dominate all the exists.

Other definition :

No other definition of A exist.

Other uses :

All uses of A are dominated by block b.

Q.37. Write short note on loop jamming and loop unrolling.

CT : S-09,10,(5M), S-11(6M), W-12, S-14(4M)

CS : W-10,12,14(6M), S-11(5M)

Ans. Loop unrolling :

- Loop unrolling involves replicating the body of the loop to reduce the required number of tests if the number of iterations are constant.
- For example, consider following loop.

```
I = 1
Where (I < = 100)
{
x [ I ] = 0
I ++;
}
```

In above case I <= 100 will be performed 100 times.

But if the body of the loop is replicated, then the number of times this test will need to be performed will be 50.

After replication of the body, the loop will be :

```
I = 1
While ( I ≤ 100 )
{
x [ I ] = 0;
I ++;
x [ I ] = 0;
I ++;
}
```

- It is possible to close always any divisor for the number of times the loop is executed and the body will be replicated many times.
- Unrolling once i.e. replicating the body to form two copies of the body it saves 50% of the maximum possible execution.

Loop Jamming :

- Loop jamming is a technique that merges the bodies of two loops if the two loops have the same number of iterations and they use the same indices.
- This eliminates the test of one loop.
- For example, consider the following loop :

```
{
for (I = 0; I < 10; I++)
for (J = 0; J < 10; J++)
X [I, J] = 0;
for (I = 0; I < 10, I ++)
X [X, J] = 1;
}
```

Here, the bodies, of the loops on I can be concatenated. Hence the result of loop jamming will be :

```
{
for (I = 0; I < 10; I ++)
  {
    for (J = 0; J < 10; J ++)
    {
    X [I, J] = 0;
    X [I, J] = 1;
    }
  }
```

- The following conditions are sufficient for making loop jamming legal :

(i) No quantity is computed by the second loop at the iteration I if it is computed by the first loop at iteration J ≥ I .

(ii) If a value is computed by the first loop at iteration J ≥ I then this value should not be used by second loop at iteration I.

Q.38. What do you mean by u-d chaining? What are the applications of u-d chaining? **CT : S-09(4M)**

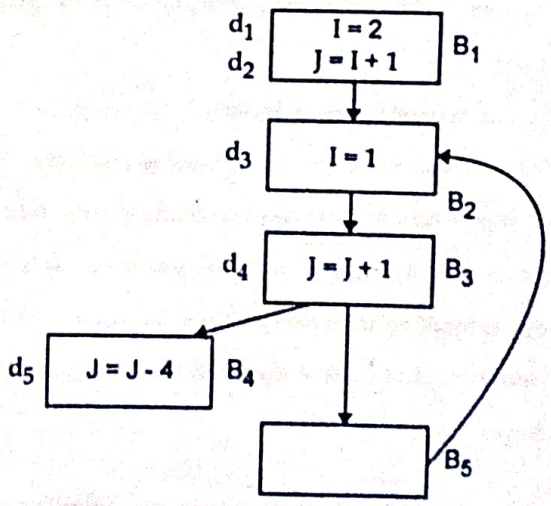
Ans. u-d chaining :

- If the use of A in block B is preceded by its definition, then the u-d chain of A contains only the last definition prior to this use of A.
- If the use of A in block B is not preceded by any definition of A, then the u-d chain for this use consist of all definition of A in IN (B).
- The u-d chain take up much space, it is important for an optimizing compiler to format them completely.

Applications of u-d chains :

- There are variety of uses to which u-d chain information can be put.
- It is used to identify the loop invariant computations.

Example :



The IN and OUT of above flow graph will be

Block	IN	OUT
B_1	ϕ	$\{d_1, d_2\}$
B_2	$\{d_5, d_1, d_2, d_3, d_4\}$	$\{d_2, d_3, d_4, d_5\}$
B_3	$\{d_2, d_3, d_4, d_5\}$	$\{d_3, d_4\}$
B_4	$\{d_3, d_4\}$	$\{d_5, d_3\}$
B_5	$\{d_3, d_4, d_5\}$	$\{d_3, d_4, d_5\}$

- In above the flow graph d_2 uses I, d_4 and d_5 uses J.
- The use of I at d_2 in block B_1 is preceded by definition of I in B_1 i.e. d_1. Hence ud-chain for I in d_2 consist $\{d_1\}$.
- Similarly use of J at d_4 in block B_3 is not preceded by the definition of J in B_3.
- Thus, we have to consider IN $[B_3]$ i.e. $[d_2, d_3, d_4, d_5]$.
- Out of these except d_3, all contains definition of J. Thus tbe ud chain for J in d_4 is $[d_2, d_4, d_5]$.
- The use of J at d_5 in block B_4 is not preceded by definition of J, so consider IN$[B_4] = [d_3, d_4]$
- Of these d_4 contains definition of j. Hence ud-chain for J and d_5 consists in $[d_4]$.
- To obtain such a graph, we must partition the intermediate code into basic blocks.
- This requires identifying leader statements which are defined as follows :

(1) The first statement is a leader statement.

(2) The target of a conditional or unconditional goto is a leader.

(3) A statement immediately follows a conditional goto is a leader.

- A basic block is a sequence of TAC statements that can be entered only at beginning and control ends after the execution of the last statement, without a halt or any possibility of branching, except at the end.

Q.39. Calculate u-d chain for the statement "a = a * i" in following code :

(i) a = 1

(ii) i = 2

(iii) if i > = 10 goto (8)

(iv) a = a * i

(v) $t_1 = i + 1$

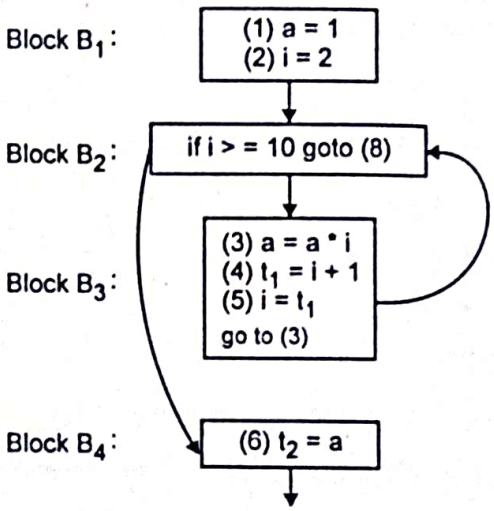
(vi) $i = t_1$

(vii) goto (3)

(viii) $t_2 = a$

CS : S-09(14M)

Ans. **Step (1) : The program flow graph will be :**



Step (2) : To find GEN and KILL code :

Block	GEN	KILL
B_1	{1, 2}	{3, 5}
B_2	{ϕ}	{ϕ}
B_3	{3, 4, 5}	{1, 3}
B_4	{6}	{ϕ}

Step (3) : To find predecessor :

Block	Predecessor
B_1	ϕ
B_2	B_1, B_4
B_3	B_2
B_4	B_2, B_3
B_5	B_4

VBD

Step (4) : To find iteration

Iteration (1) :

Block	IN	OUT
B_1	$\{\phi\}$	{1, 2}
B_2	$\{\phi\}$	$\{\phi\}$
B_3	$\{\phi\}$	{3, 4, 5}
B_4	$\{\phi\}$	{6}

Iteration (2) :

Block	IN	OUT
B_1	$\{\phi\}$	{1, 2}
B_2	{1, 2, 3, 4, 5}	{1, 2, 3, 4, 5}
B_3	$\{\phi\}$	{3, 4, 5}
B_4	$\{\phi\}$	{6}

Iteration (3) :

Block	IN	OUT
B_1	$\{\phi\}$	{1, 2}
B_2	{1, 2, 3, 4, 5}	{1, 2, 3, 4, 5}
B_3	{1, 2, 3, 4, 5}	{3, 4, 5}
B_4	{1, 2, 3, 4, 5}	{1, 2, 3, 4, 5}

Iteration (4) :

Block	IN	OUT
B_1	$\{\phi\}$	{1, 2}
B_2	{1, 2, 3, 4, 5}	{1, 2, 3, 4, 5}
B_3	{1, 2, 3, 4, 5}	{3, 4, 5}
B_4	{1, 2, 3, 4, 5}	{1, 2, 3, 4, 5, 6}

∴ OUT of iteration III and IV are same.

The use of a = a * i at 3 in B_3 is not preceded by definition of a and i. Thus we consider IN (B_3), which is from iteration IV, equal to {1, 2, 3, 4, 5} of these all for 4 are definition of a and I, so u-d chain for a = a * i at 3 is 1, 2, 3, 4, 5.

Q.40. What is basic block? Explain with suitable example. Also define the GEN and KILL set for a basic block. CS : W-10, S-12(7M)

Ans. Basic block :

- It is the sequence of three address statements that is entered in the beginning and control ends after the execution of last statement without any halt or any branching, except at the end.
- Some of the loop invariant computations are as follows :

(a) If there is an expression inside the loop such as T = constant or T = C where C = constant the expression is loop invariant.

(b) A variable initialized to a variable which is not reinitialized inside the loop.

- The set-of all the definitions capable of reaching to a point immediately after the last statement of block B will be called as OUT (B).
- We compute both IN (B) and OUT (B) for every block B, GEN (B) and KILL (B) are defined as follows :

GEN (B) = The set-of all the definitions generated in block B.

KELL (B) : The set of all definitions outside block B that define the same variable as are defined in block B.

- IN (B) and OUT (B) are defined by the following set of equations, which are called "data flow equations".

IN (B) = OUT (p)

OUT (B) = IN (B) – KILL (B) ∪ GEN (B)

Algorithm :

For computing smallest IN (B) and OUT (B) is as follows :

(1) For each block B do

{

IN (B) = ϕ

OUT (B) = GEN (B)

}

(2) flag = true.

(3) while (flag) do

{

flag = false

for each block B do

{

IN_{new} (B) = ϕ

for each predecessor P of B

$IN_{new}(B) = IN_{new}(B) \cup OUT(P)$

if $IN_{new}(B) \neq IN(B)$ then

{

flag = true

$IN(B) = IN_{new}(B)$

OUT (B) = IN (B) – KILL (B) ∪

GEN (B)

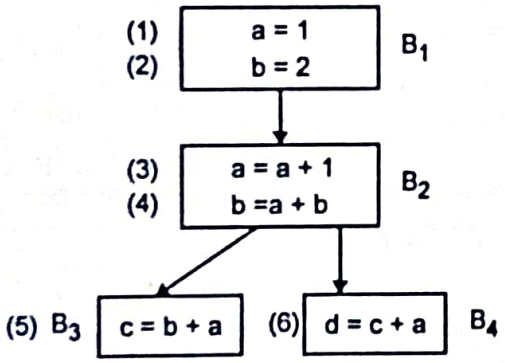
}

}

}

- Initially we take IN (B) for every block that is to be an empty set and we take OUT (B) to be GEN (B), and we compute $IN_{new}(B)$.
- If it is different from IN (B), we compute a new OUT (B) and go for the next iteration.
- This is continued until IN (B) comes out to be the same for every B in a previous and current iteration.
- For example, Consider following flow graph.

Step (1) :



Step (2) : Compute GEN and KILL :

Block	GEN	KILL
B_1	{1, 2}	{3, 4}
B_2	{3, 4}	{1, 2}
B_3	{5}	{ϕ}
B_4	{6}	{ϕ}

Step (3) : To find predecessor :

Block	Predecessor
B_1	ϕ
B_2	B_1
B_3	B_1
B_4	B_3, B_2

Step (4) :

Compute IN and OUT in iteration until IN and OUT are same :

Iteration (1) :

Block	IN	OUT
B_1	ϕ	{1, 2}
B_2	ϕ	{3, 4}
B_3	ϕ	{5}
B_4	ϕ	{6}

Iteration (2) :

Block	IN	OUT
B_1	{ϕ}	{1, 2}
B_2	{1, 2}	{3, 4}
B_3	{3, 4}	{3, 4, 5}
B_4	{3, 4, 5}	{3, 4, 5, 6}

Iteration (3) :

Block	IN	OUT
B_1	ϕ	{1, 2}
B_2	{1, 2}	{3, 4}
B_3	{3, 4}	{3, 4, 5}
B_4	{3, 4, 5}	{3, 4, 5, 6}

As OUT of iteration (2) and (3) are same. So stop iteration.

CONTROL FLOW ANALYSIS

Q.41. Explain control flow analysis in code optimization.

Ans. Control flow analysis :

- In control-flow analysis, the compiler figures out even more information about how the program does its work.
- Only now it can assume that there are no syntactic or semantic errors in the code.

Example : Following is the code for calculation of fibonacci series :

```
int fib (int base)
{
    int result;
    if (base < = 1)
      {
          result = base;
    } else
      {
          int i;
          int f0;
          int f1;
          f0 = 0;
          f1 = 1;
          i = 2;
              while (i < = base)
          {
              result = f0 + f1;
              f0 = f1;
              f1 = result;
              i = i + 1;
          }
      }
return result;
}
```

- The TAC code for above code is as follows :

```
fib :
    BeginFunc 68;
    TMP0 = 1;
    TMP 1 = base < TMP0;
    TMP 2 = base = = TMP0;
    TMP 3 = TMP 1 || TMP 2;
    IfZ TMP3 Goto_L0;
    result = base;
    Goto L 1;
L0 :
    TMP4 = 0;
    f0 = TMP4;
    TMP5 = 1;
    f1 = TMP5;
    TMP6 = 2;
    i = TMP6;
L2 :
    TMP7 = i < base;
    TMP8 = i = = base;
    TMP9 = TMP7 || TMP8;
    IfZ TMP9 Goto_L3;
    TMP 10 = f0 + f1;
    result = TMP 10;
    f0 = f1;
    f1 = result;
    TMP11 = 1;
    TMP 12 = i + TMP 11;
    i = TMP 12;
    Goto L2;
L3 :
L1 :
    Return result;
    End Func;
```

- Control-flow analysis begins by constructing a control-flow graph, which is a graph of the different possible paths program flow could take through a function. To build the graph, first divide the code into basic blocks.
- A basic block is a segment of the code that a program must enter at the beginning and exit only at the end.
- This means that only the first statement can be reached from outside the block (there are no branches into the middle of the block) and all

statements are executed consecutively after the first one is (no branches or halts until the exit). Thus a basic block has exactly one entry point and one exit point.

- If a program executes the first instruction in a basic block, it must execute every instruction in the block sequentially after it.
- A basic block begins in one of several ways :
  (i) The entry point into the function
  (ii) The target of a branch (in our example, any label)
  (iii) The instruction immediately following a branch or a return.
- A basic block ends in any of the following ways :
  (i) A jump statement
  (ii) A conditional or unconditional branch
  (iii) A return statement.

Divide the Fibonacci TAC code into basic blocks :

- Now, we can construct the control-flow graph between the blocks.
- Each basic block is a node in the graph, and the possible different routes a program might take are the connections, i.e. if a block ends with a branch, there will be a path leading from that block to the branch target. The blocks that can follow a block are called its successors.
- There may be multiple successors or just one. Similarly the block may have many, one, or no predecessors.

DATA FLOW ANALYSIS

Q.42. Explain data flow analysis in detail. **CT : W-12(2M)**

Ans. Data flow analysis :

- Data-flow analysis is a technique for gathering information about the possible set of values calculated at various points in a computer program. A programs Control Flow Graph (CFG) is used to determine those parts of a program to which a particular value assigned to a variable might propagate.
- In global optimizations, optimizations can be performed across basic block.
- Data-flow analysis is much more complicated than control-flow analysis.
- Let's consider a global common subexpression elimination optimization as our example.
- Detail analysis across blocks can determine whether an expression is alive on entry to a block. Such an expression is said to be available at that point.
- Once the set of available expressions is known, common sub-expressions can be eliminated on a global basis.
- Each block is a node in the flow graph of a program.
- The successor set (succ (x)) for a node x is the set of all nodes that x directly flows into.
- The predecessor set (pred (x)) for a node x is the set of all nodes that flow directly into x.
- An expression is defined at the point where it is assigned a value and killed when one of its operands is subsequently assigned a new value.
- An expression is available at some point p in a flow graph if every path leading to p contains a prior definition of that expression which is not subsequently killed.

  avail[B] = set of expressions available on entry to block B.

  exit[B] = set of expressions available on exit from B.

  avail [B] = $\cap$ exit [x]; x $\in$ pred[B] (i.e. B has available the intersection of the exit of its predecessors).

  killed[B] = set of expressions killed in B

  defined[B] = set of expressions defined in B

  exit [B] = avail[B] – killed [B] + defined [B]

  avail[B] = $\cap$ (avail [x] – killed [x] + defined [x]) : x $\in$ pred[B]
- Here is an algorithm for global common sub-expression elimination :

(1) First, compute defined and killed sets for each basic block (this does not involve any of its predecessors or successors).

(2) Iteratively compute the avail and exit sets for each block by running the following algorithm until we hit a stable fixed point :

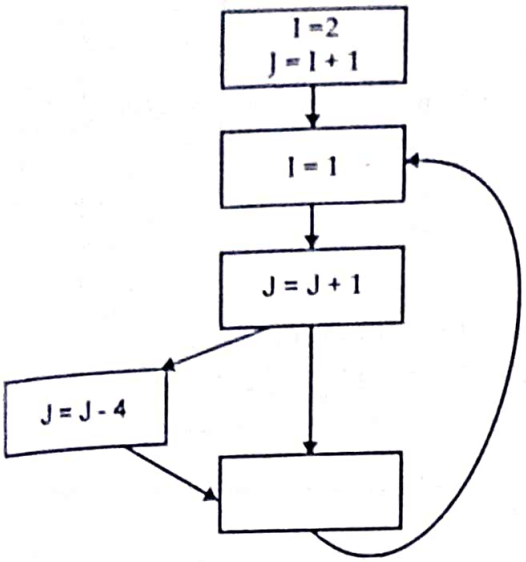
(a) Identify each statements of the form a = b op c in some block B such that b op c is available at the entry to B and neither b nor c is redefined in B.

(b) Follow flow of control backward in the graph passing back to but not through each block that defines b op c. The last computation of b op c in such a block reaches s.

(c) After each computation d = b op c identified in step 2a, add statement t = d to that block where t is a new temp.

(d) Replace s by a = t.

Q 43. Consider the flow graph :

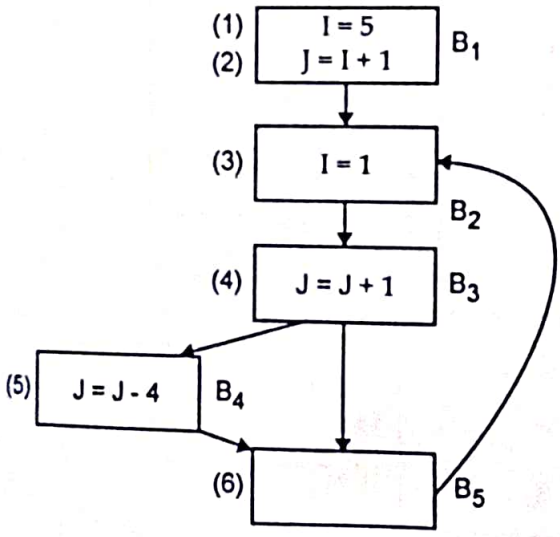


(i) Compute IN and OUT of each block

(ii) Compute u-d chain for "J = J + 1".

CT : S-09(9M), S-12(10M)

Ans. Step (1) : Basic blocks with partitioned code are as follows :



Step (2) : To find GEN and KILL code :

Block	GEN	KILL
B_1	{1, 2}	{3, 4, 5}
B_2	{3}	{1}
B_3	{4}	{2, 5}
B_4	{5}	{2, 4}
B_5	ϕ	ϕ

Step (3) : To find predecessor :

Block	Predecessor
B_1	ϕ
B_2	B_1 , B_5
B_3	B_2
B_4	B_3
B_5	B_3, B_4

Step (4) : To find Iteration :

Iteration (1) :

Block	IN	OUT
B_1	ϕ	{1, 2}
B_2	ϕ	{3}
B_3	ϕ	{4}
B_4	ϕ	{5}
B_5	ϕ	ϕ

Iteration (2) :

Block	IN	OUT
B_1	{3}	{1, 2}
B_2	{1, 2}	{2, 3}
B_3	{3}	{3, 4}
B_4	{4}	{5}
B_5	{4, 5}	{4, 5}

Iteration (3) :

Block	IN	OUT
B_1	{2, 3}	{1, 2}
B_2	{1, 2, 4, 5}	{2, 3, 4, 5}
B_3	{2, 3}	{3, 4}
B_4	{3, 4}	{3, 5}
B_5	{3, 4, 5}	{3, 4, 5}

Iteration (4) :

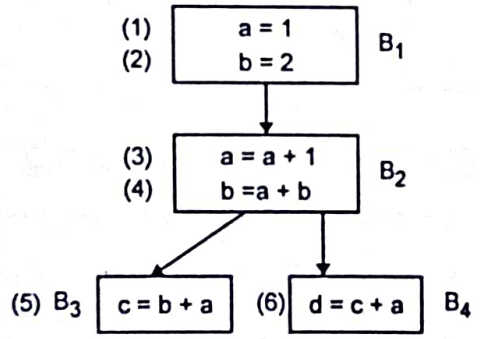
Block	IN	OUT
B_1	{2, 4, 5, 3}	{1, 2}
B_2	{1, 2, 3, 4, 5}	{2, 3, 4, 5}
B_3	{2, 3, 4, 5}	{3, 4}
B_4	{3, 4}	{3, 5}
B_5	{3, 4, 5}	{3, 4, 5}

Iteration (5) :

Block	IN	OUT
B_1	{2, 4, 5, 3}	{1, 2}
B_2	{1, 2, 3, 4, 5}	{2, 3, 4, 5}
B_3	{2, 3, 4, 5}	{3, 4}
B_4	{3, 4}	{3, 5}
B_5	{3, 4, 5}	{3, 4, 5}

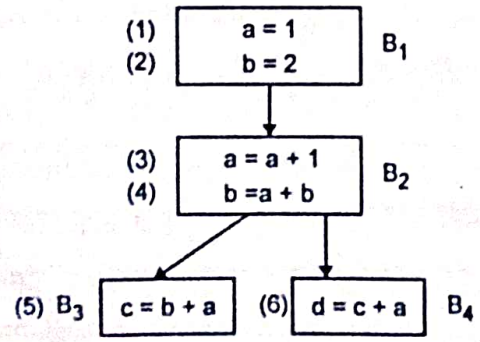
OUT of iteration (5) and (4) is same, so the values of IN and OUT are final.

Q.44. Consider following flow graph.



Compute IN and OUT of each basic block. **CS : W-10(14M)**

Ans. Step (1) :



Step (2) : To find GEN and KILL :

Block	GEN	KILL
B_1	{1, 2}	{3, 4}
B_2	{3, 4}	{1, 2}
B_3	{5}	{ϕ}
B_4	{6}	{ϕ}

Step (3) : To find predecessor :

Block	Predecessor
B_1	ϕ
B_2	B_1
B_3	B_2
B_4	B_3, B_2

Step (4) : To find Iteration :

Iteration (1) :

Block	IN	OUT
B_1	ϕ	{1, 2}
B_2	ϕ	{3, 4}
B_3	ϕ	{5}
B_4	ϕ	{6}

Iteration (2) :

Block	IN	OUT
B_1	{ϕ}	{1, 2}
B_2	{1, 2}	{3, 4}
B_3	{3, 4}	{3, 4, 5}
B_4	{3, 4, 5}	{3, 4, 5, 6}

Iteration (3) :

Block	IN	OUT
B_1	{ϕ}	{1, 2}
B_2	{1, 2}	{3, 4}
B_3	{3, 4}	{3, 4, 5}
B_4	{3, 4, 5}	{3, 4, 5}

OUT of iteration (2) and (3) are same, so the value of IN and OUT are find.

VRD

Q45. Use data flow equations to compute IN and OUT for following

TAC :

```
i = m - 1
j = n
a = m
do
    i = i + 1
    j = J - 1
if c1 then
    a = j
    else i = m
while c2
```

CS : S-10(13M)

OR Construct flow graph for following program and find IN and OUT for each block.

```
i = m - 1
j = n
a = u1
do
    i = i + 1
    j = j - 1
if c1 then
    a = u2
else
    i = u3
while c2
```

find for each block :

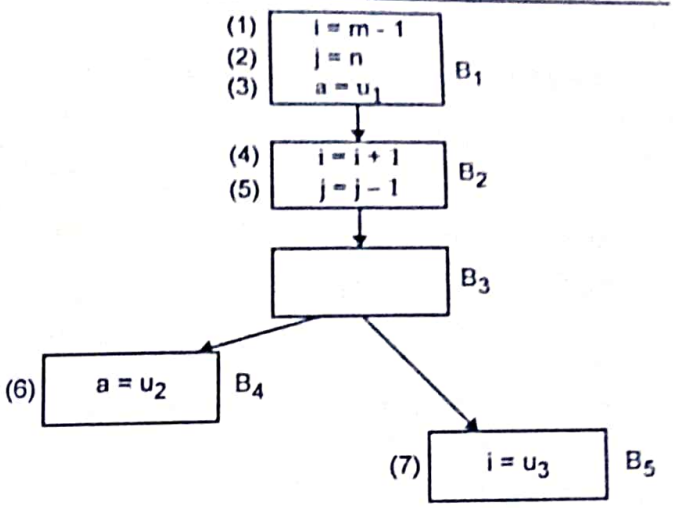
(1) IN and OUT

(2) Ud chain and du-chain.

CT : W-10(10M), W-12,13(13M)

Ans. Step (1) :

The basic blocks with partitioned code are as follows :



Step (2) : To find GEN and KILL :

Block	GEN	KILL
B_1	{1, 2, 3}	{4, 5, 6, 7}
B_2	{4, 5}	{1, 2, 7, 3}
B_3	{ϕ}	{ϕ}
B_4	{6}	{3}
B_5	{7}	{1, 4}

Step (3) : To find predecessor :

Block	Predecessor
B_1	ϕ
B_2	B_1
B_3	B_2
B_4	B_3
B_5	B_3

Step (4) : To find Iteration :

Iteration (1) :

Block	IN	OUT
B_1	{ϕ}	{1, 2, 3}
B_2	{ϕ}	{4, 5}
B_3	{ϕ}	{ϕ}
B_4	{ϕ}	{6}
B_5	{ϕ}	{7}

Iteration (2) :

Block	IN	OUT
B_1	{ϕ}	{1, 2, 3}
B_2	{1, 2, 3}	{3, 4, 5}
B_3	{4, 5}	{ϕ}
B_4	{ϕ}	{6}
B_5	{ϕ}	{7}

Iteration (3) :

Block	IN	OUT
B_1	{ϕ}	{1, 2, 3}
B_2	{1, 2, 3}	{3, 4, 5}
B_3	{3, 4, 5}	{3, 4 , 5}
B_4	{4, 5}	{4, 5, 6}
B_5	{4, 5, 3}	{5, 7, 3}

Iteration (4) :

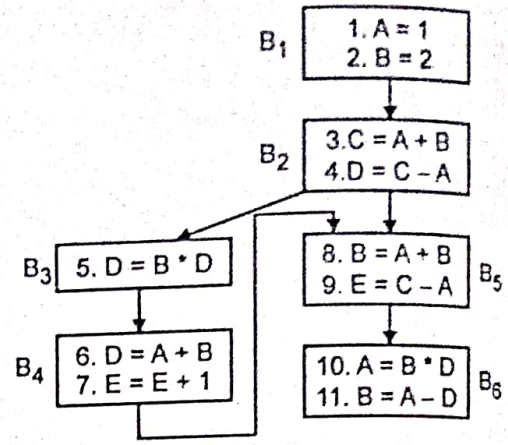
Block	IN	OUT
B_1	{ϕ}	{1, 2, 3}
B_2	{1, 2, 3}	{3, 4, 5}
B_3	{3, 4, 5}	{3, 4, 5}
B_4	{3, 4, 5}	{4, 5, 6}
B_5	{3, 4, 5}	{3, 5, 7}

Iteration (5) :

Block	IN	OUT
B_1	{ϕ}	{1, 2, 3}
B_2	{1, 2, 3}	{3, 4, 5}
B_3	{3, 4, 5}	{3, 4, 5}
B_4	{3, 4, 5}	{4, 5, 6}
B_5	{3, 4, 5}	{3, 5, 7}

Out of iteration (4) and (5) are same so the value of IN and OUT are final.

Q.46. Find IN and OUT of data flow equation for the program graph given below : CS : S-11(10M), S-12(13M), W-14(10



Ans. Step (1) : To find GEN and KILL :

Block	GEN	KILL
B_1	{1, 2}	{6, 10, 11}
B_2	{3, 4}	{5, 8}
B_3	{5}	{4, 8}
B_4	{6, 7}	{2, 9, 11}
B_5	{8, 9}	{4, 5, 7}
B_6	{10, 11}	{1, 2, 6}

Step (2) : Compute iteration :

Iteration (1) :

Block	IN	OUT
B_1	ϕ	{1, 2}
B_2	ϕ	{3, 4}
B_3	ϕ	{5}
B_4	ϕ	{6, 7}
B_5	ϕ	{8, 9}
B_6	ϕ	{10, 11}

Iteration (2) :

Block	IN	OUT
B_1	{ϕ}	{1, 2}
B_2	{1, 2, 6, 7}	{1, 2, 3, 4, 6, 7}

B_3	{3, 4, 8, 9}	{3, 5, 9}
B_4	{5}	{5, 6, 7}
B_5	{5}	{8, 9}
B_6	{6, 7}	{7, 10, 11}

Iteration (3) :

Block	IN	OUT
B_1	{ϕ}	{1, 2}
B_2	{1, 2, 5, 6, 7}	{1, 2, 3, 4, 6, 7}
B_3	{1, 2, 3, 4, 6, 7, 8, 9}	{1, 2, 3, 5, 6, 7, 9}
B_4	{3, 5, 9}	{3, 5, 6, 7}
B_5	{3, 5, 9}	{3, 8, 9}
B_6	{5, 6, 7}	{5, 7, 10, 11}

Iteration (4) :

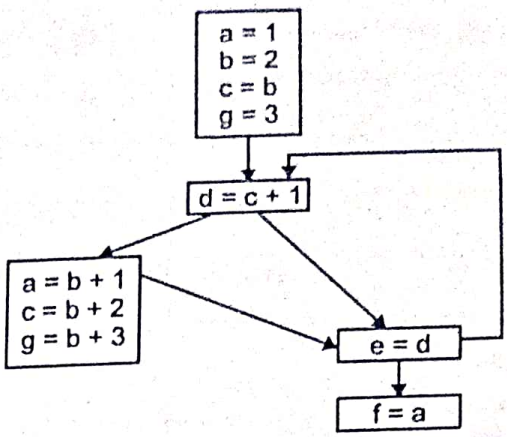
Block	IN	OUT
B_1	{ϕ}	{1, 2}
B_2	{1, 2, 3, 5, 6, 7}	{1, 2, 3, 4, 6, 7}
B_3	{1, 2, 3, 4, 6, 7, 8, 9}	{1, 2, 3, 5, 6, 7, 9}
B_4	{1, 2, 3, 5, 6, 7, 9}	{1, 3, 5, 6, 7}
B_5	{1, 2, 3, 5, 6, 7, 9}	{1, 2, 3, 6, 8, 9}
B_6	{3, 5, 6, 7}	{3, 5, 7, 10, 11}

Iteration (5) :

Block	IN	OUT
B_1	{ϕ}	{1, 2}
B_2	{1, 2, 3, 5, 6, 7}	{1, 2, 3, 4, 6, 7}
B_3	{1, 2, 3, 4, 6, 7, 8, 9}	{1, 2, 3, 5, 6, 7, 9}
B_4	{1, 2, 3, 5, 6, 7, 9}	{1, 3, 5, 6, 7}
B_5	{1, 2, 3, 5, 6, 7, 9}	{1, 2, 3, 6, 8, 9}
B_6	{1, 3, 5, 6, 7}	{3, 5, 7, 10, 11}

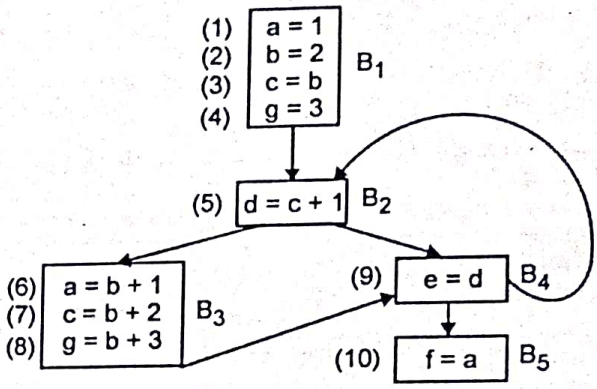
OUT of iteration (4) and (5) are same, so the value of IN and OUT are final.

Q.47. Consider following flow graph. Remove all loop invariant computation. CS : S-13(13M)



Ans.

Step (1) :



Step (2) : To find GEN and KILL :

Block	GEN	KILL
B_1	{1, 2}	{3, 5}
B_2	{ϕ}	{ϕ}
B_3	{3, 4, 5}	{1, 3}
B_4	{6}	{ϕ}

Step (3) : To find Predecessor :

Block	Predecessor
B_1	ϕ
B_2	B_1, B_2
B_3	B_2
B_4	B_2, B_3
B_5	B_4

Step (4) : To find Iteration :

Iteration (1) :

Block	IN	OUT
B_1	{ϕ}	{1, 2}
B_2	{ϕ}	{ϕ}
B_3	{ϕ}	{3, 4, 5}
B_4	{ϕ}	{6}

Iteration (2) :

Block	IN	OUT
B_1	{ϕ}	{1, 2}
B_2	{1, 2, 3, 4, 5}	{1, 2, 3, 4, 5}
B_3	{ϕ}	{3, 4, 5}
B_4	{ϕ}	{6}

Iteration (3) :

Block	IN	OUT
B_1	{ϕ}	{1, 2}
B_2	{1, 2, 3, 4, 5}	{1, 2, 3, 4, 5}
B_3	{1, 2, 3, 4, 5}	{3, 4, 5}
B_4	{1, 2, 3, 4, 5}	{1, 2, 3, 4, 5}

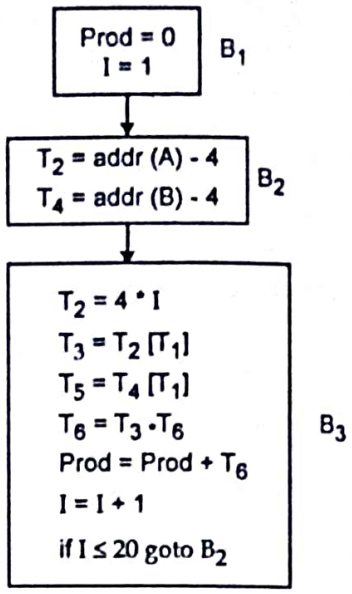
Iteration (4) :

Block	IN	OUT
B_1	{ϕ}	{1, 2}
B_2	{1, 2, 3, 4, 5}	{1, 2, 3, 4, 5}
B_3	{1, 2, 3, 4, 5}	{3, 4, 5}
B_4	{1, 2, 3, 4, 5}	{1, 2, 3, 4, 5}

So OUT of iteration (3) and (4) are same.

∴ Value of IN and OUT are final.

Q.48. Consider the flow graph and eliminate the induction variables. Show result of each step. **CT : W-10(13M), CS : S-13(13M)**



OR Write three-address-code statements equivalent to following source code. Apply different code optimization techniques and get equivalent optimized code.

```
PROD : = 0;
    I : = 1;
do
    begin
    PROD : = PROD + A[I] * B[I];
        I : = I + 1
    end
    while I ≤ 20
    end
```

CT : W-09, S-12(13M), S-11(6M)

Ans. TAC is as follows :

(1) PROD = 0

(2) I = 1

(3) $T_1 = I * 4$

(4) T_2 = addr (A) – 4

(5) $T_3 = T_2 [T_1]$

(6) T_4 = addr (B) – 4

(7) $T_5 = T_4 [T_1]$

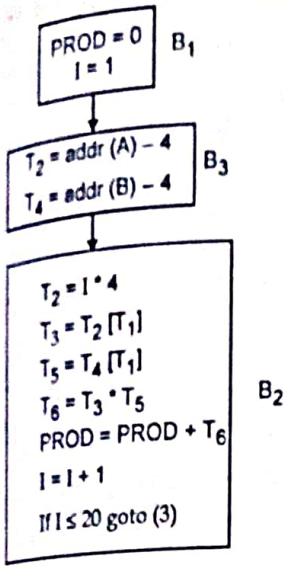
(8) $T_1 = T_3 * T_5$

(9) PROD = PROD + T_6

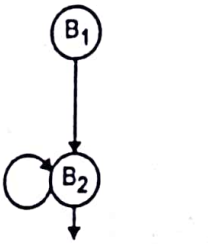
(10) I = I + 1

(11) If I ≤ 20 goto (3)

Basic blocks :



Flow graph :

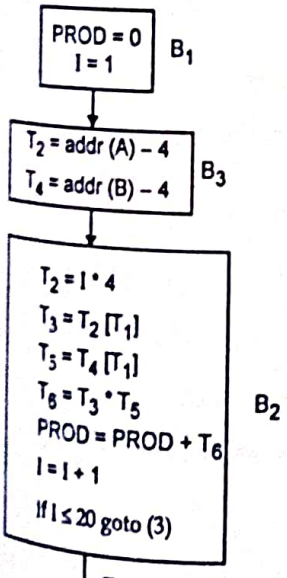


To the block begining with statement following (11)

Code motion :

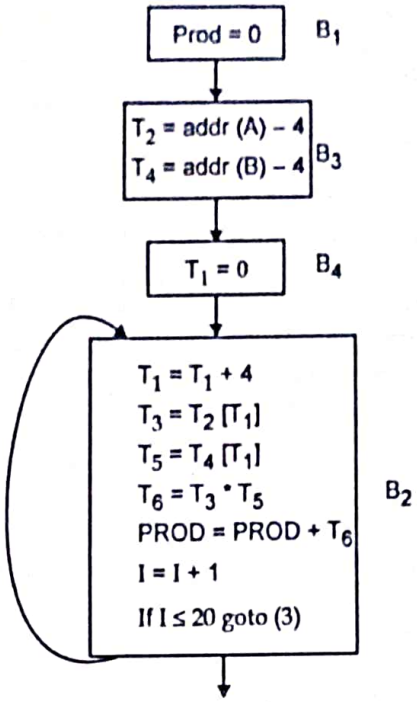
Here T_2 = addr (A) – 4

T_4 = addr (B) – 4 are loop invariant computation, hence can be placed outside the loop.



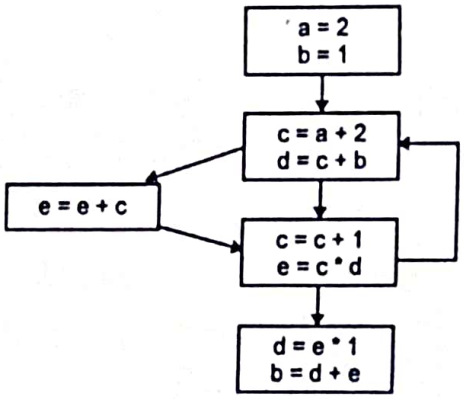
To the block begining with statement following (11)

Induction variable elimination :



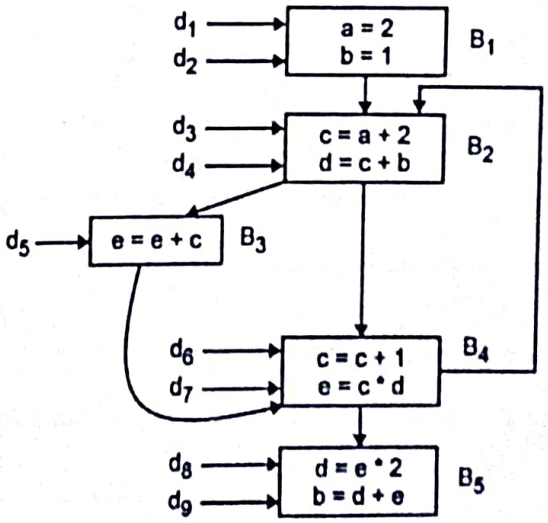
To the block begining with statement following (11)

Q.49. Consider flow graph as given. Compute the reaching definitions for each basic block. **CT : S-10(9M)**



Ans.

Step (1) : The basic blocks with partitioned code is as follows :



VBD

Step (2) : To find GEN and KILL :

Block	GEN	KILL
B_1	{1, 2}	{9}
B_2	{3, 4}	{6, 8}
B_3	{5}	{7}
B_4	{6, 7}	{3, 5}
B_5	{8, 9}	{2, 4}

Step (2) : To find iteration :

Iteration (1) :

Initially we have IN [B] = ϕ and OUT [B] = GEN [B] therefore we get

Block	IN	OUT
B_1	ϕ	{1, 2}
B_2	ϕ	{3, 4}
B_3	ϕ	{5}
B_4	ϕ	{6, 7}
B_5	ϕ	{8, 9}

Iteration (2) :

Block	IN	OUT
B_1	ϕ	{1, 2}
B_2	{1, 2, 6, 7}	{1, 2, 3, 4, 7}
B_3	{1, 2, 3, 4, 7}	{1, 2, 3, 4, 5}
B_4	{1, 2, 3, 4, 5, 7}	{1, 2, 4, 6, 7}
B_5	{1, 2, 4, 6, 7}	{1, 6, 7, 8, 9}

Iteration (3) :

Block	IN [B]	OUT [B]
B_1	ϕ	{1, 2}
B_2	{1, 2, 4, 6, 7}	{1, 2, 3, 4, 7}
B_3	{1, 2, 3, 4, 7}	{1, 2, 3, 4, 5}
B_4	{1, 2, 3, 4, 5, 7}	{1, 2, 4, 6, 7}
B_5	{1, 2, 4, 6, 7}	{1, 6, 7, 8, 9}

Iteration (4) :

Block	IN [B]	OUT [B]
B_1	ϕ	{1, 2}
B_2	{1, 2, 4, 6, 7}	{1, 2, 3, 4, 7}
B_3	{1, 2, 3, 4, 7}	{1, 2, 3, 4, 5}
B_4	{1, 2, 3, 4, 5, 7}	{1, 2, 4, 6, 7}
B_5	{1, 2, 4, 6, 7}	{1, 6, 7, 8, 9}

For 3rd and 4th iteration values of IN [B] & OUT [B] for each block is same hence we will stop.

$\therefore$ Reaching definition for each block means IN of that block

$\therefore$ i.e.

$IN[B_1] = \phi$, $IN[B_2] = \{1, 2, 4, 6, 7\}$

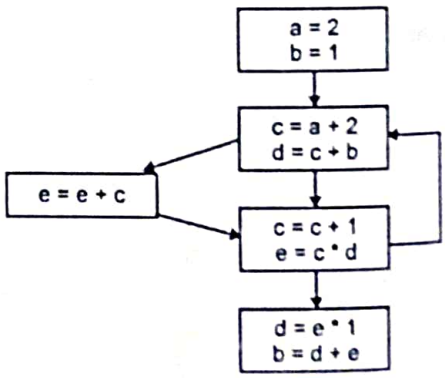
$IN[B_3] = \{1, 2, 3, 4, 7\}$,

$IN[B_4] = \{1, 2, 3, 4, 5, 7\}$

and $IN[B_5] = \{1, 2, 4, 6, 7\}$

Q.50. Explain the significance of computing U-d chain information. Give U-d chain information for each block of Fig.

CT-S-10&10



Ans. Ud_chain :

- If a use of a variable A is preceded in its block by a definition of A, then only the last definition of A in the block prior to this use reaches the use.
- Thus the list or ud_chain for this use consists of only this one definition.

If use of A is preceded in its block B by no definition of A, then the ud chain for this use consists of all definitions of A in IN[B]

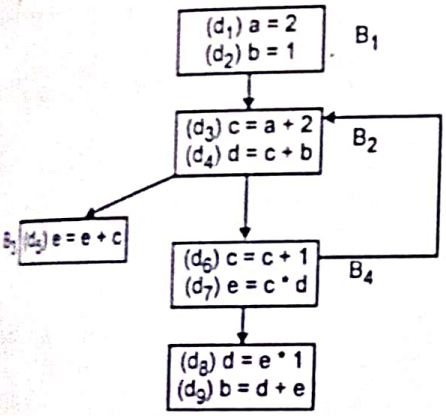
use definition of J is defined as d_5 in Block B_4. The use of J at d_5 of block B_4 is not preceded by a definition of J. So we consider IN[B_4] = { d_1, d_2 }

Of these, only d_4 defines J, so the ud_chain for J in d_5 of Block B_4 consists of d_4 only

ud chain information used to identify the loop invariant computations.

If there are any loop invariant computations inside any block of a loop ; such computations can be moved out side of a block in the form of preheader. This process is called as code motion.

The blocks are as follows :



Block B_1 : ud chain for block $B_1 = \phi$

Block B_2 :

For definition d_3 : c = a + 2 , a is not defined in same block

∴ ud chain for d_3 will be computed using IN $[B_2] = \{d_1, d_2, d_4, d_5, d_7\}$ out of these definitions, the only definition for a is d_1

∴ ud chain for block B_2 for definition d_3 : c = a + 2 is d_1

For definition d_4 : d = c + b

b is defined in B_2 but b is not defined in same block

∴ ud chain for block $d_4 \Rightarrow$ s

$= d_1 \cup$ definition for b in IN B_2

$= d_1 \cup d_2$

ud chain for $d_4 = \{d_2, d_3\}$

Block B_3 :

For definition d_5 : e = e + c

Hence e and c are not defined in same block

∴ IN[B_3] is considered.

$IN[B_3] = \{d_1, d_2, d_3, d_4, d_7\}$

Out of these definitions d_3 and d_7 are related with e and c

∴ ud chain : $\{d_3, d_7\}$

Block B_4 :

For definition d_6 : ud chain for d_6

$= IN[B_4] = \{d_1, d_2, d_3, d_4, d_5, d_7\}$

Out of these definitions; definition of c $\Rightarrow d_3$

∴ ud chain for definition $d_6 = \{d_3\}$

For definition d_7 : e = c * d

Definition of c in this definition is preceded by its use $\Rightarrow d_6$ and for d we have to use

$IN[B_4] = \{d_1, d_2, d_3, d_4, d_5, d_7\} \Rightarrow d_4$

∴ ud chain $\Rightarrow \{d_4, d_6\}$

Block B_5 :

For definition d_8 : e is used but not defined in same block.

∴ $IN[B_5] = \{d_1\ d_2\ d_4\ d_6\ d_7\} = \{d_7\}$

For definition d_9 : b = d + e use of d is preceded by its definition

∴ for d $\Rightarrow d_8$

Use of e is not preceded by its definition in same block

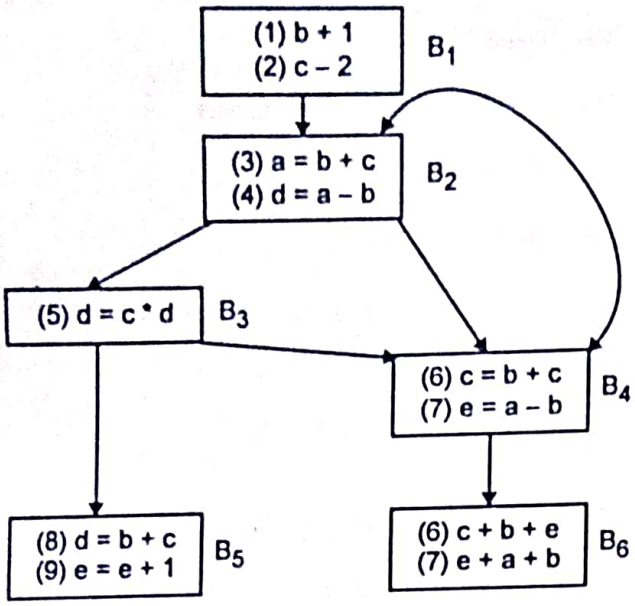
∴ IN[B_5] is used $\Rightarrow \{d_1\ d_2\ d_4\ d_6\ d_7\}$

Out of these definitions ; definition for e is d_7

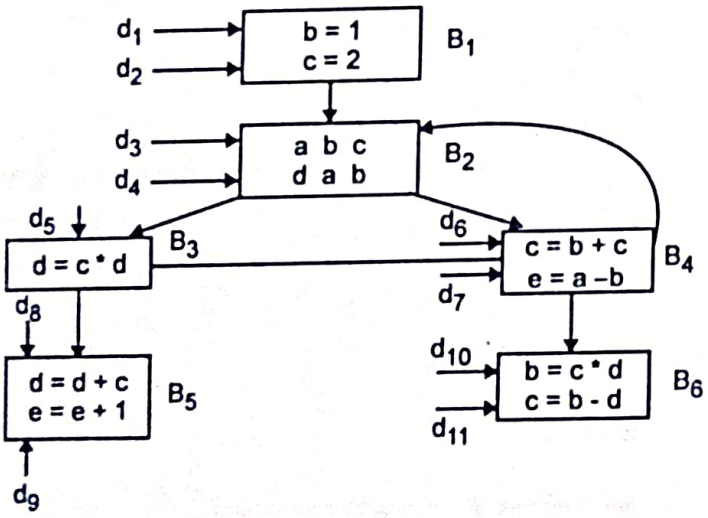
∴ Complete ud chain for definition

$d_9 \Rightarrow d_7, d_8$

Q.51. Compute IN (B) and OUT (B) for the following Blocks and find out no of iteration. **CT : S-13(13M)**



Ans.



Step (1) : To find GEN and KILL :

BLOCK	GEN	KILL
B_1	$\{d_1, d_2\}$	$\{d_6, d_{10}, d_{11}\}$
B_2	$\{d_3, d_4\}$	$\{d_5, d_8\}$
B_3	$\{d_5\}$	$\{d_4, d_8\}$
B_4	$\{d_6, d_7\}$	$\{d_4, d_9, d_{11}\}$
B_5	$\{d_8, d_9\}$	$\{d_4, d_5, d_7\}$
B_6	$\{d_{10}, d_{11}\}$	$\{d_1, d_2, d_6\}$

Step (2) : To find iteration :

Iteration (1) :

BLOCK	IN	OUT
B_1	ϕ	$\{d_1, d_2\}$
B_2	ϕ	$\{d_3, d_4\}$
B_3	ϕ	$\{d_5\}$
B_4	ϕ	$\{d_6, d_7\}$
B_5	ϕ	$\{d_8, d_9\}$
B_6	ϕ	$\{d_{10}, d_{11}\}$

Iteration (2) :

BLOCK	IN	OUT
B_1	$\{\phi\}$	$\{d_1, d_2\}$
B_2	$\{d_1, d_2, d_6, d_7\}$	$\{d_1, d_2, d_3, d_4 d_6, d_7\}$
B_3	$\{d_3, d_4, d_8, d_9\}$	$\{d_3, d_5, d_9\}$
B_4	$\{d_3, d_4, d_5\}$	$\{d_3, d_4, d_5, d_6, d_7\}$
B_5	$\{d_5\}$	$\{d_8, d_9\}$
B_6	$\{d_6, d_7,\}$	$\{d_7, d_{10}, d_{11}\}$

Iteration (3) :

BLOCK	IN	OUT
B_1	$\{\phi\}$	$\{d_1, d_2\}$
B_2	$\{d_1, d_2, d_6, d_7\}$	$\{d_1, d_2, d_3, d_4 d_6, d_7\}$
B_3	$\{d_1, d_2, d_3, d_4, d_6, d_7, d_8, d_9\}$	$\{d_1, d_2, d_3, d_5 d_6, d_7, d_9\}$
B_4	$\{d_1, d_2, d_3, d_4, d_5, d_6, d_7, d_9\}$	$\{d_1, d_3, d_4, d_5, d_6, d_7\}$
B_5	$\{d_3, d_5, d_9\}$	$\{d_3, d_8, d_9\}$
B_6	$\{d_3, d_4, d_5, d_6, d_7,\}$	$\{d_3, d_4, d_5, d_7, d_{10}, d_{11}\}$

Iteration (4) :

BLOCK	IN	OUT
B_1	$\{\phi\}$	$\{d_1, d_2\}$
B_2	$\{d_1, d_2, d_3, d_4, d_5, d_6, d_7\}$	$\{d_1, d_2, d_3, d_4, d_6, d_7\}$
B_3	$\{d_1, d_2, d_3, d_4, d_6, d_7, d_8, d_9\}$	$\{d_1, d_2, d_3, d_5, d_6, d_7, d_9\}$
B_4	$\{d_1, d_2, d_3, d_4, d_5, d_6, d_7, d_9\}$	$\{d_1, d_3, d_4, d_5, d_6, d_7\}$
B_5	$\{d_1, d_2, d_3, d_5, d_6, d_7, d_9\}$	$\{d_1, d_2, d_3, d_6, d_8, d_9\}$
B_6	$\{d_1, d_3, d_4, d_5, d_6, d_7\}$	$\{d_1, d_3, d_4, d_5, d_7, d_{10}, d_{11}\}$

Iteration (5) :

BLOCK	IN	OUT
B_1	$\{\phi\}$	$\{d_1, d_2\}$
B_2	$\{d_1, d_2, d_3, d_4, d_5, d_6, d_7\}$	$\{d_1, d_2, d_3, d_4, d_6, d_7\}$
B_3	$\{d_1, d_2, d_3, d_4, d_6, d_7, d_8, d_9\}$	$\{d_1, d_2, d_3, d_5, d_6, d_7, d_9\}$
B_4	$\{d_1, d_2, d_3, d_4, d_5, d_6, d_7, d_9\}$	$\{d_1, d_3, d_4, d_5, d_6, d_7\}$
B_5	$\{d_1, d_2, d_3, d_5, d_6, d_7, d_9\}$	$\{d_1, d_2, d_3, d_6, d_8, d_9\}$
B_6	$\{d_1, d_3, d_4, d_5, d_6, d_7\}$	$\{d_1, d_3, d_4, d_5, d_7, d_{10}, d_{11}\}$

Since OUT after 4th iteration is same as that of OUT after 3rd iteration, hence we stop.

∴ We conclude that the definitions reaching

Upto start of B_2 are $d_1, d_2, d_3, d_4, d_5, d_7$.

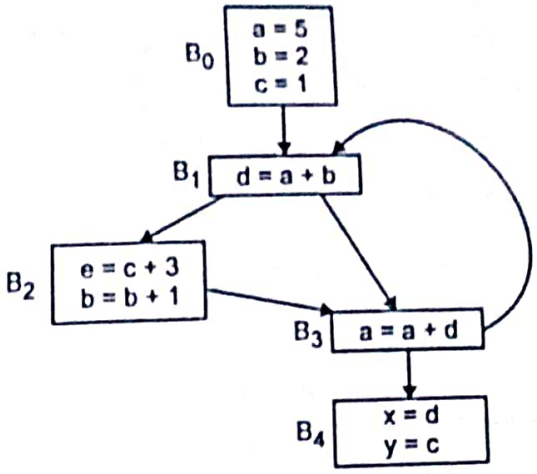
Upto start of B_3 are $d_1, d_2, d_3, d_5, d_6, d_7, d_9$

Upto start of B_4 are $d_1, d_3, d_4, d_5, d_6, d_7$

Upto start of B_5 are $d_1, d_2, d_3, d_6, d_8, d_9$.

Upto start of B_6 are $d_1, d_3, d_4, d_5, d_7, d_{10}, d_{11}$

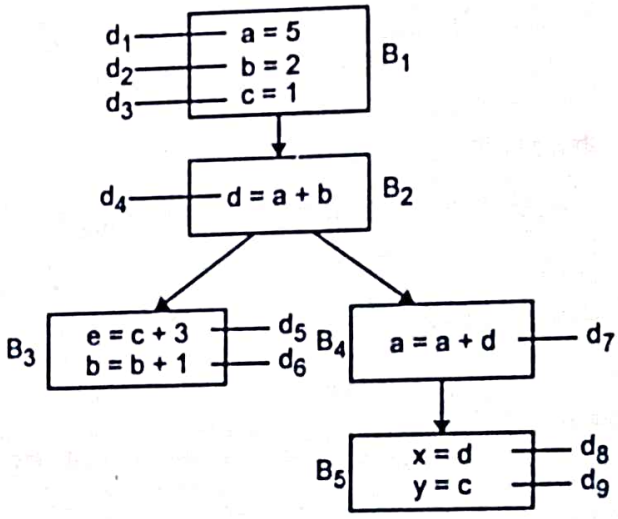
Q.52. Consider following flow graph :



Block B1 – B2 – B3 is forming a loop. Find whether this loop contains any loop-invariant computation. If yes then rewrite the code after eliminating loop-invariant computation.

CS : W-13(14M)

Ans.



Step (1) : To find GEN and KILL :

Block	GEN	KILL
B_1	$\{d_1, d_2, d_3\}$	$\{d_5, d_6\}$
B_2	$\{d_4\}$	$\{\phi\}$
B_3	$\{d_5, d_6\}$	$\{d_2, d_3\}$
B_4	$\{d_9\}$	$\{\phi\}$
B_5	$\{d_8, d_9\}$	$\{\phi\}$

Step (2) : To find iteration :

Iteration (1) :

Block	IN	OUT
B_1	ϕ	$\{d_1, d_2, d_3, d_4\}$
B_2	ϕ	$\{d_4\}$
B_3	ϕ	$\{d_5, d_6\}$
B_4	ϕ	$\{d_7\}$
B_5	ϕ	$\{d_8, d_9\}$

Iteration (2) :

Block	IN	OUT
B_1	$\{\phi\}$	$\{d_1, d_2, d_3\}$
B_2	$\{d_1, d_2, d_3\}$	$\{d_1, d_2, d_3, d_4, d_7\}$
B_3	$\{d_1, d_2, d_3, d_4, d_7\}$	$\{d_1, d_4, d_5, d_6, d_7\}$
B_4	$\{d_1, d_2, d_3, d_4, d_5, d_6, d_7\}$	$\{d_1, d_2, d_3, d_4, d_5, d_6, d_7\}$
B_5	$\{d_1, d_2, d_3, d_4, d_5, d_6, d_7\}$	$\{d_1, d_2, d_3, d_4, d_5, d_6, d_7\}$

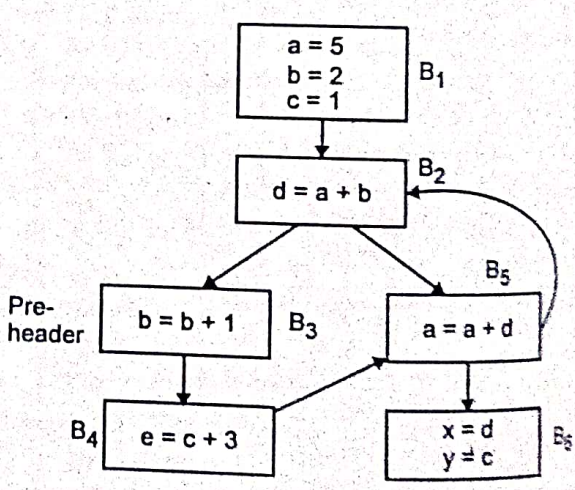
Iteration (3) :

Block	IN	OUT
B_1	$\{\phi\}$	$\{d_1, d_2, d_3\}$
B_2	$\{d_1, d_2, d_3, d_4, d_5, d_6, d_7\}$	$\{d_1, d_2, d_3, d_4, d_5, d_6, d_7\}$
B_3	$\{d_1, d_2, d_3, d_4, d_5, d_6, d_7\}$	$\{d_1, d_4, d_5, d_6, d_7\}$
B_4	$\{d_1, d_2, d_3, d_4, d_5, d_6, d_7\}$	$\{d_1, d_2, d_3, d_4, d_5, d_6, d_7\}$
B_5	$\{d_1, d_2, d_3, d_4, d_5, d_6, d_7\}$	$\{d_1, d_2, d_3, d_4, d_5, d_6, d_7, d_8, d_9\}$

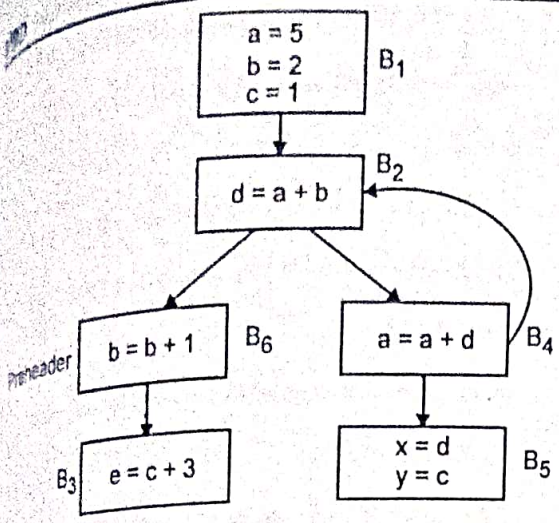
Iteration (4) :

Block	IN	OUT
B_1	$\{\phi\}$	$\{d_1, d_2, d_3,\}$
B_2	$\{d_1, d_2, d_3, d_4, d_5, d_6, d_7\}$	$\{d_1, d_2, d_3, d_4, d_5, d_6, d_7\}$
B_3	$\{d_1, d_2, d_3, d_4, d_5, d_6, d_7\}$	$\{d_1, d_2, d_3, d_4, d_5, d_6, d_7\}$
B_4	$\{d_1, d_2, d_3, d_4, d_5, d_6, d_7\}$	$\{d_1, d_2, d_3, d_4, d_5, d_6, d_7\}$
B_5	$\{d_1, d_2, d_3, d_4, d_5, d_6, d_7\}$	$\{d_1, d_2, d_3, d_4, d_5, d_6, d_7, d_8, d_9\}$

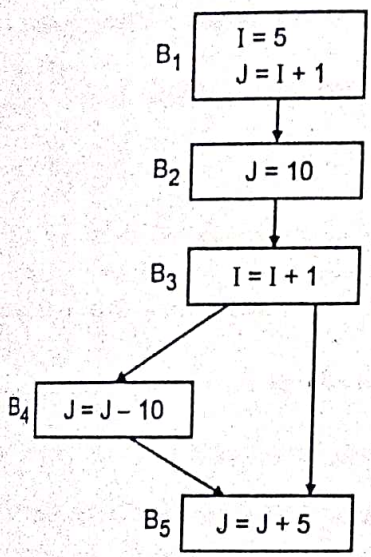
- Since values of IN and OUT after 3rd and 4th iteration are same hence we stop.
- Ud chain information is used to identify the loop invariant computations.
- If there are any loop invariant computation inside any block of loop such computations can be moved outside of a block in the form of preheater.
- This process is called as code motion.
- In the above example, computations e = c + 3, b = b + 1 are loop invariant computation. Therefore these are moved outside of loop block as shown below :



- But moving some computation is to preheader changes meaning of the program hence such computation can not be move into preheader like variable b is used in computation of e in the B_4 block.
- So without affecting the meaning of program, the flowgraph after eliminating loop invariant computation is :

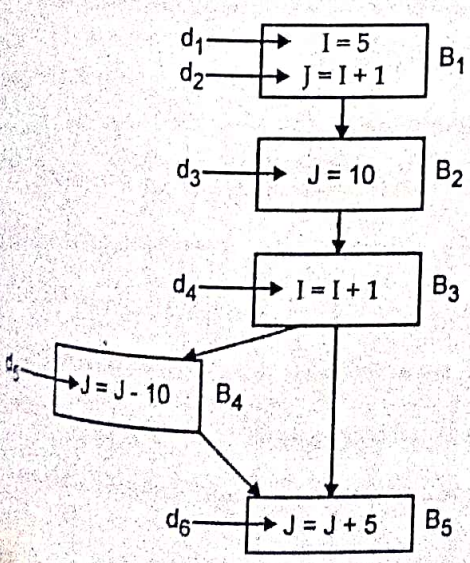


Q.5. Consider the flowgraph :



Compute IN and OUT using iterative solution of data flow equation.

CT : S-14(9M)



Step (1) : To find GEN and KILL :

Block	GEN	KILL
B_1	$\{d_1, d_2\}$	$\{d_3, d_4, d_5, d_6\}$
B_2	$\{d_3\}$	$\{d_2, d_5, d_6\}$
B_3	$\{d_4\}$	$\{d_1\}$
B_4	$\{d_5\}$	$\{d_2, d_3, d_6\}$
B_5	$\{d_6\}$	$\{d_2, d_3, d_5\}$

Step (2) : To find iteration :

Iteration (1) :

Block	IN	OUT
B_1	ϕ	$\{d_1, d_2\}$
B_2	ϕ	$\{d_3\}$
B_3	ϕ	$\{d_4\}$
B_4	ϕ	$\{d_5\}$
B_5	ϕ	$\{d_6\}$

IN(B) and OUT(B) is calculated using the following data flow equation

New IN = U OUT [P] = IN [B]

Para predecessors

OUT [B] = IN [B] $\underline{U}$ KILL [B] $\underline{U}$ GEN [B]

Iteration (2) :

Block	IN(B)	OUT(B)
B_1	$\{\phi\}$	$\{d_1, d_2\}$
B_2	$\{d_1, d_2\}$	$\{d_1, d_3\}$
B_3	$\{d_1, d_3\}$	$\{d_3, d_4\}$
B_4	$\{d_3, d_4\}$	$\{d_4, d_5\}$
B_5	$\{d_3, d_4, d_5\}$	$\{d_4, d_6\}$

VBD

Iteration (3) :

Block	IN(B)	OUT(B)
B_1	ϕ	$\{d_1, d_2\}$
B_2	$\{d_1, d_2\}$	$\{d_1, d_3\}$
B_3	$\{d_1, d_3\}$	$\{d_3, d_4\}$
B_4	$\{d_3, d_4\}$	$\{d_4, d_5\}$
B_5	$\{d_3, d_4, d_5\}$	$\{d_4, d_6\}$

IN (B) and OUT (B) after iteration 2 and 3 is solve. Therefore we stop.

SETTING UP DATA FLOW EQUATIONS TO COMPUTE REACHING DEFINITIONS

Q.54. Explain reaching definitions in detail.

Ans. Reaching definitions :

- A definition of a variable x is a statement that assigns, or may assign, a value to x. The most common forms of definition are assignments to x and statements that read a value from an I/O device and store it in x.
- These statements certainly define a value for x and they are referred to as unambiguous definitions of x.
- There are certain other kinds of statements that may define a value for x ; they are called ambiguous definitions.
- The most usual forms of ambiguous definitions of x are :

(1) A call of a procedure with x as a parameter (other than a by-value parameter) or a procedure that can access x because x is in the scope of the procedure. We also have to consider the possibility of "aliasing", where x is not in the scope of the procedure, but x has been identified with another variable that is passed as a parameter or is in the scope.

(2) An assignment through a pointer that could refer to x. For example, the assignment *q = y is a definition of x if it is possible that q points to x. In the absence of any knowledge to the contrary, we must assume that an assignment through a pointer is a definition of every variable.

- We say a definition d reaches a point p if there is a path from the point immediately following d to p, such that d is not "killed" along that path.
- Intuitively, if a definition d of some variable a reaches point p, then d might be the place at which the value of a used at p might last have been defined.
- We kill a definition of a variable a if between two points along the path there is a definition of a. Thus, a point can be reached by an unambiguous definition and an ambiguous definition of the same variable appearing later along one path.
- For example, both the definitions i : = m – 1 and j : = n in block B_1 in fig. reach the beginning of block B_2 , as does the definition j = j-1, provided there are no assignment to or reads of j in B_4, B_5 or the portion of B_3 following that definition.
- However, the assignment to j in B_3 kills the definition j : = n, so the latter does not reach B_4, B_5 or B_6 .
- By defining reaching definitions as we have, we sometimes allow inaccuracies. However, they are all in the "safe" or "conservative" direction. For example, notice our assumption that all edges of a flow graph can be traversed. This may not be true in practice.
- For example, for no values of a and b can control actually reach the assignment a : = 4 in the following program fragment :

  if a = b then a : = 2

  else if a = b then a : = 4
- To decide in general whether each path in a flow graph can be taken is an undecidable problem, and we shall not attempt to solve it.
- A recurring theme in the design of code improving transformations is that we must make only conservative decisions in the face of any doubt, although conservative strategies may cause us to miss some transformations that we actually could make safely.

- A decision is conservative if it never leads to a change in what the program computes. In applications of reaching definitions, it is normally conservative to assume that a definition can reach a point even if it might not.
- Thus, we allow paths that may never be traversed in any execution of the program and we allow definitions to pass through ambiguous definitions of the same variable.

Q.55. Explain the computation of reaching definition by using global data flow analysis with one example.

Ans.

- As we know data flow analysis is done by solving the system of equations that relate information at various points in the program.
- The equation is :

OUT [S] = GEN [S] $\cup$ (IN [S] – KILL [S])

We define GEN, KILL, IN and OUT set of a basic block.

Here 'S' is a basic block.

- GEN [S] = Set of all those definitions which gets generated in block S.
- KILL [S] = Set of all those definitions outside S defining the same name which are defined in S.
- IN[S] = Set of all definitions reaching upto a point immediately before the 1st statement of block, i.e. start of block.
- OUT[S] = Set of all definitions reaching upto a point immediately after the last statement of block S.
- A causing to definitions IN [S] = $\cup$ OUT [P]
- For every predecessor 'P' or 'S'.

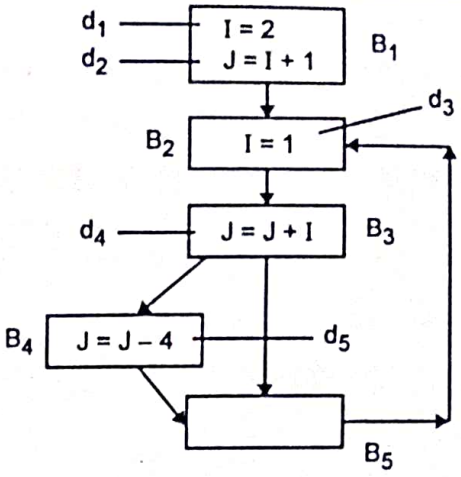
OUT[S] = IN [S] – KILL [S] $\cup$ GEN [S]

These equations are called data flow equations.

- To get IN [S], we have to solve the above equations.
- To start with IN [S] = 0, and

OUT [S] = GEN [S] for every S.

- Using these values we start our iterations and compute IN [S], iterations are continued till we find that there is no change in what is computed in previous iterations.
- We take one example, consider following program flow graph



GEN and KILL information :

Block	GEN	KILL
B_1	$\{d_1, d_2\}$	$\{d_3, d_4, d_5\}$
B_2	$\{d_3\}$	$\{d_1\}$
B_3	$\{d_4\}$	$\{d_2, d_5\}$
B_4	$\{d_5\}$	$\{d_2, d_4\}$
B_5	ϕ	ϕ

Initially, we have

Block	IN	OUT
B_1	ϕ	$\{d_1, d_2\}$
B_2	ϕ	$\{d_3\}$
B_3	ϕ	$\{d_4\}$
B_4	ϕ	$\{d_5\}$
B_5	ϕ	ϕ

After 1st iteration :

Block	IN	OUT
B_1	ϕ	$\{d_1, d_2\}$
B_2	$\{d_1, d_2\}$	$\{d_2, d_3\}$
B_3	$\{d_3\}$	$\{d_3, d_4\}$
B_4	$\{d_4\}$	$\{d_5\}$
B_5	$\{d_4, d_5\}$	$\{d_4, d_5\}$

Here IN and OUT sets are calculated by using the two data flow equations.

After 2nd iteration :

Block	IN	OUT
B_1	ϕ	$\{d_1, d_2\}$
B_2	$\{d_1, d_2, d_4, d_5\}$	$\{d_2, d_3, d_4, d_5\}$
B_3	$\{d_2, d_3\}$	$\{d_3, d_4\}$
B_4	$\{d_3, d_4\}$	$\{d_4, d_5\}$
B_5	$\{d_3, d_4, d_5\}$	$\{d_3, d_4, d_5\}$

- Since OUT after 3rd iteration is same as that of OUT after 2nd iteration we stop.

$\therefore$ We conclude that the definitions reaching

upto start of B_2 are d_1, d_2, d_3, d_4, d_5

upto start of B_3 are d_2, d_3, d_4, d_5

upto start of B_4 are d_3, d_4

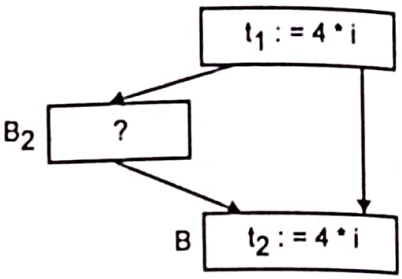
upto start of B_5 are d_3, d_4, d_5.

AVAILABLE EXPRESSIONS

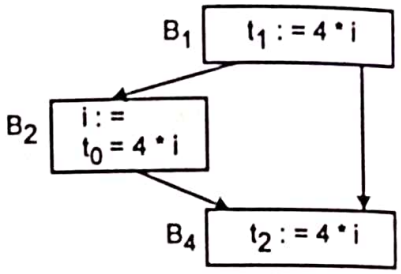
Q.56. Explain available expressions in detail.

Ans. Available expressions :

- An expression x + y is available at a point p if every path (not necessarily cycle-free) from the initial node to p evaluates x + y and after the last such evaluation prior to reaching p, there are no subsequent assignments to x or y.
- For available expressions we say that a block kills expression x + y if it assigns (or may assign) x or y and does not subsequently recomputed x + y.
- A block generates expression x + y if it definitely evaluates x + y and does not subsequently redefine x or y.
- Note that the notion of "killing" or "generating" an available expression is not exactly the same as that for reaching definitions.
- Nevertheless, these notions of "kill" and "generate" obey the same laws as they do for reaching definitions.
- The primary use of available expressions information is for detecting common subexpressions.
- For example, in fig. the expression 4 * i in the block B_3 will be a common subexpression if 4 * i is available at the entry point of block B_3.
- It will be available if i is not assigned a new value in block B_2 or if as in fig. (b), 4 * i is recomputed after i is assigned in B_2.



(a)



(b)

Fig. Potential common subexpressions across blocks

- We can easily compute the set of generated expressions for each point in a block, working from beginning to end of the block.
- At the point prior to the block, assume no expressions are available.
- If at point p set A of expressions is available and q is the point after p, with statement x : = y + z between them, then we form the set of expressions available at q by the following two steps.

(1) Add to A the expression y + z.

(2) Delete from A any expression involving x.

- Note the steps must be done in the correct order, as x could be the same as y or z.
- After we reach the end of the block, A is the set of generated expressions for the block.
- The set of killed expressions is all expressions, say, y + z such that either y or z is defined in the block and y + z is not generated by the block.

Q.57. Give algorithm for available expression.

Ans. Algorithm for available expressions :

Input : A flow graph G with e_kill [B] and e_gen[B] computed for each block B. The initial block is B_1.

Output : The set IN[B] for each block B.

```
Method : IN[B1] := ø ;
OUT[B1] := e_GEN[B1] ;
        /* in and out never change for the initial node, B1 */
for B ≠ B1 do out [B] := U – e_kill[B] ;
                        /* initial estimate is too large */
CHANGE := true ;
while CHANGE do begin
    CHANGE := false ;
    for B ≠ B1 do begin
        IN[B] := ∩ (P a predecessor of B) OUT [P] ;
        OLDOUT := OUT[B] ;
        Out[B] := e_GEN[B] ∪
                        (IN[B] – e_KILL[B]) ;
        if OUT[B] ≠ OLDOUT then
                        CHANGE := true
    end
end
```

LIVE VARIABLES

Q.58. What is live variable? Explain live variable analysis.

Ans. Live variable :

- A variable is live if it holds a value that may be needed in the future otherwise it is said to be dead.

Live-variable analysis :

- A number of code improving transformations depend on information computed in the direction opposite to the flow of control in a program.
- In live-variable analysis we wish to know for variable x and point p whether the value of x at p could be used along some path in the flow graph starting at p. If so, we say x is live at p ; otherwise x is dead at p.
- An important use for live-variable information comes when we generate object code.
- After a value is computed in a register and presumably used within a block, it is not necessary to store that value if it is dead at the end of the block.
- Also, if all registers are full and we need another register, we should favor using a register with a dead value, since that value does not have to be stored.
- Let us define IN[B] to be the set of variables live at the point immediately before block B and define OUT[B] to be the same at the point immediately after the block.
- Let def[B] be the set of variables definitely assigned, values in B prior to any use of that variable in B and let use[B] be the set of variables whose values may be used in B prior to any definition of the variable.
- Then the equations relating def and use to the unknowns IN and are :

$$IN[B] = use[B] \cup (OUT[B] - DEF[B])$$

$$OUT[B] := \bigcup_{\text{S a successor of B}} IN[S]$$

- The first group of equations says that a variable is live coming into a block if either it is used before redefinition in the block or it is live coming out of the block and is not redefined in the block.
- The second group of equations says that a variable is live coming out of a block if and only if it is live coming into one of its successors.
- The relations between equation above and the reaching-definitions equations should be noticed. Here, IN and OUT have their roles interchanged and use and def substitute for gen and kill, respectively.

Algorithm for Live variable analysis :

Input : A flow graph with def and use computed for each block.

Output : OUT[B], the set of variable live on exit from each block B of the flow graph.

Method :

for each block B do $IN[B_1] := ø$;

while changes to any of the IN's occur do

    for each block B do begin

$$OUT[B] := \bigcup_{\text{S a successor of B}} IN[S] ;$$

$$IN[B] ; = USE[B] \cup (OUT[B] - DEF[B])$$

end

(4)	Static allocation is simple, but not efficient memory management technique.	Heap allocation does memory management in efficient way.
(5)	Static allocation strategy is faster in accessing data as compared to heap allocation.	Heap allocation is slow in accessing as there is chance of creation of holes in reusing the free space.
(6)	It is easy to implement.	It is comparatively expensive to implement.

UNIT - IV

GLOBAL OPTIMIZATION

Q.22. Explain Global Optimization.

Ans. Global Optimization:

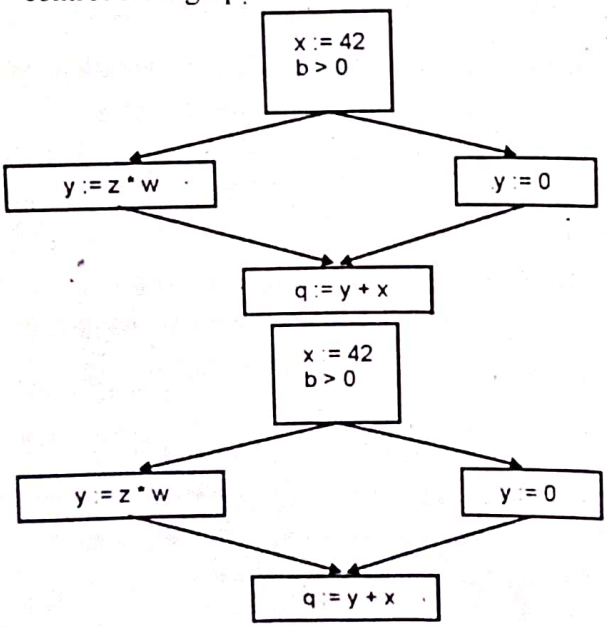
The scope of Global optimization is generally a program unit, usually spanning over number of basic blocks.

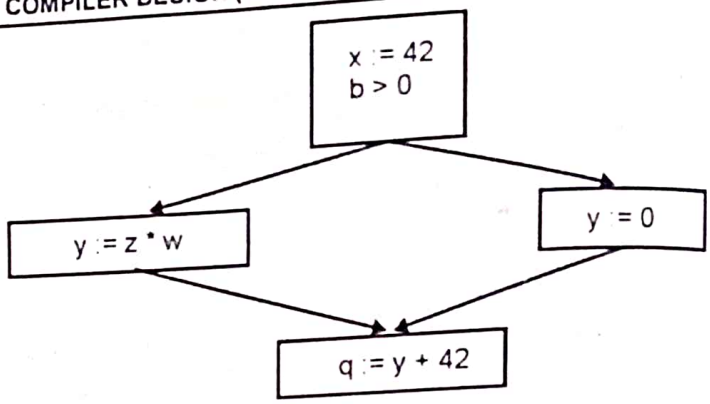
- It can achieved better transformation leading to more optimized.
- Program flow graph normally undergoes two types analysis:

(a) Control Flow Analysis.

(b) Data Flow Analysis.

- These optimizations can be extended to an entire control-flow graph.





UNIT - V

ARCHITECTURE DEPENDENT CODE IMPROVEMENT IN COMPILER DESIGN: INSTRUCTION SCHEDULING

Q.23.Explain Architecture dependent code improvement in compiler design: Instruction Scheduling.

OR Explain Code Optimizing Process.

OR What is Code Optimization?

Ans. Code Optimization:

- Code optimization is a program transformation approach that aims to enhance code by reducing resource consumption (i.e. CPU and memory) while maintaining high performance.
- In code optimization, high-level generic programming structures are substituted with low-level programming codes. The three guidelines for code optimization are as follows:

(i) In no way should the output code alter the program's meaning.

(ii) The program's speed should be increased, and it should use fewer resources if at all feasible.

(iii) The optimization step should be quick and not hinder the compilation process.

- At several stages of the compilation process, efforts to optimize the code might be made. Users can alter/rearrange the code at first or create the code using better algorithms. The compiler can improve loops and address computations after completing intermediate code. The compiler can leverage Memory Hierarchy and CPU registers while generating the target machine code.

Q.24.Why Code Optimization is necessary?

Ans. Code Optimization:

- Optimizing code in compiler design is important because it directly affects the performance of the compiled code.
- A well-optimized code runs faster and consumes fewer resources, leading to improved overall system performance and reduced energy consumption.
- Additionally, optimization can reduce the size of the generated code, which is important for embedded systems with limited memory.
- The optimization process can also help identify and eliminate bottlenecks in the code, leading to more efficient algorithms and improved software design. Overall, optimization is a critical step in the compiler design process that can greatly improve the end-user experience.

Q.25.Explain types of Code Optimization.

Ans. The code optimization process can be broadly classified into two types:

(i) Machine Independent Optimization

(ii) Machine Dependent Optimization

(i) Machine Independent Optimization:

This step of code optimization aims to optimize the intermediate code to produce a better target code. No CPU registers or absolute memory addresses are involved in the section of the intermediate code that is translated here.

(ii) Machine Dependent Optimization:

- After the target code has been created and converted to fit the target machine architecture, machine-dependent optimization is performed. It may use absolute memory references rather than relative memory accesses and requires CPU registers. Machine-dependent optimizers make a concerted attempt to maximize the memory hierarchy's benefits.
- A code has several statements, loops, branches, etc. So code optimization must be performed on all of them. The code optimization is done differently, considering the following:

(1) Loop Optimization:

The majority of programs in the system operate in a loop. It is vital to optimize the loops to save CPU cycles and memory. The following strategies can be used to improve loops.

(i) Loop-invariant Code:

It is a piece of code that sits in the loop and computes the same value each time an iteration is performed. This code may be moved out of the loop by storing it to be calculated just once rather than with each iteration.

(ii) Induction Analysis:

If a loop-invariant value changes the value of a variable within the loop, it is termed an induction variable.

(iii) Strength Reduction:

Some expressions use more CPU cycles, time, and memory than others. These expressions should be replaced with less expensive expressions without sacrificing the expression's output. For example, multiplication (x * 2) uses more CPU cycles than (x << 1) but produces the same output.

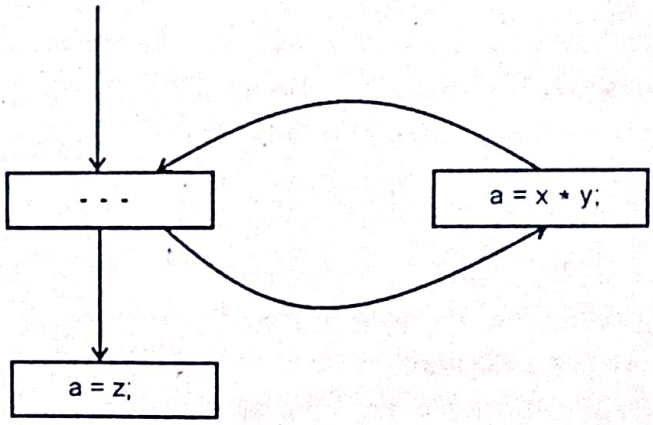


Fig.1. Loop Code flow chart

(2) Partially Dead Code:

- Some code statements include calculated values utilized only in particular conditions, i.e. the values are used sometimes and not others. Partially dead-code refers to such codes.
- The control flow diagram below shows a program section in which the variable 'a is utilized to assign the output of the equation 'x * y'. Let's

pretend that the 'a variable's value is never utilized within the loop. 'a' is given the variable 'z' value, which will be utilized later in the program, immediately after the control leaves the loop. We may infer that because the assignment code 'a' is never utilized anywhere, it is suitable for deletion.

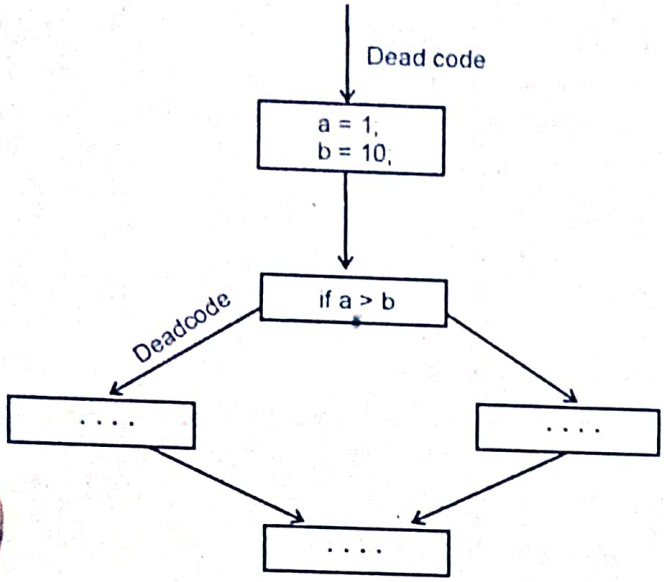


Fig.2. Dead Code flow chart

- Similarly, the conditional statement in the image above is always false, meaning that the code written in the "true" case will never be run and so may be eliminated.

(3) Unreachable Code Elimination:

A control flow graph should be created first. An inaccessible code block does not have an incoming edge. The inaccessible branches can be deleted after continual propagation and folding.

(4) Function Inlining:

The body of the function takes the place of a function call. This saves a lot of time by eliminating the need to copy all parameters, store the return address, and so on.

(5) Function Cloning:

For different calling arguments, specialized codes for a function are constructed. Overloading a function is an example of this.

(6) Partial Redundancy:

In a parallel route, redundant expressions are calculated many times without changing the operands. Partial-redundant expressions, on the other hand, are calculated several times along a path without changing the operands. By employing a code-motion approach, loop-invariant code may be rendered largely redundant.

An example of a partially redundant code can be:

```
If (condition) {
    a = y OP z;
} else {
    ...
}
c = y OP z;
```

We assume that the operands' values (y and z) do not change when variable an is assigned to variable c. If the condition statement is true, y OP z is calculated twice; otherwise, it is computed once. As stated below, code motion may be utilized to remove redundancy:

```
If (condition) {
    ...
    tmp = y OP z;
    a = tmp;
    ...
} else {
    ...
    tmp = y OP z;
}
c = tmp;
```

y OP z should only be computed once, regardless of whether the condition is true or not.

TARGET CODE GENERATION

Q.16.Explain Target Code Generation.

Ans. Target Code Generation:

Target code generation is the final Phase of Compiler:

(i) Input: Optimized Intermediate Representation.

(ii) Output: Target Code.

(iii) Task Performed: Register allocation methods and optimization, assembly level code.

Gencode (t_3)	Call gencode(E)
	POP R_1
	call gencode (t_2)
	generate an instruction
	SUB R_0, R_1
	PUSH R_1
		R_1, R_0
		R_0
		R_1, R_0
			MOV E, R_1
			MOV C, R_0
			ADD D, R_0
			SUB R_0, R_1

Gencode(E)	Generate an instruction MOV E_1R_1	R_1, R_0	MOV E, R_1
Gencode (t_2)	Gencode(C) Gencode an instruction ADD D, R_0	R_0	MOV C, R_0 ADD D, R_0
Gencode(C)	Generate an instruction MOV C, R_0	R_0	
Gencode((t_1))	Gencode (A) Generate an instruction ADD B, R_0	R_0	MOV A, R_0 ADD B, R_0
Gencode(A)	Generate an instruction MOV A, R_0	R_0	MOV A, R_0

PEEPHOLE OPTIMIZATION

Q.34. Explain peephole optimization techniques in detail.

CS : W-10, S-12(7M), W-14(5M), CT : W-11 (7M)

OR Write short note on peephole optimization. **CS :S-11(7M),**

CT : S-11, 14, W-10(5M), W-12(4M), S-13(7M)

Ans. Peephole optimization

- Most of the compilers produce good code by proper instruction selection and register allocation but few of them use alternate strategy.
- They generate naive code and improve the quality of code by applying optimizing transformation.
- The term optimizing is misleading because there is no guarantee that the resulting code is optimal.
- So the simple and effective technique is used that is peephole optimization which is done by examining a sliding window of target instruction and replacing the instruction sequence by a shorter or faster sequence within the peephole.
- It can be applied directly after intermediate code.
- It can be small sliding window on the program.
- The characteristic of peephole optimization is as follows :

(1) Elimination of redundant loads and stores.

(2) Elimination of multiple jumps

(3) Elimination of unreachable code

(4) Algebraic simplifications

(5) Reducing for strength

(6) Use of machine idioms.

(1) Eliminating redundant loads and stores :

- If the target code contains the instruction sequence :

(1) MOV R, a

(2) MOV a, R

we can delete the second instruction if it an unlabeled instruction.

- This is because the first instruction ensures that the value of a is already in the register R.
- If it is labeled, there is no guarantee that step 1 will always be executed before step 2.

(2) Eliminating multiple jumps :

- If we have jumps to other jumps, then the unnecessary jumps can be eliminated in either intermediate code or the target code. If we have a jump sequence :

```
    goto L1
    ....
L1 : goto L2
```

then this can be replaced by :

```
    goto L2
    ....
L1 : goto L2
```

VBD

If there are now no jumps to L1, then it may be possible to eliminate the statement, provided it is preceded by an unconditional jump. Similarly, the sequence

```
    if a < b goto L1
    ...
L1  goto L2
```

can be replaced by

```
    if a < b goto L2
    ...
L1  goto L2
```

Eliminating unreachable code :

An unlabeled instruction that immediately follows an unconditional jump can possibly be removed and this operation can be repeated in order to eliminate a sequence of instructions.

For debugging purposes a large program may have within it certain segments that are executed only if a debug variable is one.

For example, the source code may be :

```
#define debg 0
...
if (debug)
    {
      print debugging information
    }
```

This if statement is translated in the intermediate code to :

```
if debug = 1 goto L1
goto L2
L1 : print debugging information
L2 :
```

One of the optimizations is to replace the pair :

```
if debug = 1 goto L1
goto L2
```

within the statements with a single conditional goto statement by negating the condition and changing its target, as shown below :

```
    if debug ≠ 1 goto L2
Print debugging information
L2 :
```

Since debug is a constant zero by constant propagation, this code will become :

```
if 0 ≠ 1 goto L2
Print debugging information
L2:
```

Since 0 ≠ 1 is always true this will become :

```
goto L2
Print debugging information
L2 :
```

Therefore, the statements that print the debugging information are unreachable and can be eliminated one at a time.

(4) Algebraic simplifications :

If statements like :

$a = a + 0$

$a = a * 1$

are generated in the code, they can be eliminated, because zero is an additive identity and one is a multiplicative identity.

(5) Reducing strength :

- Certain machine instructions are considered to be cheaper than others. Hence, if we replace expensive operations by equivalent cheaper ones on the target machine, then the efficiency will be better.
- For example, x^2 is invariable cheaper to implement as X * X than as a call to an exponential routine.
- Similarly, fixed-point multiplication or division by a power of two is cheaper to implement as a shift.

(6) Using machine idioms :

- The target machine may have hardware instructions to implement certain specific operation efficiently.
- Detecting situations that permit the use of these instructions can reduce execution time significantly.
- For example, some machines have auto-increment and auto-decrement addressing modes.
- Using these modes can greatly improve the quality of the code when pushing or popping a stack. These modes can also be used for implementing statements like a = a + 1.

VBD