B.Tech.

SIXTH SEMESTER EXAMINATION, 2009-10

COMPILER DESIGN

TCS-502

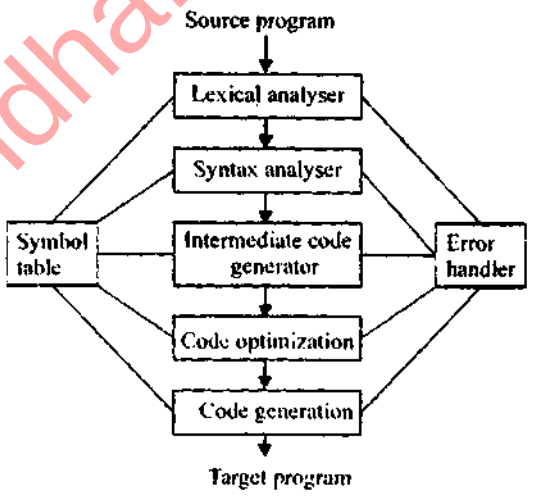
Time : *3 Hours]* *[Total Marks : 100*

Note : *(1) Attempt* ***all*** *questions.*

Q. 1. Attempt any two parts of the following : **10 × 2 = 20**

(a) Explain all the necessary phases and passes in a compiler design. Write down the purpose of each pass. What is bootstraping ?

Ans. Phases of compiler : The compilation process is divided into a series of subprocesses called phases of compiler. A phase is a logically cohesive operation that take as input one representation of the source program and produces as output another representation.



Phases of compiler

Lexical Analyser : The first phase called the lexical Analyser or scanner is the interface between the source program and the compiler. The lexical analyser reads the source program one character at time, separates character of the source program into groups that logically belong together called lexems. The category of lexemes is called **tokens**. Each token represents a sequence of characters that can be treated as a signal logical entity. The usual tokens are

Identifier
Keywords
Constants
Operators
Punctuation symbols

Syntax Analyser : The syntax Anayser groups tokens together into syntatic structure. For example, the three tokens representing $A + B$ might be group into a synatatiure structure called an expression. Expressions might further be combined to form statements. Often the syntatic structure can be regarded as a tree whose leaves are the tokens. The interior nodes of the tree represent strings of tokens that logically belong together.

Intermediate code generator : It uses the structure produced by the syntax analyser to create a stream of simple instructions. Many styles of intermediate code are possible. One common style uses instructions with one operator and a small number of operands. This style of intermediate code is called three-address-code.

e.g. The three address code for the statement

$A \setminus B * C$ is

$T_1 := A \setminus B$

$T_2 := T_1 * C$

Code Optimizer : This is an optional phase designed to improve the intermediate code so that ultimate object program runs faster and/or takes less space. Its output is another intermediate code program that does the same jobs as the original but perhaps in a way that save time and/or space.

Code Generator : This phase produces the object code by deciding on the memory locations for data, selecting code to access each datum, and selecting the register in which each computation is to be done Designing a code generator that produces truly efficient object programs is one of the most difficult parts of compiler design, both practically and theoretically.

Table management or bookkeeping : It keeps track of the names used by the program and records essential information about each. Such as its type (integer, real etc) the data structure used to record this information is called a symbol table.

Error Handler : The Error Handler is invoked when a flaw in the source program is detected. It must warn the programmer by issuing a diagnostic and adjust the information being passed from phase to phase so each phase can proceed.

The purpose of passes : In an implementation of compiler, portions of one or more phases are combined into a module called a pass. A pass reads the source program or the output of the previous pass, makes the transformations specified by its phases, and writes output into an intermediate file, which may then be read by a subsequent pass. If several phases are ground into one pass, then the operation of the phases may be interleaved, with control alternating among several phases.

Bootstrapping : Suppose we have a compiler C_A^{LA} for a language L written in machine language A and produces the object code for machine A .

Now suppose we want to produce another compiler for L to run on machine B to produce code for B.

We can easily write a compiler C_L^{LB} in L for language L that produces code for machine B, Since L Language is available on machine A the compiler C_L^{LB} can be compiled on A

i.e. $C_L^{LB} \rightarrow \boxed{C_A^{LA}} \rightarrow C_A^{LB}$ (Cross compiler)

We have got C_A^{LB} a cross-compiler which run on machine A and produces object code for machine B.

Now we run C_L^{LB} through this cross-compiler to produce the desired compiler for L that runs on machine B and produces object code for B.

i.e. $C_L^{LB} \rightarrow \boxed{C_A^{LB}} \rightarrow C_B^{LB}$

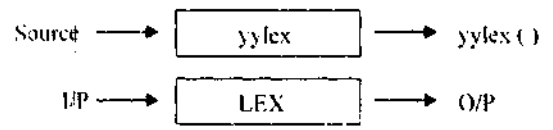
Bootstrapping of a compiler.

Q. 1. (b) What do you understand by lexical-analyzer generator and LEX-compiler.

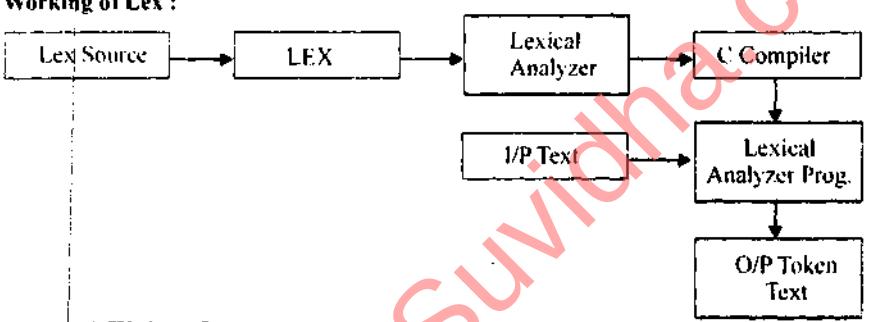
Ans. Lex is a program generator designed for lexical processing of character input/output stream. **Anything from simple text search program** that looks for pattern in its input-output file to a 'C' **compiler that transforms a program** into optimized code.

In program with structure input-output two tasks occur over and over. We divide the input output **into meaningful units and** then discovering the relationships among the units for C

program (the units are variable names, constants, and strings. This division into units (called tokens) is known as Lexical Analyzer or Lexing. Lex helps by taking a set of descriptions of possible tokens n producing a routine called a Lexical Analyzer or Lexer or Scanner. The set of descriptions given to lex are called lex specification. The Token description that the lex uses are known as Regular Expressions. When we write a lex specification we create a set of pattern which lex matches against the input-output. The lex program divides the input-output into string which we call Tokens.



Working of Lex :



Q. 1. (c) Write short notes on:

(i) Context free grammars. Give the examples of context free grammars.

Ans. The context free grammar came into existence in 1965 by Chomsky. A context free grammar describes a language by recursive rules known as productions. A context free grammar can be described as:

CFG consist of four touples namely

V → Set of variables (Non-terminals)

T → Set of terminals

S → Starting symbol

P → Set of Productions

These all are represented in the form of G (V, T, P, S) context free grammar in the form of $A \rightarrow \alpha$, where 'A' is a Non Terminal & 'α' is denoted by $(V \cup T)^*$.

According to Question, for example generate palindrom for binary number

V = {S}

T = {a, b},

S = {S}

Production : $S \rightarrow aS|bS|a|b|\in$

Q. 1. (c) (ii) Parse trees. Give an example of parse tree.

Ans. We can create a graphical representation for derivations that filters out the choice regarding replacement order. This representation is called the parse tree, and it has the important purpose of making explicit the hierarchical syntatic structure of sentences that is implied by the grammar.

Now, According to Question, for example :

Expression : id + id * id

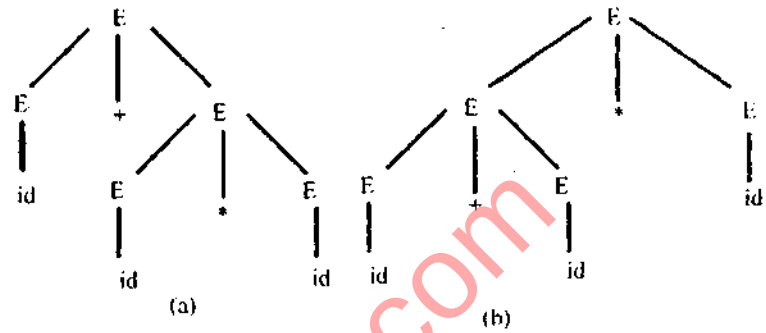
$E \rightarrow E + E$

$E \rightarrow id + E$

$E \rightarrow id + E * E$

$E \rightarrow id + id * E$

$E \rightarrow id + id * id$



Parse tree for the above expression

Parse tree are of two type shown as above i.e. left most derivation and right most derivation.

Q. 2. Attempt any two parts of the following : **10 × 2 = 20**

(a) Explain about basic parsing techniques. What is top down parsing ?

Ans. Parsing is a technique that takes as input string W and produce as output either a parse tree for W. If W is a valid sentence of grammar or an error message indicating that W is not a sentence of grammar. The goal of parsing is to determine the syntatic validity of a source string. If the string is valid, a tree is build.

There are two types of parsing :

1. Top down Parsing 2. Bottom up Parsing

Now, according to question the description of top down Parsing.

To find the left most derivation for the input string W. Since string W is scan by the parser left to right, one symbol at a time. Left most generate the leaves the parse tree in left to right order which match the input order. Top down Parse to find the left most derivation for input string. Basically in top down mechanism every terminal symbol generating by some production of the grammar is match with the input string symbol pointed to by the string marker. If the match is successful the parse can continue if a mismatch occurs then gone wrong. We will reject previous string & then string marker is reset to the position when the rejected production was made. This is known as back tracking. And back tracking is one of the major drawback of top down Parsing left recursion & left factoring comes under the top down parsing.

Q. 2. (b) Explain the following:

(i) Constructing SLR parsing tables. (ii) Constructing LALR parsing tables.

Ans. (i) INPUT C , the canonical collection of sets of items for an augmented grammar G.

OUTPUT : It possible, an LR parsing table consisting of a parsing action function ACTION and a goto function GOTO.

METHOD : Let $C = \{I_0, I_1, \ldots I_n\}$. The states of the parser are 0, 1, ... n, state i being constructed from I_i. The parsing actions for state i are determined as follows:

(1) If $[A \rightarrow \alpha a\beta]$ is in I_i and GOTO $(I_i, a) = I_j$, then set ACTION $[i, a]$ to "shift j". Here a is a terminal.

(2) If $[A \rightarrow \alpha]$ is in I_i, then set ACTION $[i, a]$ to reduce $A \rightarrow \alpha$" for all a in FOLLOW (A).

(3) If $[S' \rightarrow S]$ is in I_i, then set ACTION $[i, \$]$ to "accept".

If any conflicting actions are generated by the above rules, we say the grammar is not SLR (1). The algorithm fails to produce a valid parser in this case.

The goto transitions for state i are constructed using the rule :

(4) If GOTO $(I_i, A) = I_j$, then Goto $[i, A] = j$

(5) All entries not defined by rules (1) through (4) are made "error".

(6) The initial state of the parser is the one constructed from the set of items containing $[S' \rightarrow .S]$

(ii) INPUT : A grammar G augmented by production $S' \rightarrow S$

OUTPUT : The LALR parsing tables ACTION and GOTO

METHOD: (1) Construct $C = \{I_0, I_1, \ldots, I_n\}$, the collection of set of LR (1) items.

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

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

(4) The GOTO table is constructed as follows. If J is the union of one or more sets of LR(1) items, i.e., $J = I_1 UI_2 U \ldots UI_m$, then the cores of GOTO (I_1, X), GOTO (I_2, X),

.

.

.

GOTO(I_k, X) are the same,

since $I_1, I_2, \ldots I_k$ all have the same core. Let k be the union of all sets of items having the same core as GOTO (I_1, X). Then GOTO $(J, X) = K$

The table produced by this algorithm is called the LALR parsing table for G. If there are no parsing action conflicts, then the given grammar is said to be an LALR (1) grammar.

Q. 2. (c) How do you implement the LR parsing tables ? Why do we need LR parsing tables ?

Ans. LR Parsing table can be divided into two parts:

(1) Action table (2) Goto table

Description of Action table :

(1) If $(A \rightarrow \alpha . a \beta)$ is in I_i

goto $(I_i, a) = I_j$ then

set action $[I_i, a]$ to shift j (S_j)

(2) Reduce

If $(A \rightarrow \alpha.)$ is in I_i then for energy b in follow (A)

set action $[I_i, b]$ = reduce $A \rightarrow \alpha$. (R_k, k) is the number of production $A \rightarrow \alpha$ in grammar)

(3) accept :

if $(S' \rightarrow S)$ is in I_i

set action $[I_i, \$]$ = accept

Description of Go To table:

(1) if goto $(I_i, A) = I_j$

set goto $[I_i, A] = j$

(2) No entry → error.

Need of LR parsing Table:

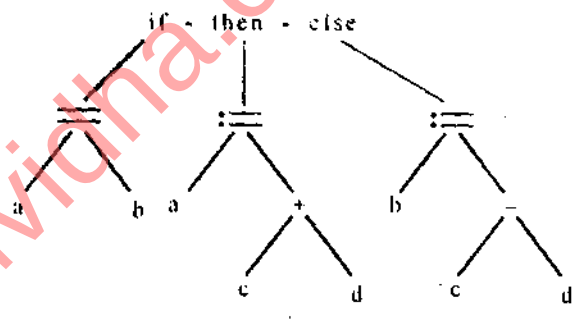
There are many different parsing tables that can be used in an LR parser for a given grammar. Some parsing tables may detect errors sooner than others, but they all accept the same sentences, exactly the sentences generated by the grammar. The three different LR is easiest to implement unfortunately, it may fail to produce a table for certain grammars on which the other methods succeed.

The second method, called canonical LR, is the most powerful and will work on a very large loss of grammars. Unfortunately, the canonical LR method can be very expensive to implement. The third method, called look ahead LR (LALR for short), is intermediate in power between the SLR and the canonical LR methods. The LALR method will work on most programming language grammars and, with some effort, can be implemented efficiently.

Q. 3. Attempt any two parts of the following : $10 \times 2 = 20$

(a) What is the intermediate code in Syntax-directed Translation ? What is a syntax tree ? Give an example of syntax tree.

Ans. In many compilers the source code is translated into a language which is intermediated in complexity between a programming language and machine code. Such a language is therefore called intermediate code. It is possible to translate directly from source to machine or assembly language in a syntax-directed way but, as we have mentioned, doing so makes generation of optimal, or even relatively good, code a difficult task. Four kinds of intermediate code often used in compilers are postfix notation, syntax trees, quadruples, and triples. One such variant of a parse tree is what is called an syntax tree, a tree in which each leaf represents an operand and each interior node an operator.



For example: The syntax tree for the statement if a = b then a:= c + d else b: = c – d

Q. 3. (b) What is postfix translation ? Explain it with a suitable example.

Ans. We have called a translation scheme postfix if for each production $A \rightarrow \alpha$, the translation rule for A. CODE consists of the concatenation of the CODE translations of the non-terminals in α, in the same order as the non terminals appear in α, followed by a tail of output. We have seen that postfix translations can be implemented by emitting the tail as each production is recognized. Therefore, to reduce space requirements, it is quite, useful that CODE be a postfix translation, for if not, we must use a scheme like generation of a parse tree, followed by a walk of the tree, to produce the intermediate-language form of the source program.

For example:

a * – (b + c) translate into postfix form

Step I

a * – (b + c) → a * – (bc +)

Step II

a * –(bc +) → a * –bc +

Q. 3. (c) Explain the following:

(i) Effect of the statements that alter the flow of control (of a program) in Syntax-directed translation.

(ii) Role of Array-references in the arithmetic expressions in syntax-directed translation.

Ans. *(i)* The translation of source statements that effect the flow of control in a program. We fix our attention on the generation of quadruples, and the notation regarding translation filed names and list-handling procedures from that section carriers over to this section as well.

Unconditional Jumps: We can describe the syntax of labelled statements with productions such as

S → LABEL : S

LABEL → id

The semantic action associated with LABEL → *id* is to

(1) Install that identifier in the symbol table if it is 't' already there,

(2) Record that the quadruple referred to by this label is the current value of NEXTQUAD, and finally

(3) back patch the list of goto's whose targets are the label just discovered.

Structured Flow-of-Control Constructs:

Some difficult example of flow of control are nested or structured control statements

S—Statement

L—Statement list

A—assignment statement

E—Boolean—valued expression,

(1) S → if E then S

(2) S → if E then S else S

(3) S → While E do S

(4) S → begin L end

(5) S → A

(6) L → L; S

(7) | S

(ii) Role of array references in Arithmetic Expressions : In this we expand on that translation, permitting array references as operands one approach is to leave a reference such as A [I, j] intact in the intermediate code, leaving it to the code-generation phase to produce object code that computes the offset of *A* [*I*, *j*] from the base of array A and then performs an indexing operations.

Grammar for Array References

A → *L* := *E*

L → *id*[elist] | *id*

elist → elist, E | E

L → E ' E | (E, L

Thus is, an assignment statement A is an l-value followed by an assignment symbol followed by an expression E

Q. 4. Attempt any two parts of the following : **10 × 2 = 20**

(a) What information is represented by symbol tables ? Explain the data structure for symbol tables.

Ans. Information represented by symbol table are :

- The entries in the symbol table are for declaration of names.
- When an occurrence of a name in the source text is looked up in the symbol table the entry for appropriate declaration of that name must be returned.
- The scope rule of the source language determine which declaration is appropriate.
- A simple method is to maintain a separate symbol table for each scope.

- The symbol table for a procedure or scope is the compile time equivalent of an activation record.
- Information for the non locals of a procedure is found by scanning the symbol tables for the enclosing procedures following the scope rules of the language.
- Most closely nested scope rules can be implemented by adapting the data structures as discussed before.
- Block must be also be numbered if the language is block structured.

Data Structures For Symbol Tables Are : The three symbol-table mechanisms we discuss in this section are linear lists, trees, and hash tables

Lists : The conceptually simplest and easiest-to-implement data structure for a symbol table is the linear list of records depicted. We use a single array, or equivalently several arrays to store names and their associated information. New names are added to the list in the order in which they are encountered. To retrieve information about a name, we search from the beginning of the array upto to the position marked by pointer AVAILABLE which indicates the begining of the empty portion of the array.

Search Trees : A more efficient approach to symbol-table organization is to add two link fields, LEFT and RIGHT to each record. We use these fields to link the records into a binary search tree. This tree has the property that all names NAME *j* accessible from NAME *i* by following the link LEFT *i* and then following any sequence of links will precede NAME *i* in alphabetical order (symbolically, NAME *j* < NAME *i*). Similarly, all names NAME *K* accessible starting with RIGHT *i* will have the property that Name < Name *K* >

Algorithm to look for NAME in a binary search tree, where P is initially a pointer to the root.

(1) while P ≠ null do

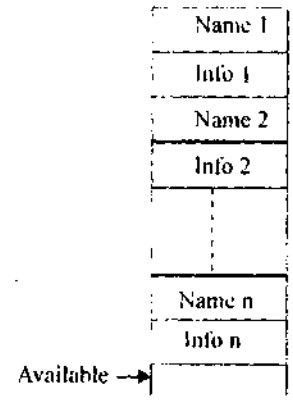
(2) if NAME = Name (P) then - - -/* NAME found, take action on success */

(3) else if Name < Name (P) then P : LEFT (P)

/* visit left child */

(4) else/* Name (P) < Name */ P: = RIGHT (P)

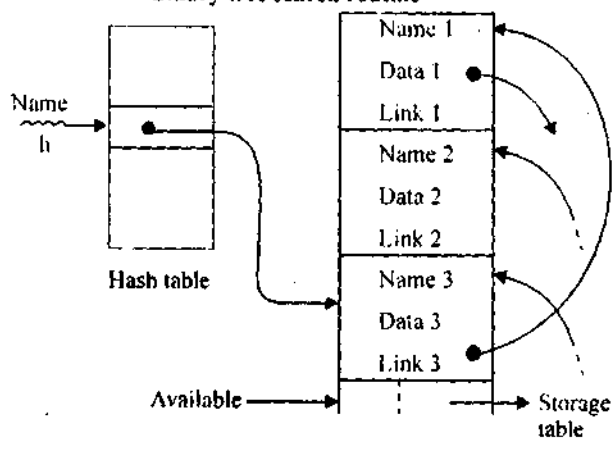
/ * if we fail through the loop, we have failed to find Name */



A linear list of records

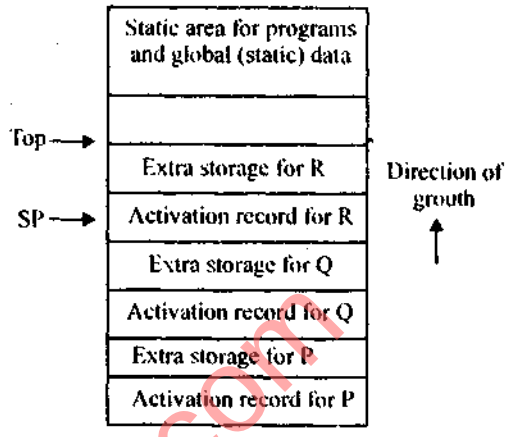
Hash Tables : Many variations of the important searching techniques known as hashing have been implemented in compilers. Here we shall consider a rather simple variant. Even this scheme gives us the capability of performing m accesses on n names in time proportional to n (n + m)/k. This method is generally superior to linear lists or search trees and is the method of choice for symbol tables in most situations, especially if storage is not particularly costly.

Binary tree search routine



Q. 4. (b) Explain the Implementation of simple stack allocation scheme while Run-Time administration.

Ans. Implementation of A simple Stack-Allocation Scheme : As an introduction to stack allocation, we are going to consider an implementation of the UNIX programming language, while allows somewhat simpler implementation than some other stack-oriented languages like ALGOL.



Data in C can be global, meaning it is allocated static storage and available to any procedure, or local, meaning it can be accessed only by the procedure in which it is declared. A program consists of a list of global data declarations and procedures; there is no block structure or nesting or procedures. However, recursion is permitted, so local names must be allocated space on a stack.

Starting from the highest numbered available memory location is the run-time stack shows a procedure P, which has called procedure Q which, in turn, has called procedure R.

We show two pointers to the stack, which are actually permanently allocated registers. One, called the stack pointer (SP), always points to a particular position in the activation resource for the currently active procedure. The second, called TOP, always points to the top of the stack.

Q. 4. (c) Explain the following. Give examples also :

(i) Lexical phase errors (ii) Syntatic phase errors.

Ans. Lexical -Phase Errors : The function of the lexical analyzer is to carve the steam of characters constituting the source program into a sequence of tokens. Each token class has a specification which is typically a regular set. If, after some processing, the lexical analyzer discovers that no prefix of the remaining input fits the specification of any token class, it can invoke an error-recovery routine that can take a variety of remedial actions.

Unfortunately, there is no one remedial action that will ideally suit all situations the problem of recovering from lexical errors is further hampered by the lack of redundancy at the lexical level of a language. The simplest expedient is to skip erroneous characters until the lexical analyzer can find another token.This action will likely cause the parser to see a deletion error.

Syntatic-Phase Errors : Often much of the eorror detection and recovery in a compiler is centered around syntax analysis. One reason is the high degree of precision we can achieve in the syntatic specification of programming languages using context-free grammars. From a grammar we can generate a parser that recognizes exactly the language specified by that grammar. Violations of the syntatic specification will be caught automatically by the parser. Although a considerable amount of theoretical and practical effort has been expended in exploring recovery and repair techniques for syntatic errors, the optimal strategy for any programming language is still an open question .

A parser detects an error when it has no legal move from its current configuration, which is determined by its state, stack contents and the current input symbol.To recover from an error a parser should ideally locate the position of the error. correct the error, revise its current configuration, and resume parsing. All existing methods approximate this ideal and will resume parsing, but there is never a guarantee that the error has been successfully corrected.

Q. 5. Attempt any two parts of the following : **10 × 2 = 20**

(a) Explain the following in the organization of the code optimizer :

(i) Control flow analysis (ii) Data flow analysis (iii) Transformations.

Ans. Control flow analysis : The translation of source statements that effect the flow of control in a program we discuss what code to generate for unconditional jumps and for "structured" flow-of-control constructs such as if then and while statements. We fix our attention on the generation of quadruples and the notation regarding translation filed names and list-handling procedures from that section carriers over to this section as well.

(ii) Data flow analysis : In order to the code optimization and a good job of code generation, a compiler needs to collect information about the program as whole and to distribute this information to each block in the flow graph. For example, we need to know that variables are live on exist from each block could improve register usage and how could we use knowledge of global common sub expressions to eliminate redundant ampitations and we also need to know how a compiler could take advantage of reading definitions, such as knowing where a variable like debug was last defined before reaching a given block, in order to perform transformations like constant folding and dead-lock elimination. These facts are the data-flow information that an optimizing compiler, collects by a process known as data-flow analysis.

Data-flow information can be collected by a program setting up and solving systems of information or equations that relate information at various points in a program. A typical equation has the form :

out [S] = gen[S] U (in [S]-kill [S])

and can be read as, "the information at the end of a statements is either generated within the statement, or enters at the beginning and is not killed as control flows through the statements". Such statements are called data-flow equations.

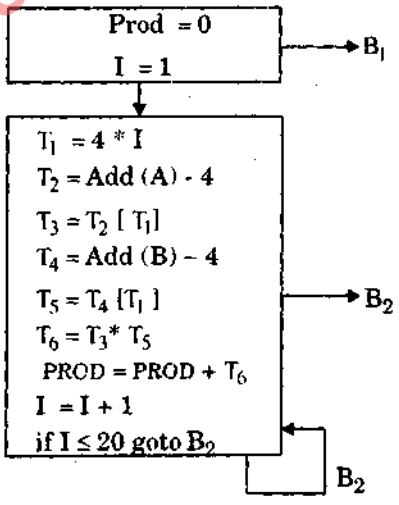
Transformations : Transformations are like a translator that takes as input a program written in one programming language (the source language) and produces as output a program in another language (the object or target language).

Other transform a programming language into a simplified language, called intermediate code, which can be directly executed using a program called an interpreter.

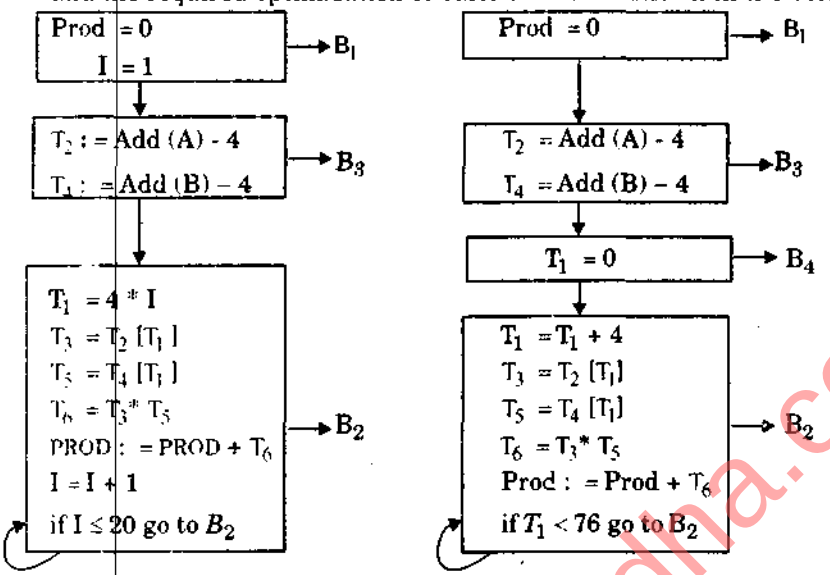
So, there are basically the transformation which are used in compiler.

Q. 5. (b) Explain the optimization of basic blocks. Also explain the DAG representation of basic blocks.

Ans. Optimization of basic blocks : Firstly basic block B_1, B_2 and flow graph is shown below



and the required optimization of basic blocks are shown in the below figures.



Now, **the DAG** representation of basis block :

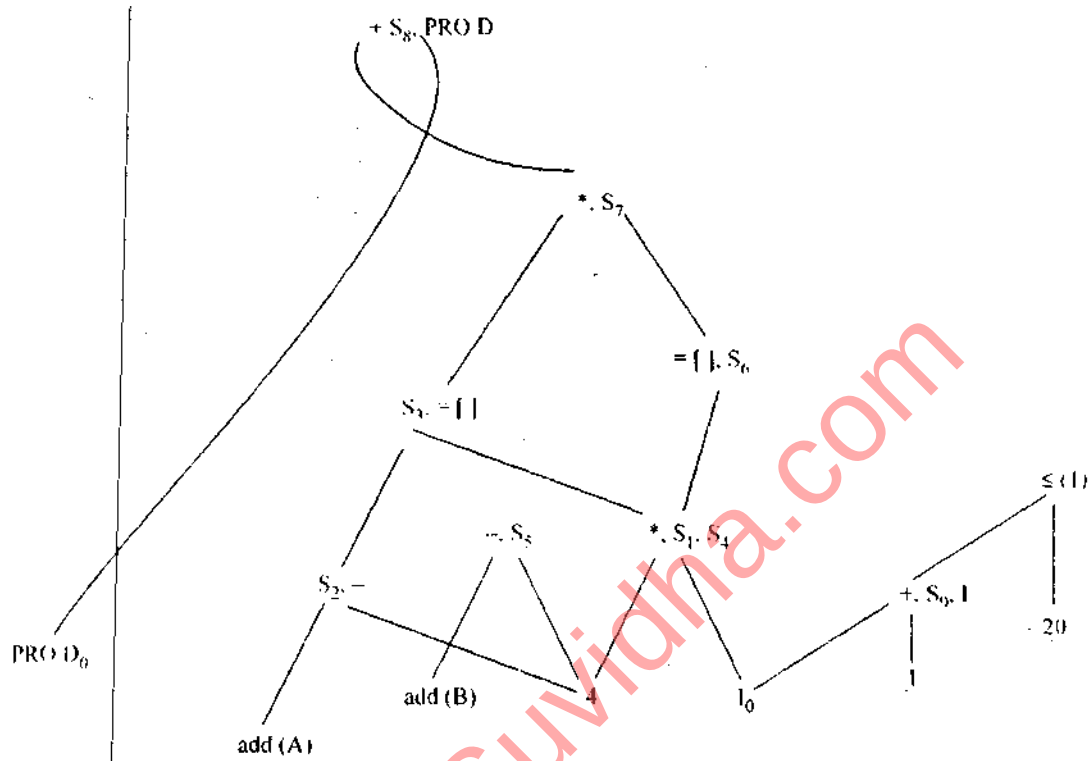
Directed acyclic graphs are useful data structures for implementing transformation on basic block. A dag gives a picture of how the value computed by each statement in a basic block is used in subsequent statements of the block. Constructing a dag from three address statements is a good way of determining common subexpressions within a block, determining which names are used inside the block but evaluated outside the block, and determining which statements of the blocks could have their computed values used outside the block.

A dag for a basic block is a directed acyclic graph with the following lobes on the nodes :

1. Leaves are labelled with identifiers either, variable names or contants from the operator applied to a name whether the 1-value or r-value of a name is needed; most leaves represent r-values.
2. Interior nodes are labelled by an operator symbol.
3. Nodes are also optionally given a sequence of identifiers for labels. The intention is that interior nodes represent computed values, and the identifiers labelling a node are deemed to have that value.

The three address code representation are :

(1) $S_1 = 4 * I$
(2) $S_2 = add\ (A) - 4$
(3) $S_3 = S_2[S_1]$
(4) $S_4 = 4 * I$
(5) $S_5 = add\ (B) - 4$
(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) $S_9 = I + I$



(11) $I = S_9$

(12) If $I \leq 20$ go to (1)

Now, the DAG representation of three address code.

Q. 5. (c) Explain what constitutes a loop in a flow graph and how will you do loop optimizations in the code optimization of a compiler.

Ans. It is useful to make a picture the basic blocks and their successor relationships by a directed graph called a flow graph. The nodes of the flow graph are the basic blocks. One node is distinguished as initial; it is the block whose leader is the first statement. There is a directed edge from block B to block B_2 could immediately follow, B during execution, that is, if

(1) there is a conditional or unconditional jump from the last statement of B_1 to the first statement of B_2 or,

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

When we perform code optimization, however, we may move quadruples from block to block to create new blocks, so an extension of the quadruple array is necessary if the quadruples in each block are to be kept in consecutive storage. An alternative is to make a linked list of the quadruples in each block.

The running time of a program may be improved if we decrease the length of one of its loops, especially an inner loop, even if we increase the amount of code outside the loops.