APJ ABDUL KALAM TECHNOLOGICAL UNIVERSITY

Sixth Semester B.Tech Degree Regular Examination

Course Code: CST302

Course Name: COMPILER DESIGN

Max. Marks: 100 **Duration: 3 Hour**

Answer Key

PART-A

Answer All Questions. Each Question Carries 3 Marks

1. Specify the analysis and synthesis parts of compilation.

Ans: The process of compilation is split up into following phases:

Analysis Phase-Analysis Phase performs 4 actions namely:

a. Lexical analysis

b. Syntax Analysis

c. Semantic analysis

d. Intermediate Code Generation

Synthesis Phase performs 2 actions namely:

a. Code Optimization

b. Code Generation

2. Define the terms token, lexemes and patterns with examples.

Token: It is basically a sequence of characters that are treated as a unit as it cannot be further broken down. In programming languages like C language- keywords (int, char, float, const, goto, continue, etc.) identifiers (user-defined names), operators (+, -, *, /), delimiters/punctuators like comma (,), semicolon(;), braces ({ }), etc. , strings can be considered as tokens

Lexeme

It is a sequence of characters in the source code that are matched by given predefined language rules for every lexeme to be specified as a valid token.

Example:
main is lexeme of type identifier(token)

(,),{,} are lexemes of type punctuation(token)

Pattern

It specifies a set of rules that a scanner follows to create a token.

3. Is the grammar S --> S | (S) S / Ɛ ambiguous? Justify your answer
Assume the string w=()(). We get the following parse trees when we derive the string

a) S-->S(S)S
→(S)S
---> ()S
--->()S(S)S
→()(S)S
→() () S
→()()

b) S-->S(S)S
→S(S)S(S)S
→(S)S(S)S
→()S(S)S
→()(S)S
→()()S
→()()

Since we have derived two leftmost derivations the given grammar is ambiguous

4. What is left recursive grammar? Give an example. What are the steps in removing left recursion

A grammar is said to be left –recursive if it has a non-terminal A such that there is a derivation A □□Aα, for some string α.

Top–down parsing methods cannot handle left recursive grammars, so a transformation that eliminates left-recursion is needed. The left-recursive pair of productions **A □□Aα|β** could be replaced by two non-recursive productions.

$$A \rightarrow \beta A'$$

$$A' \rightarrow \alpha A' \mid \varepsilon$$

Consider the following grammar and eliminate left recursion-

E → E + T / T

T → T x F / F

F → id

The grammar after eliminating left recursion is-

E → TE’

E’ → +TE’ / ∈

T → FT’

T’ → xFT’ / ∈

F → id

5. Compare different bottom-up parsing techniques

SLR Parser	**LALR Parser**	**CLR Parser**
It is very easy and cheap to implement.	It is also easy and cheap to implement.	It is expensive and difficult to implement.
SLR Parser is the smallest in size.	LALR and SLR have the same size. As they have less number of states.	CLR Parser is the largest. As the number of states is very large.
Error detection is not immediate in SLR.	Error detection is not immediate in LALR.	Error detection can be done immediately in CLR Parser.
SLR fails to produce a parsing table for a certain class of grammars.	It is intermediate in power between SLR and CLR i.e., SLR ≤ LALR ≤ CLR.	It is very powerful and works on a large class of grammar.
It requires less time and space complexity.	It requires more time and space complexity.	It also requires more time and space complexity.

6. What are the possible actions of a shift reduce parser.

A shift-reduce parser can possibly make the following four actions-

1. Shift-

In a shift action,

- The next symbol is shifted onto the top of the stack.

2. Reduce-

In a reduce action, the handle appearing on the stack top is replaced with appropriate non-terminal symbol

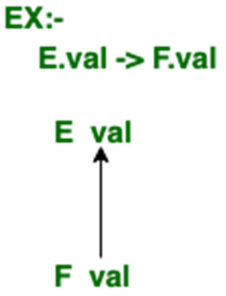
3.Accept: In accept action the parser announces successful completion of parsing

4. Error: it invokes error recovery routine

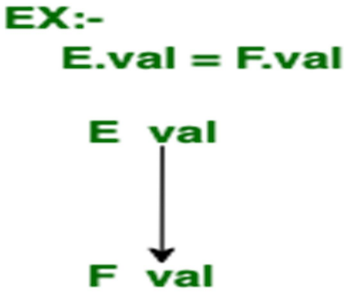
7. Differentiate synthesized and inherited attributes with examples.

In Syntax Directed Definition, two attributes are used one is Synthesized attribute and another is inherited attribute. An attribute is said to be **Synthesized attribute** if its parse tree node value is determined by the attribute value at child nodes whereas An attribute is said to be **Inherited attribute** if its parse tree node value is determined by the attribute value at parent and/or siblings node.

Synthesized attribute



Inherited attribute



8. Translate a[i] = b * c – b * d, to quadruple

t1=b*c
t2=b*d
t3=t1-t2
a[i]=t3

.

op	Arg1	Arg2	result
*	b	c	T1
*	b	d	T2
-	T1	T2	T3
[]=	a	i	T3

9. What is the role of peephole optimization in the compilation process

Peephole optimization is a simple and effective technique for locally improving target code. This technique is applied to improve the performance of the target program by examining the short sequence of target instructions (called the peephole) and replace these instructions replacing by shorter or faster sequence whenever possible. Peephole is a small, moving window on the target program.

10. What are the issues in the design of a code generator

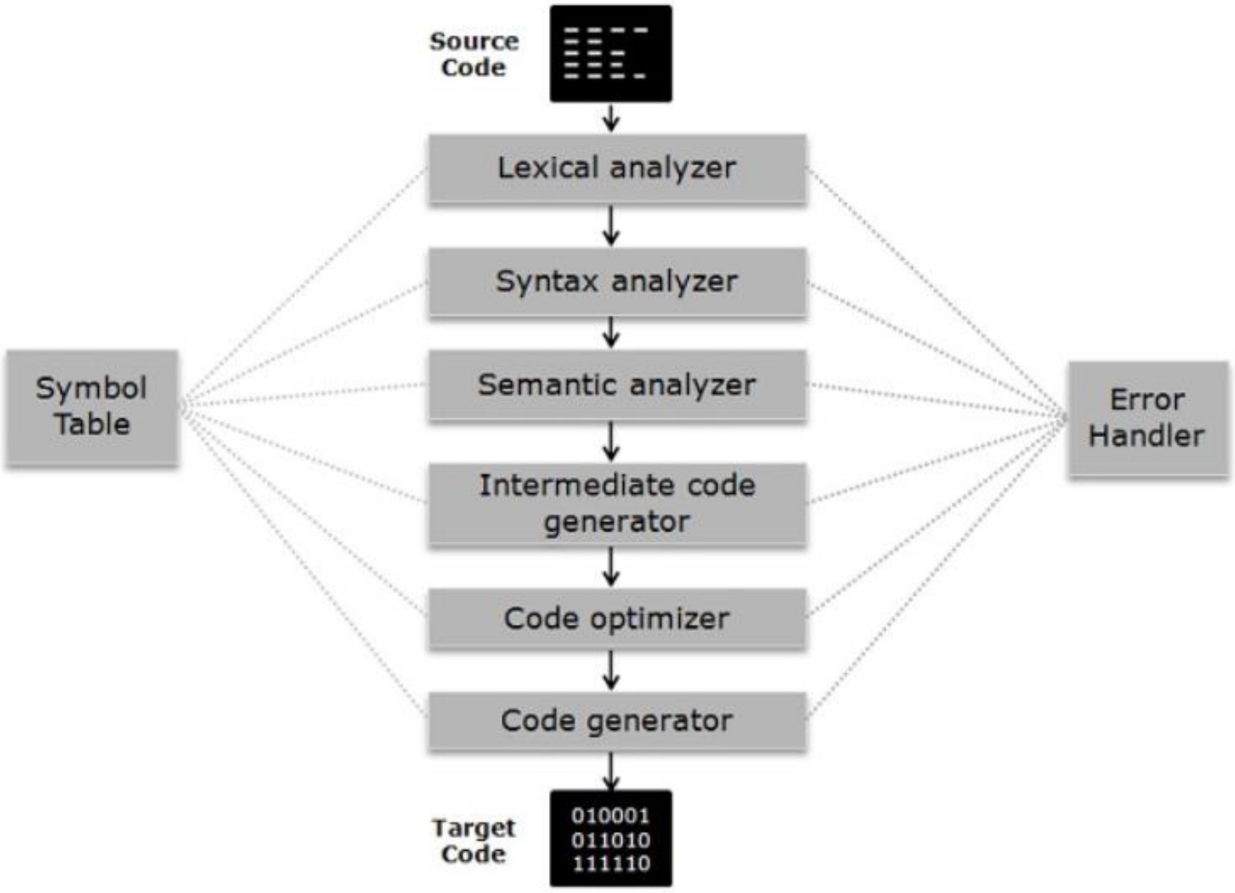
The following issues arise during the code generation phase:

1.Input to code generator
2. Target program
3. Memory management
4. Instruction selection
5. Register allocation
6. Evaluation order
7. Approaches to code generation

Part B

(Answer any one question from each module. Each question carries 14 Marks)

11.(a) Explain the different phases of a compiler with a running example.



Lexical Analysis

The first phase of a compiler is called lexical analysis or scanning.

The lexical analyzer reads the stream of characters making up the source program and groups the characters into meaningful sequences called lexemes.

For each lexeme, the lexical analyzer produces as output a token of the form

< token- name, attribute-value >
that it passes on to the subsequent phase, syntax analysis.
In the token, the first component token- name is an abstract symbol that is used during syntax analysis, and the second component attribute-value points to an entry in the symbol table for this token.

Information from the symbol-table entry 'is needed for semantic analysis and code generation.

For example, suppose a source program contains the assignment statement

position = initial + rate * 60
The characters in this assignment could be grouped into the following lexemes and mapped into the following tokens passed on to the syntax analyzer:

1. position is a lexeme that would be mapped into a token <id, 1>, where id is an abstract symbol standing for identifier and 1 points to the symbol table entry for position. The symbol- table entry for an identifier holds information about the identifier, such as its name and type.

2. The assignment symbol = is a lexeme that is mapped into the token < = >. Since this token needs no attribute-value, we have omitted the second component.

3. initial is a lexeme that is mapped into the token < id, 2> , where 2 points to the symbol-table entry for initial .

4. + is a lexeme that is mapped into the token <+>.

5. rate is a lexeme that is mapped into the token < id, 3 >, where 3 points to the symbol-table entry for rate.

6. * is a lexeme that is mapped into the token <* > .

7. 60 is a lexeme that is mapped into the token <60>

Blanks separating the lexemes would be discarded by the lexical analyzer. The representation of the assignment statement position = initial + rate * 60 after lexical analysis as the sequence of tokens as:

< id, l > < = > <id, 2> <+> <id, 3> < * > <60>

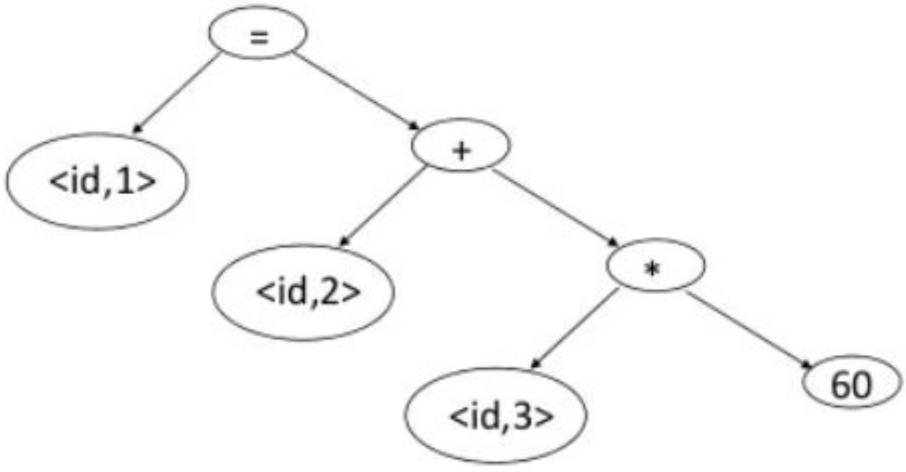
Syntax Analysis

The second phase of the compiler is syntax analysis or parsing.

The parser uses the first components of the tokens produced by the lexical analyzer to create a tree-like intermediate representation that depicts the grammatical structure of the token stream.

A typical representation is a syntax tree in which each interior node represents an operation and the children of the node represent the arguments of the operation.

The syntax tree for above token stream is:



The tree has an interior node labeled with (id, 3) as its left child and the integer 60 as its right child.

The node (id, 3) represents the identifier rate.

The node labeled * makes it explicit that we must first multiply the value of rate by 60.

The node labeled + indicates that we must add the result of this multiplication to the value of initial.

The root of the tree, labeled =, indicates that we must store the result of this addition into the location for the identifier position.

Semantic Analysis

The semantic analyzer uses the syntax tree and the information in the symbol table to check the source program for semantic consistency with the language definition.

It also gathers type information and saves it in either the syntax tree or the symbol table, for subsequent use during intermediate-code generation.

An important part of semantic analysis is type checking, where the compiler checks that each operator has matching operands.

For example, many programming language definitions require an array index to be an integer; the compiler must report an error if a floating-point number is used to index an array.

Some sort of type conversion is also done by the semantic analyzer.

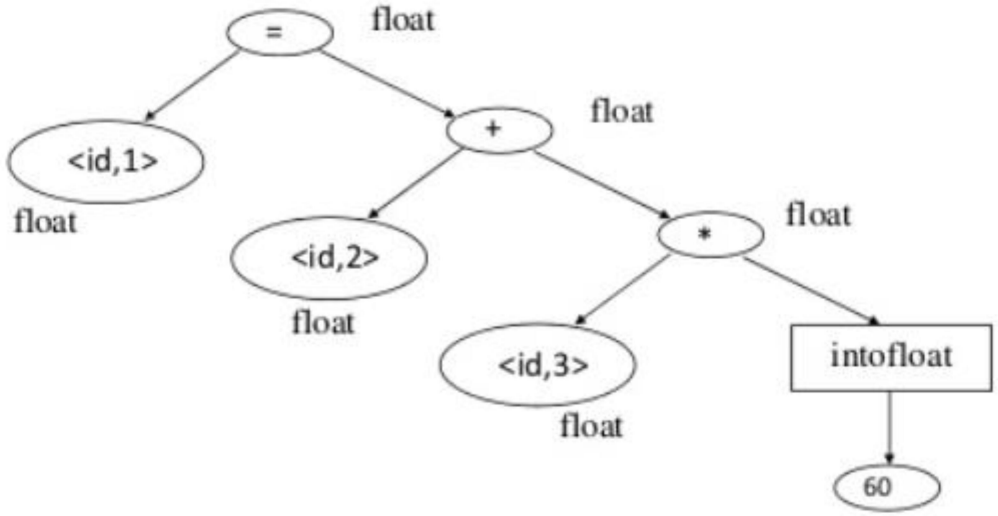
For example, if the operator is applied to a floating point number and an integer, the compiler may convert the integer into a floating point number.

In our example, suppose that position, initial, and rate have been declared to be floating- point numbers, and that the lexeme 60 by itself forms an integer.

The semantic analyzer discovers that the operator * is applied to a floating-point number rate and an integer 60.

In this case, the integer may be converted into a floating-point number.

In the following figure, notice that the output of the semantic analyzer has an extra node for the operator inttofloat , which explicitly converts its integer argument into a floating-point number



Intermediate Code Generation

In the process of translating a source program into target code, a compiler may construct one or more intermediate representations, which can have a variety of forms.

Syntax trees are a form of intermediate representation; they are commonly used during syntax and semantic analysis.

After syntax and semantic analysis of the source program, many compilers generate an explicit low-level or machine-like intermediate representation, which we can think of as a program for an abstract machine.

In our example, the intermediate representation used is three-address code, which consists of a sequence of assembly-like instructions with three operands per instruction.

In our example, the intermediate representation used is three-address code, which consists of a sequence of assembly-like instructions with three operands per instruction.

t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3

Code Optimization

The machine-independent code-optimization phase attempts to improve the intermediate code so that better target code will result.

The objectives for performing optimization are: faster execution, shorter code, or target code that consumes less power.

In our example, the optimized code is:

t1 = id3 * 60.0
id1 = id2 + t1

Code Generator

The code generator takes as input an intermediate representation of the source program and maps it into the target language.

If the target language is machine code, registers or memory locations are selected for each of the variables used by the program.

Then, the intermediate instructions are translated into sequences of machine instructions that perform the same task

A crucial aspect of code generation is the judicious assignment of registers to hold variables.

If the target language is assembly language, this phase generates the assembly code as its output.

In our example, the code generated is:
LDF R2, id3
MULF R2, #60.0
LDF R1, id2
ADDF R1, R2
STF id1, R1
The first operand of each instruction specifies a destination.

The F in each instruction tells us that it deals with floating-point numbers.

The above code loads the contents of address id3 into register R2, then multiplies it with floating-point constant 60.0.

The # signifies that 60.0 is to be treated as an immediate constant.

The third instruction moves id2 into register Rl and the fourth adds to it the value previously computed in register R2.

Finally, the value in register Rl is stored into the address of idl , so the code correctly implements the assignment statement **position = initial + rate * 60.**

Symbol Table

An essential function of a compiler is to record the variable names used in the source program and collect information about various attributes of each name.

These attributes may provide information about the storage allocated for a name, its type, its scope (where in the program its value may be used), and in the case of procedure names, such things as the number and types of its arguments, the method of passing each argument (for example, by value or by reference), and the type returned.

The symbol table is a data structure containing a record for each variable name, with fields for the attributes of the name.

The data structure should be designed to allow the compiler to find the record for each name quickly and to store or retrieve data from that record quickly.

Error Detection And Reporting

Each phase can encounter errors.

However, after detecting an error, a phase must somehow deal with that error, so that compilation can proceed, allowing further errors in the source program to be detected

b) List and explain any three compiler construction tools.

Parser Generators.

Input : Grammatical description of a programming language

Output : Syntax analyzers.

These produce syntax analyzers, normally from input that is based on a context-free grammar.

In early compilers, syntax analysis consumed not only a large fraction of the running time of a compiler, but a large fraction of the intellectual effort of writing a compiler.

This phase is one of the easiest to implement.

Scanner Generators

Input : Regular expression description of the tokens of a language

Output : Lexical analyzers.

These automatically generate lexical analyzers, normally from a specificalion based on regular expressions.

The basic organization of the resulting lexical analyzer is in effect a finite automaton.

Syntax-directed Translation Engines
Input : Parse tree.
Output : Intermediate code.
These produce collections of routines that walk the parse tree, generating intermediate code. The basic idea is that one or more "translations" are associated with each node of the parse tree, and each translation is defined in terms of translations at its neighbour nodes in the tree

12. (a) What is a regular definition? Give the regular definition of an unsigned integer

If Σ is an alphabet of basic symbols, then a regular definition is a sequence of definitions of the form
d1 → r1
d2 → r2
…
dn → rn

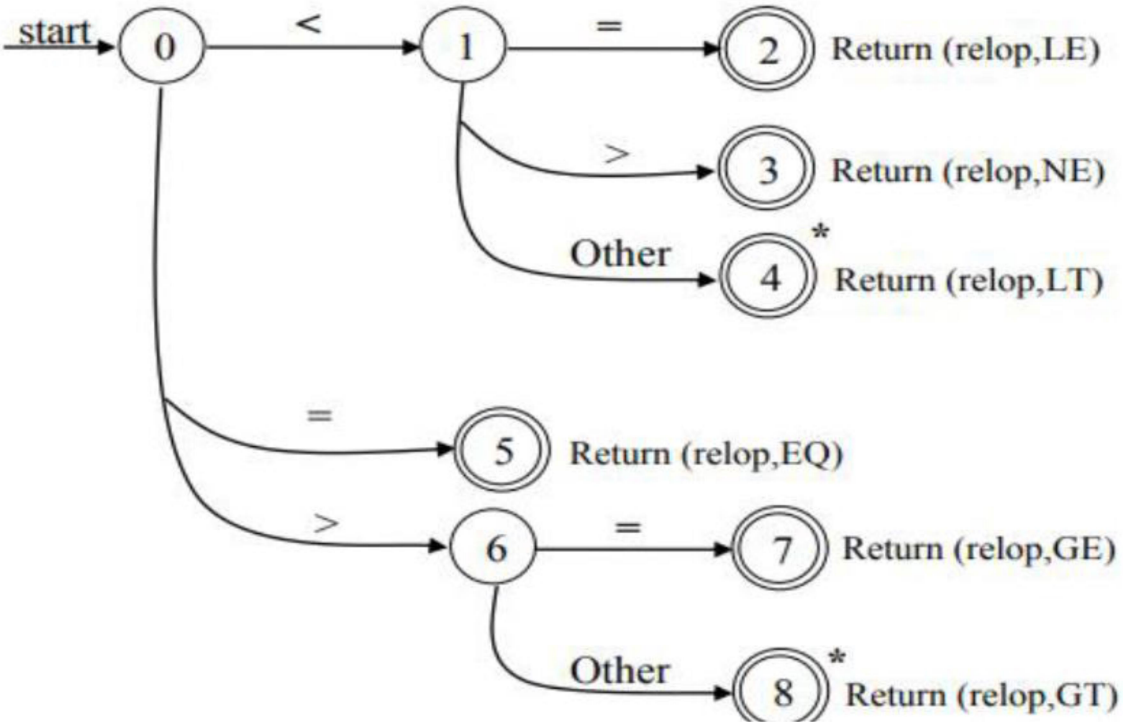
Unsigned numbers (integer or floating point) are strings such as 5280, 0.01234, 6.336E4, or 1.89E-4. The regular definition
– digit → 0|1|2 |… | 9
– digits → digit digit*
– optionalFraction → .digits | ε
– optionalExponent →(E(+ |- | ε) digits) | ε
– number → digits optionalFraction optionalExponent

b) Express the role of transition diagrams in recognition of tokens.

As an intermediate step in the construction of a lexical analyzer, we first produce a flowchart, called a r diagram. Transition diagrams.

Transition diagram depict the actions that take place when a lexical analyzer is called by the parser to get the next token.

Example: A Transition Diagram for the token relation operators "**relop**" is shown in Figure below:



. 13. (a) What is Recursive Descent parsing? List the challenges in designing such a parser?

It is the most general form of top-down parsing. It may involve **backtracking**, that is making repeated scans of input, to obtain the correct expansion of the leftmost non-terminal. Unless the grammar is ambiguous or left- recursive, it finds a suitable parse tree

For implementing a recursive descent parser for a grammar.

The grammar must not be left recursive.

The grammar must be left factored that means it should not have common prefixes for alternates

We need a language that has recursion facility

13 (b) Consider the following grammar

E-→E or T | T
T→ T and F |F
F→not F | (E) | true | false

(i) Remove left recursion from the grammar.

(ii) Construct a predictive parsing table. (iii) Justify the statement "The grammar is LL (1)".

(i)After removing left recursion

E→TE'

E'→or TE'|Ɛ

T→FT'

T'→and FT'|Ɛ

F→not F|(E)|true|false

(ii) Compute FIRST and FOLLOW to construct predictive parsing table

Non terminals	FIRST	FOLLOW
E	{not,(,true, false}	{),$}
E'	{or, Ɛ}	{),$}
T	{not,(,true, false}	{or,),$}
T'	{and, Ɛ}	{or,),$}
F	{not,(,true, false}	{and, or,),$}

PREDICTIVE PARSING TABLE

Non terminals	not	(	)	True	false	and	or	$
E	E→TE'	E→TE'		E→TE'	E→TE'			
E'			E'→Ɛ				E'→ orTE'	E'→Ɛ
T	T→FT'	T→FT'		T→FT'	T→FT'			
T'			T'→Ɛ			T'→and FT'	T'→ Ɛ	T'→Ɛ
F	F→not F	F→(E)		F→true	F→false			

Since the predictive parsing table contains no multiple entries, the given grammar is in LL(1)

14.a) What is Recursive Descent parsing? List the problems in designing such a parser

It is the most general form of top-down parsing.

It may involve **backtracking**, that is making repeated scans of input, to obtain the correct expansion of the leftmost non-terminal. Unless the grammar is ambiguous or left-recursive, it finds a suitable parse tree

For implementing a recursive descent parser for a grammar.

The grammar must not be left recursive

The grammar must be left factored that means it should not have common prefixes for alternates

We need a language that has recursion facility

b) Design a recursive descent parser for the grammar S→cAd, A→ab/ b

Consider the grammar:

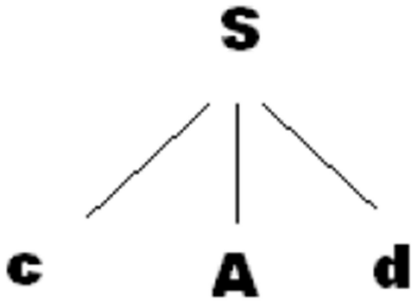
S → cAd

A → ab | a

and the input string w = cad.

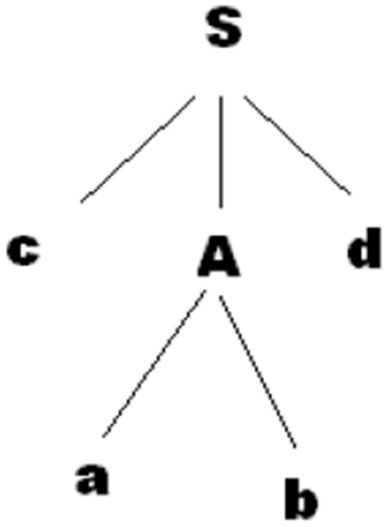
 To construct a parse tree for this string top down, we initially create a tree consisting of a single node labelled **S**.

 An input pointer points to **c**, the first symbol of w. **S** has only one production, so we use it to expand **S** and obtain the tree as:



The leftmost leaf, labeled **c**, matches the first symbol of input w, so we advance the input pointer to **a**, the second symbol of **w**, and consider the next leaf, labeled **A**.

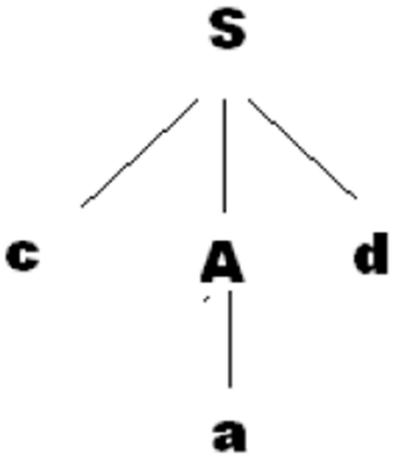
 Now, we expand **A** using the first alternative **A → ab** to obtain the tree



We have a match for the second input symbol, **a**, so we advance the input pointer to **d**, the third input symbol, and compare d against the next leaf, labeled **b**.

 Since **b** does not match **d**, we report failure and go back to **A** to see whether there is another alternative for **A** that has not been tried, but that might produce a match.

 In going back to **A**, we must reset the input pointer to position 2 , the position it had when we first came to **A**, which means that the procedure for **A** must store the input pointer in a local variable



The leaf **a** matches the second symbol of **w** and the leaf **d** matches the third symbol. Since we have produced a parse tree for **w**, we halt and announce successful completion of parsing. (that is the string parsed completely and the parser stops).

 The leaf a matches the second symbol of w and the leaf d matches the third symbol. Since we have produced a parse tree for w, we halt and announce successful completion of parsing. (that is the string parsed completely and the parser stops).

Find the FIRST and FOLLOW of the non-terminals S, A and B in the grammar (5 marks)
S→aABe
A→Abc | b
B→d

Non terminals	FIRST	FOLLOW
S	a	$
A	b	d
B	d	e

15. (a) Construct the LR(0) set of items and their GOTO function for the grammar
S→S S + | S S * | a

Step1: Write augmented grammar

S'→S

S→S S +

S→S S *

S→a

Construction of LR(0) items

I_0 : S'→.S

S→.S S +

S→.S S *

S→.a

GOTO(I_0, S)

I_1 :S→S.

S→S. S +

S→.SS+

S→.SS*

S→.a

S→S. S *

GOTO(I_0, a)

I2: S→a.

GOTO(I_1, S)

I3: S→S S. +

S→S.S+

S→.SS+

S→.SS*

S→.a

S→S.S*

S→SS.*

GOTO(I_1, a)

Same as I2

Sinc I2 is a final item

Apply GOTO(I3,+)

I4:S→SS+.

GOTO(I3,S)

Same as I3

GOTO(I3,a)

Same as I2

GOTO(I3,*)

I4:S→SS*.

b)To construct parsing table find FOLLOW of S

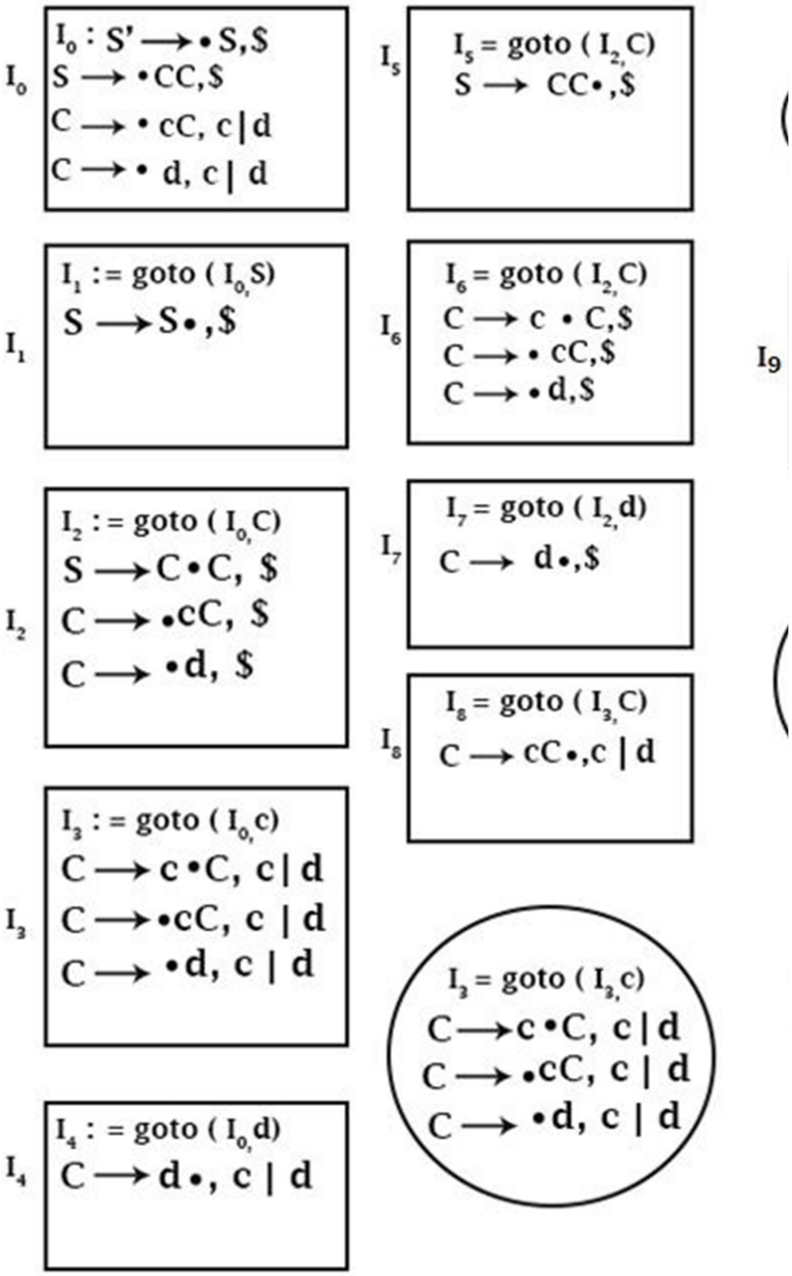
FOLLOW(S)={$,+,*}

	ACTION				GOTO
	+	*	a	$	S
0			s2		1
1			s2	accept	3
2	R3	R3		R3	
3	S4	S5	S2		3
4	R1	R1		R1	
5	R2	R2		R2	

16. a) Identify LR(1) items for the grammar **(7)**

S→ CC

C→ cC | d



b) In these states, states I_3 and I_6 can be merged because they have the same core or first component but a different second component of Look Ahead.

Similarly, states I_4 and I_7 are the same.

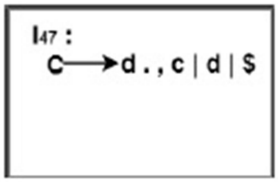
Similarly, states I_8 and I_9 are the same.

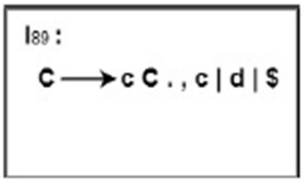
So, I_3 and I_6 can be combined to make I_{36}.

I_4 and I_7 combined to make I_{47}.

I_8 and I_9 combined to make I_{89}.

So, the states will be





Construction of LALR Parsing Table

Filling of "shift" Entries(s)

Consider goto(I_0, c) = I_{36}

∴ Action[0, c] = s36

∴ Write s36 in front of Row state 0 and column c.

Similarly, consider

goto(I_2, d) = I_{47}

∴ Action[2, d] = 47

∴ Write s47 in front of Row State 2 and column d.

Filling the "reduce" Entries (r)

Consider productions of the form A → α · ,

For example, Consider State

I_{47} = goto(I_0, d)

C → d ·, c |d |$

∴ C → d ·, c |d |$ is of form A → α · , a.

Since C → d is production number (3) in given Question.

∴ Write r3 in front of Row State 47 and column c, d, $.

Because c, d looks ahead symbols in production C → d · , c | d.

Filling of goto Entries

It can found out only for Non-Terminal.

For example, Consider

goto(I_0, S) = I_1

∴ goto [0, S] = 1

Filling of "Accept" Entry

ince, S′ → S · , $ is in I_1

∴ Write accept in front of Row state 1 & column $.

LALR Parsing table can also be obtained by merging the rows of combined states of CLR parsing, i.e., Merge Row corresponding to 3, 6, then 4, 7 and then 8, 9.

The resulting LALR Parsing table will be −

State	Action			goto	
	c	d	$	S	C
0	s36	s47		1	2
1			accept		
2	s36	s47			5
36	s36	s47			89
47	r3	r3	r3		
5			r1		
89	r2	r2	r2		

17 a) Design a Syntax Directed Translator(SDT) for the arithmetic expression (4 * 7 + 1) * 2 and draw an annotated parse tree for the same.

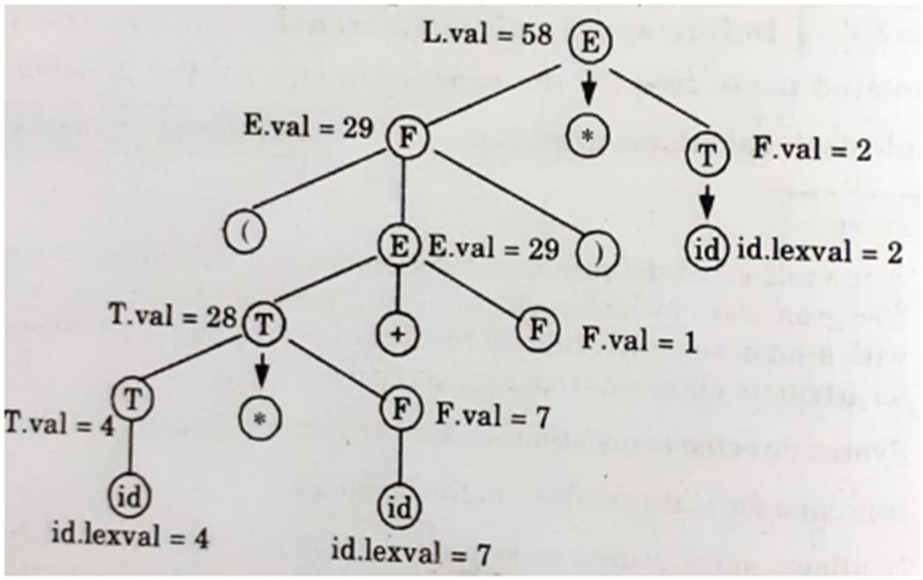
E -> E+T { E.val = E.val + T.val }

E -> T { E.val = T.val }

T -> T*F { T.val = T.val * F.val }

T -> F { T.val = F.val }

F -> INTLIT { F.val = INTLIT.lexval }



b) Consider the grammar with following translation rules and E as the start symbol

E → E1 # T {E.value=E1.value x T.value ;}

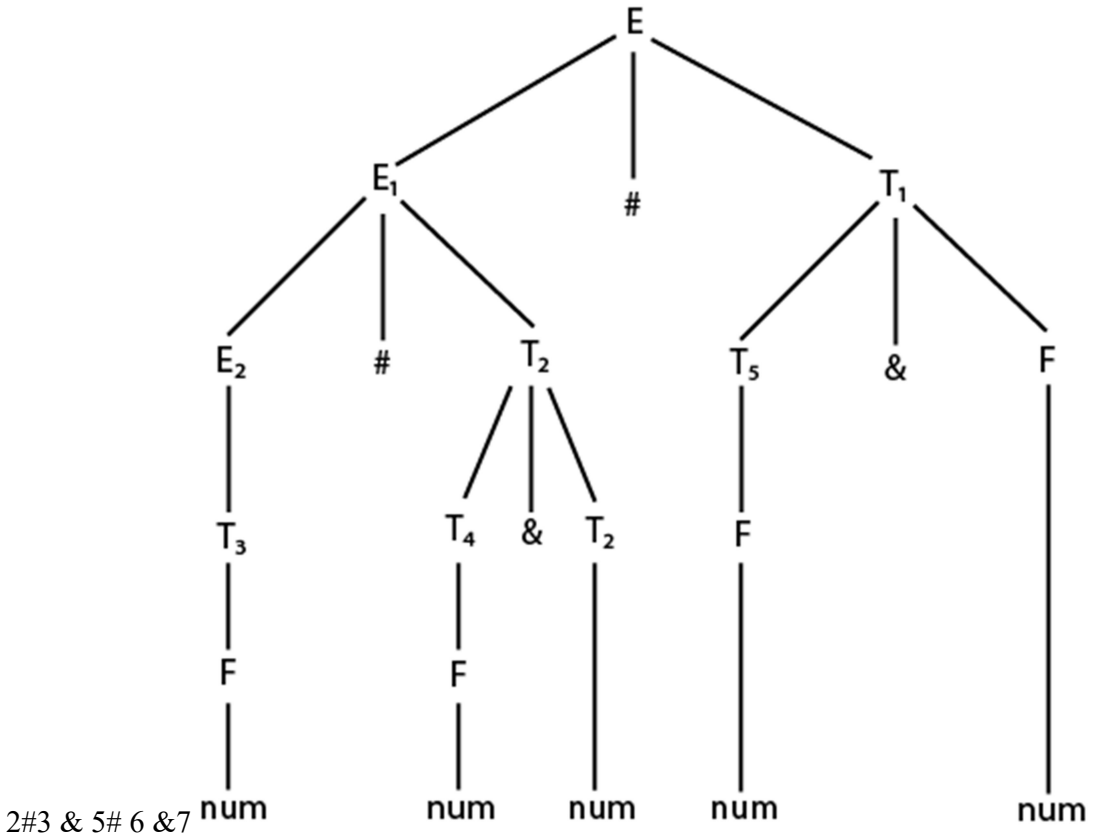
| T{E.value=T.value ;}

T → T1 & F{ T.value=T1.value + F.value ;}

| F{T.value= F.value ; }

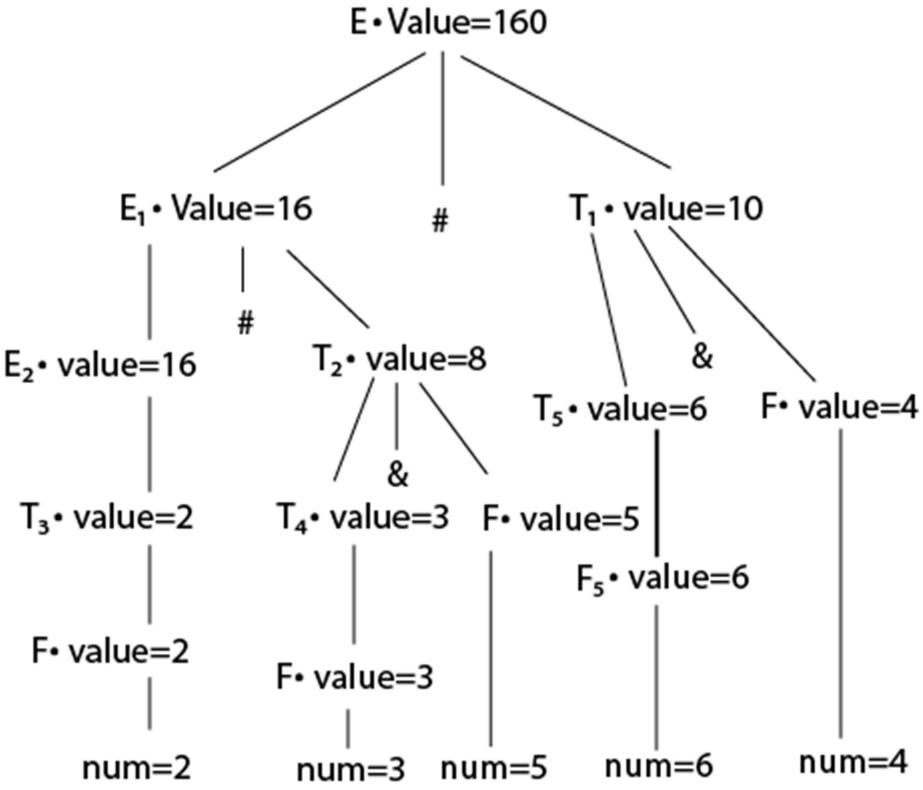
F → num { F.value=num. lvalue ;}

Compute E.value for the root of the parse tree for the expression



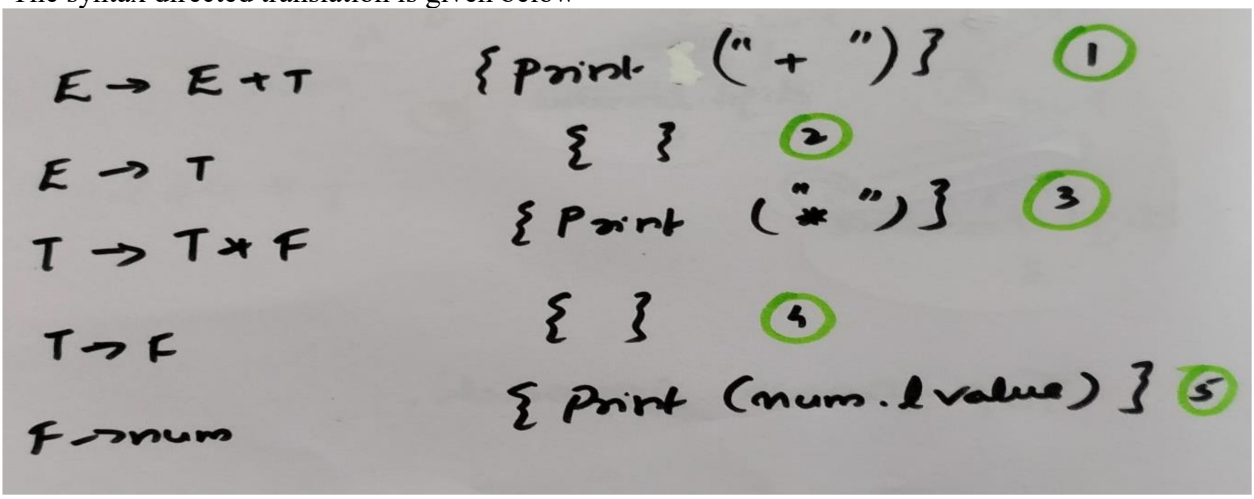
2#3 & 5# 6 &7

hen we construct the annonated parse tree or parse tree with value at the leaf node.

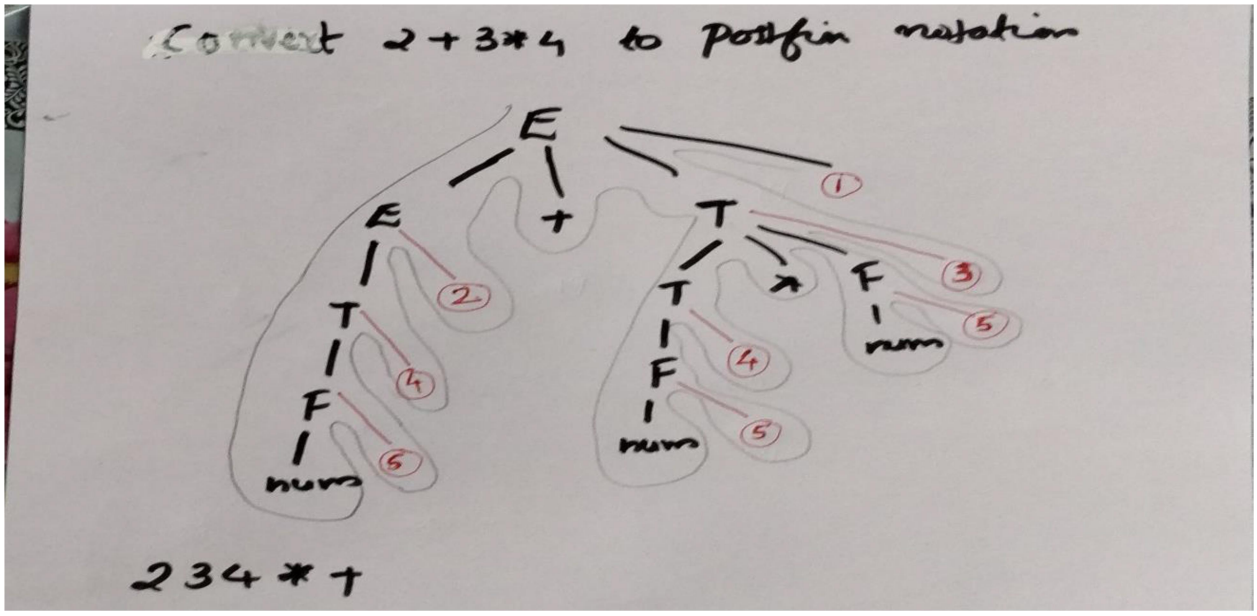


18. (a) Write Syntax Directed Translator (SDT) and parse tree for infix to postfix translation of an expression

The syntax directed translation is given below



Infix to postfix conversion is illustrated below



b) Explain the storage allocation strategies

1)**Static Allocation** In static allocation, names are bound to storage as the program is compiled, so there is no need for a run-time support package. Since the bindings do not change at runtime, every time a procedure activated, its run-time, names bounded to the same storage location. Therefore, values of local names retained across activations of a procedure. That is when control returns to a procedure the value of the local are the same as they were when control left the last time. From the type of a name, the compiler decides amount of storage for the name and decides where the activation records go. At compile time, we can fill in the address at which the target code can find the data it operates on

2.)**Stack Allocation :**Storage is organized as a stack. Each time a procedure called, space for its local variables is pushed onto a stack, and when the procedure terminates, space popped off from the stack .Locals are bound to fresh storage in each activation. The values of locals are deleted when each activation ends

- 3) **Heap Allocation :** Stack allocation strategy cannot be used if either of the following is possible : 1. The values of local names must be retained when an activation ends.

  2. A called activation outlives the caller.

  Heap allocation parcels out pieces of contiguous storage, as needed for activation records or other objects.Pieces may be deallocated in any order, so over the time the heap will consist of alternate areas that are free and in use

19. (a) Describe the principal sources of optimization

A 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

There are a number of ways in which a compiler can improve a program without changing the function it computes.

Function preserving transformations examples:

Common sub expression elimination
Copy propagation,
Dead-code elimination
Constant folding

The other transformations come up primarily when global optimizations are performed.

Frequently, a program will include several calculations of the offset in an array. Some of the duplicate calculations cannot be avoided by the programmer because they lie below the level of detail accessible within the source language.

***

Common Sub expressions 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.

- For example

t1: = 4*i
t2: = a [t1]
t3: = 4*j
t4: = 4*i
t5: = n
t6: = b [t4] +t5

The above code can be optimized using the common sub-expression elimination as

t1: = 4*i
t2: = a [t1]
t3: = 4*j
t5: = n
t6: = b [t1] +t5

The common sub expression t4: =4*i is eliminated as its computation is already in t1 and the value of i is not been changed from definition to use.

Copy Propagation:

Assignments of the form f : = g called copy statements, or copies for short. The idea behind the copy-propagation transformation is to use g for f, whenever possible after the copy statement f: = g. Copy propagation means use of one variable instead of another. This may not appear to be an improvement, but as we shall see it gives us an opportunity to eliminate x.

• For example:
x=Pi;

A=x*r*r;

The optimization using copy propagation can be done as follows: A=Pi*r*r;

Here the variable x is eliminated

Dead-Code Eliminations:

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.

Example:

```
i=0;
if(i=1)
{
a=b+5;
}
```

Here, ‘if’ statement is dead code because this condition will never get satisfied.

Constant folding:

Deducing at compile time that the value of an expression is a constant and using the constant instead is known as constant folding. One advantage of copy propagation is that it often turns the copy statement into dead code.

For example,

a=3.14157/2 can be replaced by

a=1.570 there by eliminating a division operation.

Loop Optimizations:

In loops, especially in the inner loops, programs tend to spend the bulk of their time. The running time of a program may be improved if the number of instructions in an inner loop is decreased, even if we increase the amount of code outside that loop.

Three techniques are important for loop optimization:

- Ø Code motion, which moves code outside a loop;
- Ø Induction-variable elimination, which we apply to replace variables from inner loop.

- Ø Reduction in strength, which replaces and expensive operation by a cheaper one, such as a multiplication by an addition.

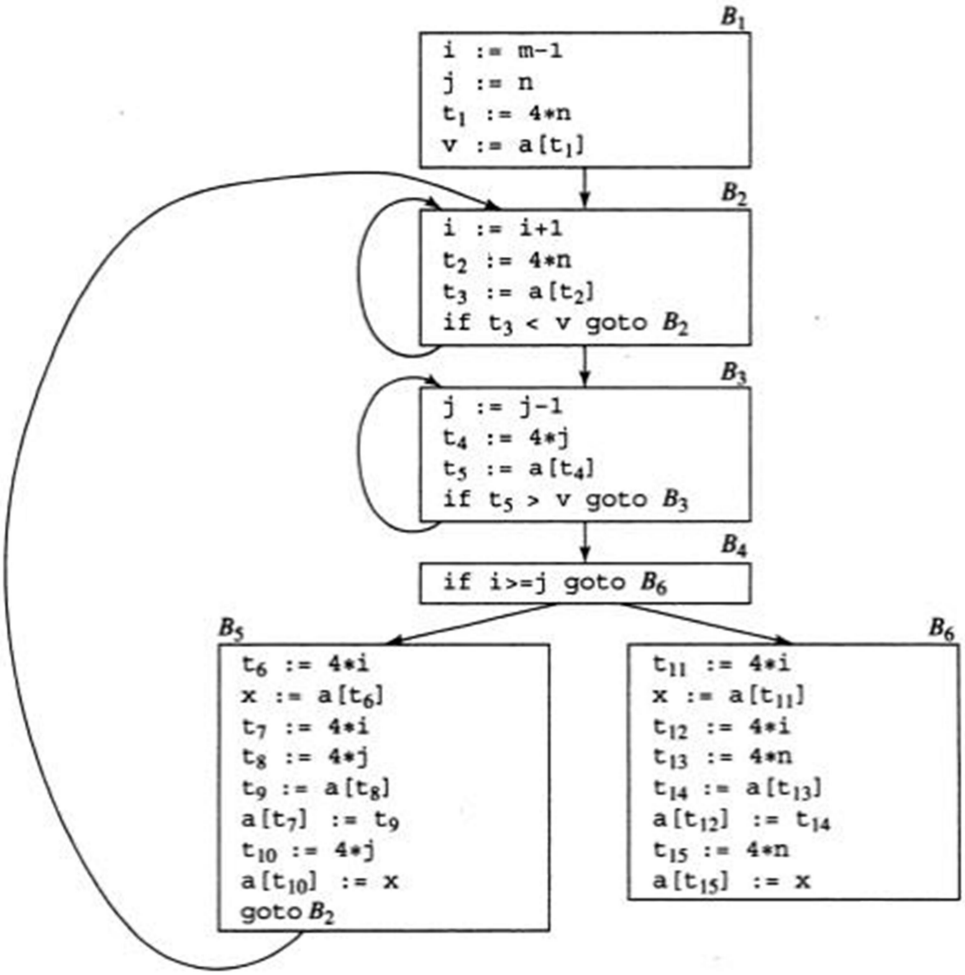


Fig. 5.2 Flow graph

Code Motion:

An important modification that decreases the amount of 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 */
```

Induction Variables :

Loops are usually processed inside out. For example consider the loop around B3. Note that the values of j and t4 remain in lock-step; every time the value of j decreases by 1, that of t4 decreases by 4 because 4*j is assigned to t4. 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 B3 in Fig.5.3 we cannot get rid of either j or t4 completely; t4 is used in B3 and j in B4.

However, we can illustrate reduction in strength and illustrate a part of the process of induction-variable elimination. Eventually j will be eliminated when the outer loop of B2- B5 is considered.

Example:

As the relationship t4:=4*j surely holds after such an assignment to t4 in Fig. and t4 is not changed elsewhere in the inner loop around B3, it follows that just after the statement j:=j-1 the relationship t4:= 4*j-4 must hold. We may therefore replace the assignment t4:= 4*j by t4:= t4-4. The only problem is that t4 does not have a value when we enter block B3 for the first time. Since we must maintain the relationship t4=4*j on entry to the block B3, we place an initializations of t4 at the end of the block where j itself is initialized, shown by the dashed addition to block B1 in Fig.5.3.

The replacement of a multiplication by a subtraction will speed up the object code if multiplication takes more time than addition or subtraction, as is the case on many machines.

Reduction In Strength:

Reduction in strength replaces expensive operations by equivalent cheaper ones on the target machine. Certain machine instructions are considerably cheaper than others and can often be used as special cases of more expensive operators. For example, x^2 is invariably cheaper to implement as x*x than as a call to an exponentiation routine. Fixed-point multiplication or division by a power of two is cheaper to implement as a shift. Floating-point division by a constant can be implemented as multiplication by a constant, which may be cheaper.

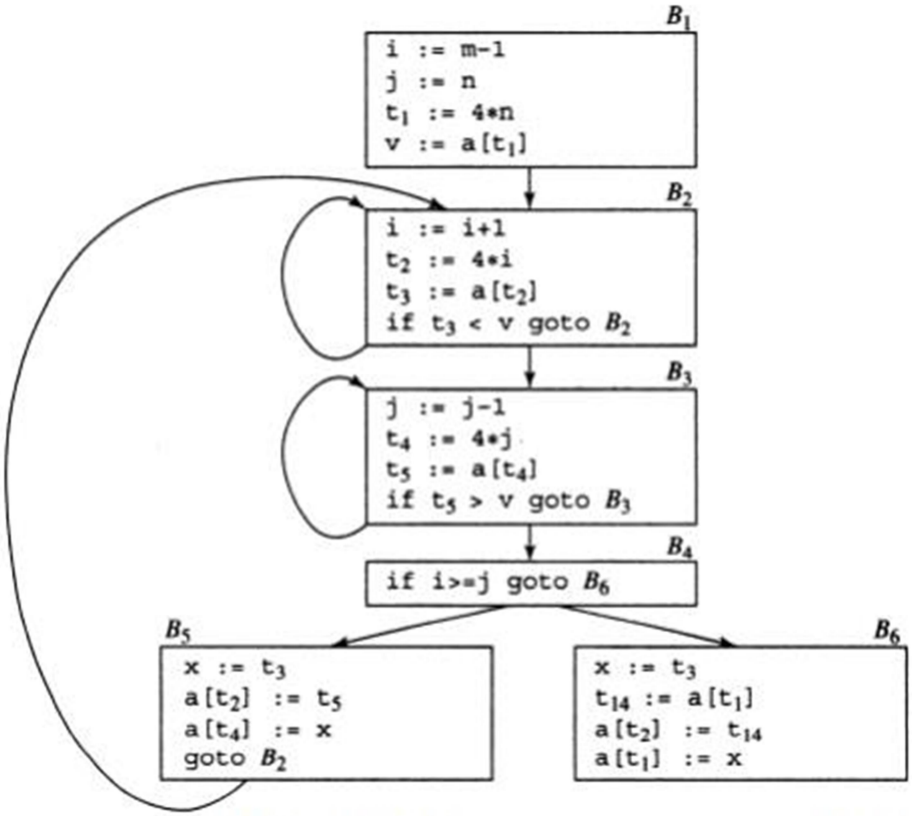


Fig. 5.3 B5 and B6 after common subexpression elimination

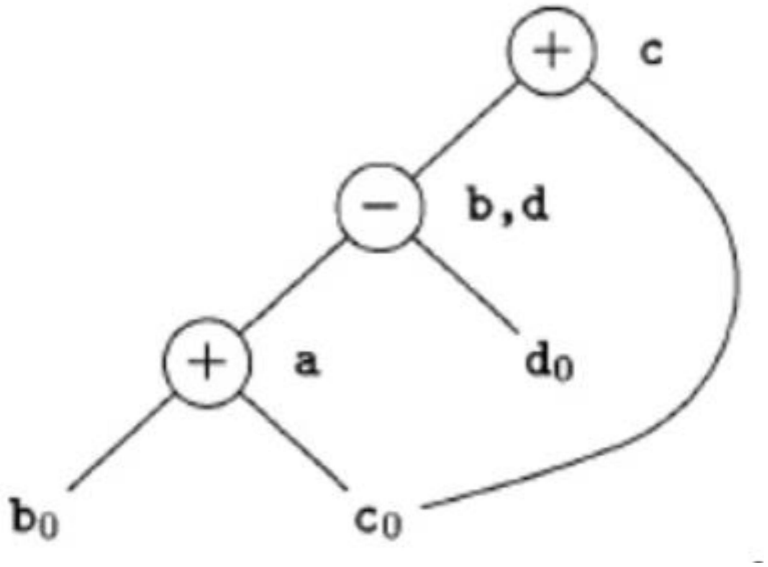
Fig. 5.3 B5 and B6 after common subexpression elimination

b) **Directed Acyclic Graph**

In compiler design, a DAG is an abstract syntax tree with a unique node for each value. DAG is an useful data structure for implementing transformation on basic block. DAG is constructed from three address code.

Common subexpression can be detected by noticing, as a new node m is about to added, whether there is an existing node n with the same children, in the same order, and with the same operator. If so, n computes the same value as m and may be used in its place.

```
a = b + c
b = a – d
c = b + c
d = a - d
```



20 .(a) Write the Code Generation Algorithm and explain the *getreg* function

The code generation algorithm uses descriptors to keep track of register contents and addresses for names.

1. **A Register Descriptor** keeps track of what is currently in each register. It is consulted whenever a new register is needed.

2. **An Address Descriptor** keeps track of the location where the current value of the name can be found at run time. The location might be a register, a stack location or a memory address. This information can be stores in the symbol table and is used to determine the accessing method for a name.

A code-generation algorithm

Code generation algorithm takes input as a sequence of three-address statements constituting a basic block. Statement of the form x = y op z performs the following actions.

1. Invoke a function getreg to determine the location L where the result of the computation y op z should be stored.

2. Consult the address descriptor for y to determine y', the current location of y. Prefer the register for y' if the value of y is currently both in memory and a register. If the value of y is not already in L, generate the instruction MOV y' , L to place a copy of y in L.

3. Generate the instruction OP z' , L where z' is a current location of z. Prefer a register to a memory location if z is in both. Update the address descriptor of x to indicate that x is in location L. If x is in L, update its descriptor and remove x from all other descriptors.

4. If the current values of y or z have no next uses, are not live on exit from the block, and are in registers, alter the register descriptor to indicate that, after execution of x : = y op z , those registers will no longer contain y or z

The Function *getreg* M

The function getreg returns the location L to hold the value of x for the assignment **x= y op z.**

1. If the name Y is in a register that holds the value of no other names and Y is not live and has no next use after X := Y op Z then return register of Y for L. Update the address descriptor of y to indicate that y is no longer in L.

2. Failing (1) return an empty register for L if there is one.

3. Failing (2) if X has a next use in the block or op is an operator, such as indexing that requires a register, find an occupied register R. Store the value of R into a memory location (by MOV R, M) If it is not already in proper memory location M, update the address descriptor for M, and return R. if R hold the value of several variables, a MOV instruction must be generated for each variable that need to be stored. A suitable occupied register might be one whose datum is referenced furthest in the future, or one whose value is also in memory. We leave the exact choice unspecified, since there is no one proven best way to make the selection.

4. If X is not used in the block. Or no suitable occupied register can be found, select the memory location of X as L.

(b) Generate target code sequence for the following statement

d := (a-b)+(a-c)+(a-c).

The assignment d : = (a-b) + (a-c) + (a-c) might be translated into the following three-address code sequence: **t := a – b**

u := a – c

v := t + u

d := v + u with d live at the end

Statements	**Code Generated**	**Register descriptor**	**Address descriptor**
		Register empty	
t : = a - b	MOV a, R_0 SUB b, R0	R_0 contains t	t in R_0
u : = a - c	MOV a , R1 SUB c , R1	R_0 contains t R1 contains u	t in R_0 u in R1
v : = t + u	ADD R_1, R_0	R_0 contains v R_1 contains u	u in R_1 v in R_0
d : = v + u	ADD R_1, R_0 MOV R_0, d	R_0 contains d	d in R_0 d in R_0 and memory