1. **How do SLR(1), LR(1) and LALR(1) methods compare against each other in the process of constructing the parsing table from the grammar?**

**SLR Parser**

The SLR parser is similar to LR(0) parser except that the reduced entry. The reduced productions are written only in the FOLLOW of the variable whose production is reduced.

**Construction of SLR parsing table –**

1. Construct C = { I0, I1, ……. In}, the collection of sets of LR(0) items for G’.
2. State i is constructed from Ii. The parsing actions for state i are determined as follow :
   * If [ A -> ?.a? ] is in Ii and GOTO(Ii , a) = Ij , then set ACTION[i, a] to “shift j”. Here a must be terminal.
   * If [A -> ?.] is in Ii, then set ACTION[i, a] to “reduce A -> ?” for all a in FOLLOW(A); here A may not be S’.
   * Is [S -> S.] is in Ii, then set action[i, $] to “accept”. If any conflicting actions are generated by the above rules we say that the grammar is not SLR.
3. The goto transitions for state i are constructed for all nonterminals A using the rule:  
   if GOTO( Ii , A ) = Ij then GOTO [i, A] = j.
4. All entries not defined by rules 2 and 3 are made error.

**CLR PARSER**

In the SLR method we were working with LR(0)) items. In CLR parsing we will be using LR(1) items. LR(k) item is defined to be an item using lookaheads of length k. So , the LR(1) item is comprised of two parts : the LR(0) item and the lookahead associated with the item.

LR(1) parsers are more powerful parser.

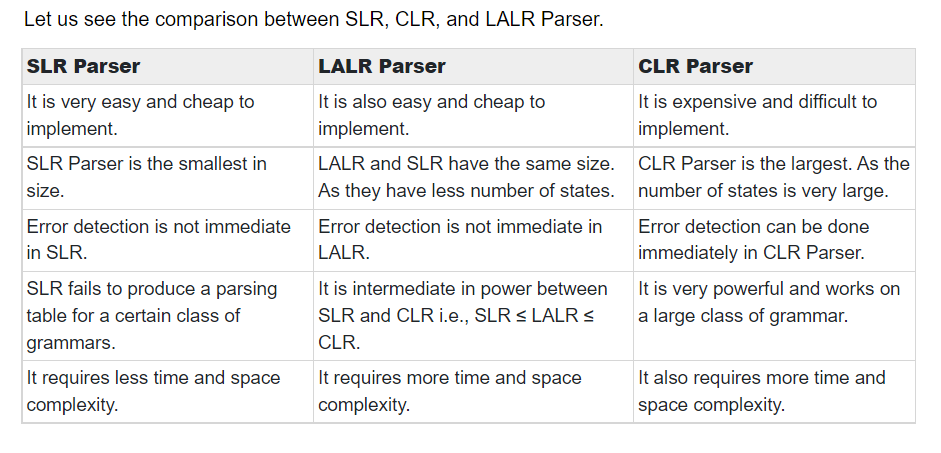
For LR(1) items we modify the Closure and GOTO function.

**Construction of CLR parsing table-**  
 Input – augmented grammar G’

1. Construct C = { I0, I1, ……. In} , the collection of sets of LR(0) items for G’.
2. State i is constructed from Ii. The parsing actions for state i are determined as follow :  
   i) If [ A -> ?.a?, b ] is in Ii and GOTO(Ii , a) = Ij, then set ACTION[i, a] to “shift j”. Here a must be terminal.  
   ii) If [A -> ?. , a] is in Ii , A ≠ S, then set ACTION[i, a] to “reduce A -> ?”.  
   iii) Is [S -> S. , $ ] is in Ii, then set action[i, $] to “accept”.  
   If any conflicting actions are generated by the above rules we say that the grammar is  
   not CLR.
3. The goto transitions for state i are constructed for all nonterminals A using the rule: if GOTO( Ii, A ) = Ij then GOTO [i, A] = j.
4. All entries not defined by rules 2 and 3 are made error.

**LALR PARSER**

LALR parser are same as CLR parser with one difference. In CLR parser if two states differ only in lookahead then we combine those states in LALR parser. After minimisation if the parsing table has no conflict that the grammar is LALR also.



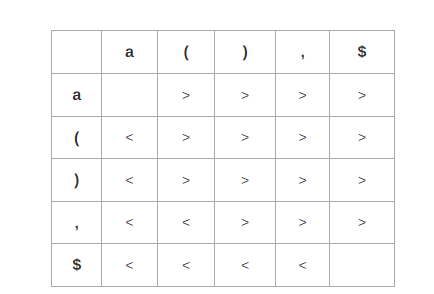
1. **Efficient code generation requires the Remember of internal architecture of the target machine. Justify your answer with an Example**

For target machines with several CPU registers, the code generator is responsible for register allocation. This means that the compiler must be aware

of which registers are being used for particular purposes in the generated program, and which become available as code is generated.

Book Page No. 507

1. **Consider the precedence table. Write the steps involved in handle pruning for the input (a,(a, a)).**

****

Ans is in copy page no. 1

1. **Consider the following grammar**

**S --> E**

**E --> E1 + T**

**E --> T**

**T --> T1 \* F**

**T --> F**

**F --> digit**

**Syntax Directed Definition (SDD)** is a kind of abstract specification. It is generalization of context free grammar in which each grammar production **X –> a**is associated with it a set of production rules of the form s = f(b1, b2, ……bk) where s is the attribute obtained from function f. The attribute can be a string, number, type or a memory location. Semantic rules are fragments of code which are embedded usually at the end of production and enclosed in curly braces ({ }).

**Example:**

E --> E1 + T { E.val = E1.val + T.val}

**Annotated Parse Tree –** The parse tree containing the values of attributes at each node for given input string is called annotated or decorated parse tree.

**Features –**

* High level specification
* Hides implementation details
* Explicit order of evaluation is not specified

The SDD for the above grammar can be written as follow



Let us assume an input string 4 \* 5 + 6 for computing synthesized attributes. The annotated parse tree for the input string is

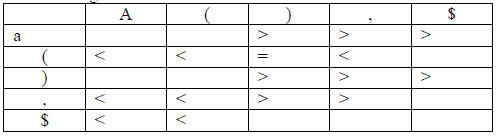


For computation of attributes we start from leftmost bottom node. The rule F –> digit is used to reduce digit to F and the value of digit is obtained from lexical analyzer which becomes value of F i.e. from semantic action F.val = digit.lexval. Hence, F.val = 4 and since T is parent node of F so, we get T.val = 4 from semantic action T.val = F.val. Then, for T –> T1 \* F production, the corresponding semantic action is T.val = T1.val \* F.val . Hence, T.val = 4 \* 5 = 20

Similarly, combination of E1.val + T.val becomes E.val i.e. E.val = E1.val + T.val = 26. Then, the production S –> E is applied to reduce E.val = 26 and semantic action associated with it prints the result E.val . Hence, the output will be 26.

1. **Draw the annotated parse tree to compute s-attribute for the input string 3 \* 5 +5**

**Draw the precedence function graph for the following table**

****

Ans in Note Copy page no 2

1. **Consider the following grammar:**

**E-> E+T|T**

**T-> TF|F**

**F-> F\*|a|b**

**Construct the SLR parsing table and also parse the input “a\*b+a”.**

Solution:

Step1 − Construct the augmented grammar and number the productions.

(0) E′ → E

(1) E → E + T

(2) E → T

(3) T → TF

(4) T → F

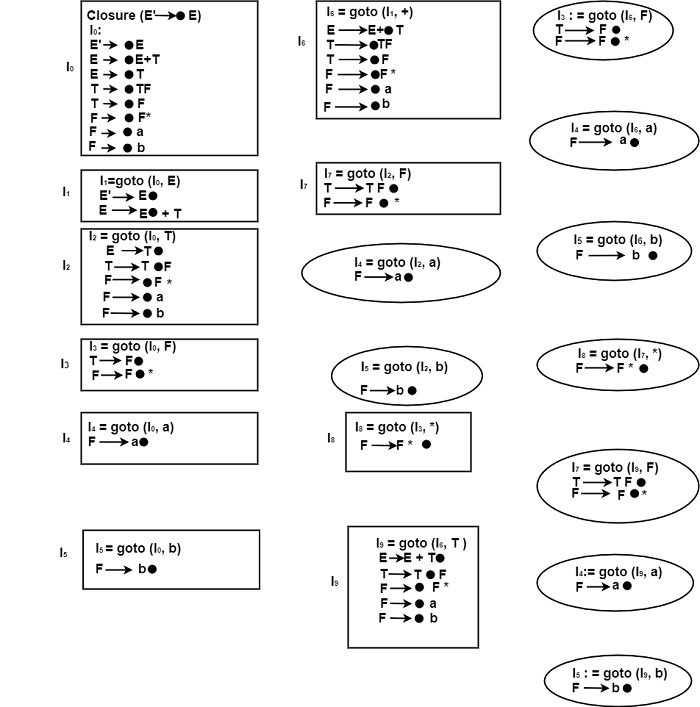
(5) F → F ∗

(6) F → a

(7) F → b.

Step2 − Find closure & goto Functions to construct LR (0) items.

Box represents the New states, and the circle represents the Repeating State.



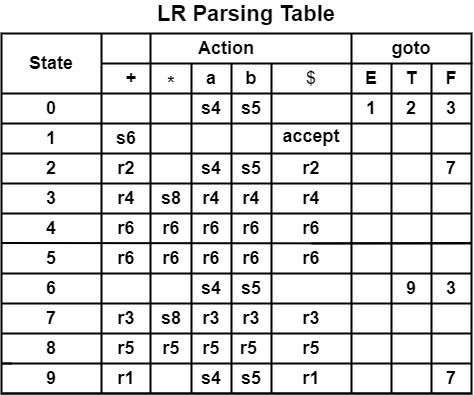
**Computation of FOLLOW**

We can find out

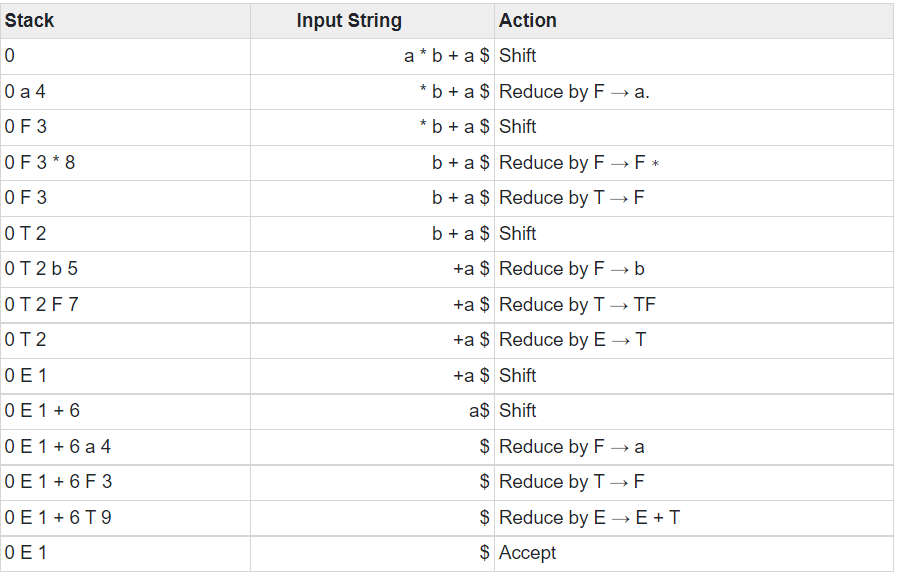
FOLLOW(E) = {+, $}

FOLLOW(T) = {+, a, b, $}

FOLLOW(F) = {+,\*, a, b, $}



**Parsing for Input String a \* b + a**



1. **Evaluate the execution state of the following C program**

**Main()**

**{**

**int i;**

**int a[10];**

**i = 1;**

**While(i <= 10)**

**{**

**a[i] = 0;**

**i = i + 1;**

**}**

**} Into A. Syntax tree. B.Postfix notation. C.3 address code.**

1. **Design an activation tree for the for the following function calls:**

**enter main()**

**enter readarray()**

**leave readarry()**

**enter quicksort(1,9)**

**enter partition(1,9)**

**leave partition(1,9)**

**enter quicksort(1,3)**

**…..**

**leave quicksort(1,3)**

**enter quicksort(5,9)**

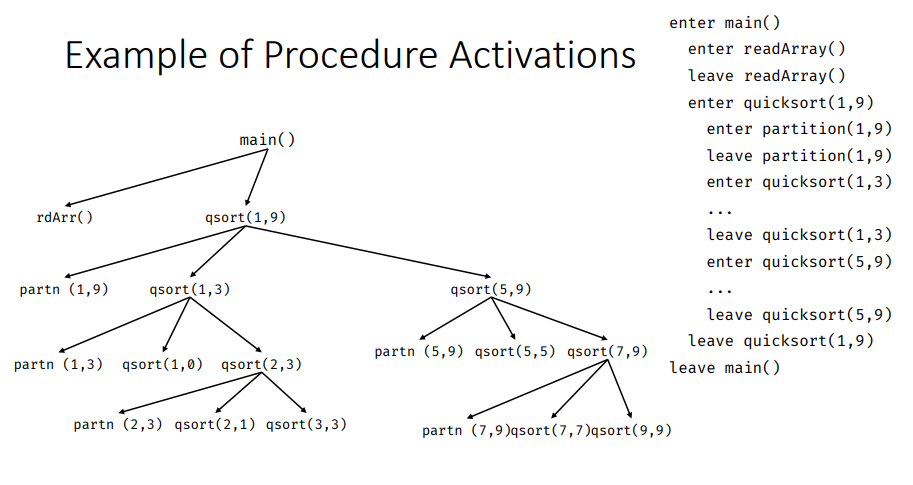
**…..**

**leave quicksort(5,9)**

**leave quicksort(1,9)**

**leave main()**

Solution

****

1. **Consider the grammar.**

**S → A a**

**S → b A c**

**S → d c**

**S → b d a**

**A → d**

1. **Parse input string "bdc" constructing LALR Parsing Table. Is the grammar LALR(1)?**

**Step1**− Construct Augmented Grammar

(0) S′ → S

(1) S → A a

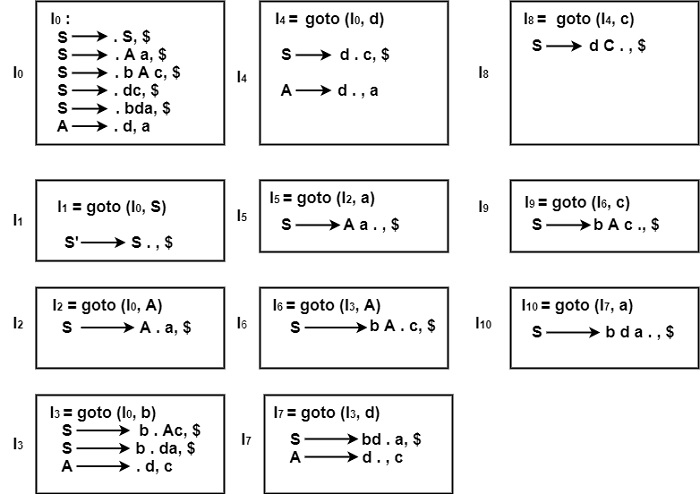
(2) S → b A c

(3) S → d c

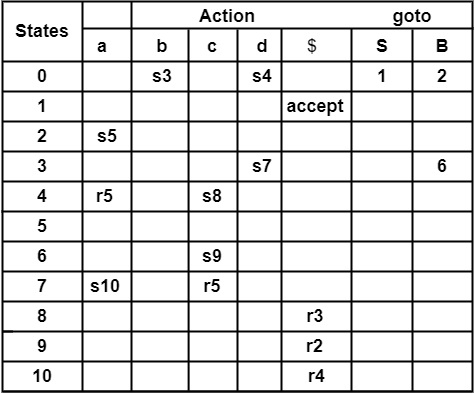
(4) S → b d a

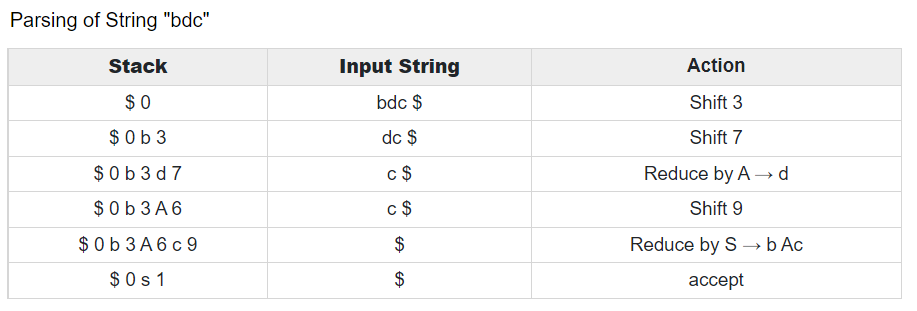
(5) A → d

**Step2**− Find Closure & goto. Find the canonical set of LR (1) items for the Grammar.



In the states, I0 to I10, no states have a similar first element or core. So, we cannot merge the states. Some states will be taken for building the LALR parsing table.



****

1. **For the following three address code, give the quadruple representation**

**t1 := -r**

**t2 := q\*t1**

**t3 := -r**

**t4 := s \* t3**

**t5 := t2 + t4**

**p := t5**

Ans in Note Copy

1. **Construct LALR(1) parsers for the following grammar and parse the sentence id=id**

**S → L = R**

**S → R**

**L → \* R**

**L → id**

**R → L**

Ans in Note Copy

1. **Only one occurrence of each object is allowable at a given moment during program execution. Justify your answer with respect to static allocation**

**Static Allocation**

* In this allocation scheme, the compilation data is bound to a fixed location in the memory and it does not change when the program executes.
* As the memory requirement and storage locations are known in advance, runtime support package for memory allocation and de-allocation is not required.
* In a static storage-allocation strategy, it is necessary to be able to decide at compile time exactly where each data object will reside at run time. In order to make such a decision, at least two criteria must be met:
  1. The size of each object must be known at compile time.
  2. **Only one occurrence of each object is allowable at a given moment during program execution.**
* Because of the first criterion, variable-length strings are disallowed, since their length cannot be established at compile time. Similarly dynamic arrays are disallowed, since their bounds are not known at compile time and hence the size of the data object is unknown.
* **Because of the second criterion, nested procedures are not possible in a static storage-allocation scheme. This is the case because it is not known at compile time which or how many nested procedures, and hence their local variables, will be active at execution time**.

1. **Design an activation tree for the Fibonacci sequence 1,1,2,3,5,8, ... defined by f(1)=f(2)=1 and, for n>2, f(n)=f(n-1)+f(n-2). Consider the function calls that result from a main program calling f(5).**

**system starts main**

**enter f(5)**

**enter f(4)**

**enter f(3)**

**enter f(2)**

**exit f(2)**

**enter f(1)**

**exit f(1)**

**exit f(3)**

**enter f(2)**

**exit f(2)**

**exit f(4)**

**enter f(3)**

**enter f(2)**

**exit f(2) s**

**enter f(1)**

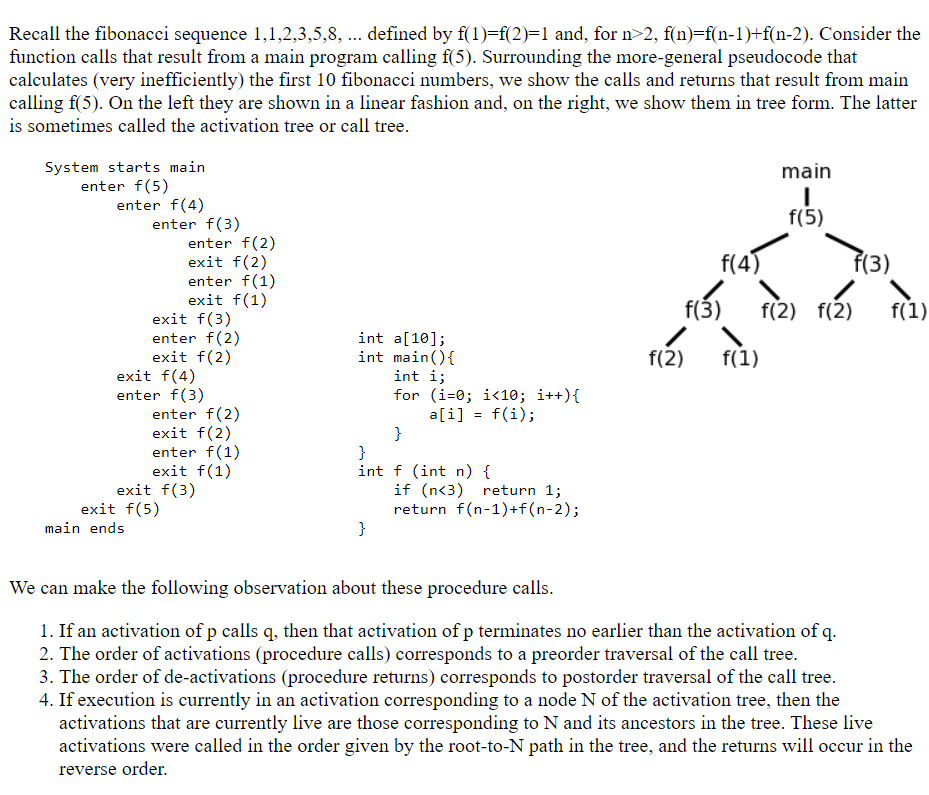
**exit f(1)**

**exit f(3)**

**exit f(5)**

**main ends**

Solution



1. **Consider the following grammar**

**S --> E**

**E --> E1 + T**

**E --> T**

**T --> T1 \* F**

**T --> F**

**F --> digit**

1. **Draw the annotated parse tree to compute s-attribute for the input string 4 \* 5 + 6**

**Syntax Directed Definition (SDD)** is a kind of abstract specification. It is generalization of context free grammar in which each grammar production **X –> a**is associated with it a set of production rules of the form s = f(b1, b2, ……bk) where s is the attribute obtained from function f. The attribute can be a string, number, type or a memory location. Semantic rules are fragments of code which are embedded usually at the end of production and enclosed in curly braces ({ }).

**Example:**

E --> E1 + T { E.val = E1.val + T.val}

**Annotated Parse Tree –** The parse tree containing the values of attributes at each node for given input string is called annotated or decorated parse tree.

**Features –**

* High level specification
* Hides implementation details
* Explicit order of evaluation is not specified

The SDD for the above grammar can be written as follow



Let us assume an input string 4 \* 5 + 6 for computing synthesized attributes. The annotated parse tree for the input string is



For computation of attributes we start from leftmost bottom node. The rule F –> digit is used to reduce digit to F and the value of digit is obtained from lexical analyzer which becomes value of F i.e. from semantic action F.val = digit.lexval. Hence, F.val = 4 and since T is parent node of F so, we get T.val = 4 from semantic action T.val = F.val. Then, for T –> T1 \* F production, the corresponding semantic action is T.val = T1.val \* F.val . Hence, T.val = 4 \* 5 = 20

Similarly, combination of E1.val + T.val becomes E.val i.e. E.val = E1.val + T.val = 26. Then, the production S –> E is applied to reduce E.val = 26 and semantic action associated with it prints the result E.val . Hence, the output will be 26.

**Write the program for dot product of two vectors. Optimize this code by Eliminating the common sub expressions.**

1. **Write the quadrapules, triples and indirect triples for the following C statements using its equivalent three address code.**

**a = b + 1**

**x = y+3**

**y = a/b**

**a = b+c**

**Implementation of Three Address Code –**

There are 3 representations of three address code namely

Quadruple

Triples

Indirect Triples

1. **Quadruple** –

It is structure with consist of 4 fields namely op, arg1, arg2 and result. op denotes the operator and arg1 and arg2 denotes the two operands and result is used to store the result of the expression.

**Advantage –**

Easy to rearrange code for global optimization.

One can quickly access value of temporary variables using symbol table.

**Disadvantage –**

Contain lot of temporaries.

Temporary variable creation increases time and space complexity.

**2. Triples –**

This representation doesn’t make use of extra temporary variable to represent a single operation instead when a reference to another triple’s value is needed, a pointer to that triple is used. So, it consist of only three fields namely op, arg1 and arg2.

Disadvantage –

* Temporaries are implicit and difficult to rearrange code.
* It is difficult to optimize because optimization involves moving intermediate code. When a triple is moved, any other triple referring to it must be updated also. With help of pointer one can directly access symbol table entry.

**3. Indirect Triples –**

This representation makes use of pointer to the listing of all references to computations which is made separately and stored. Its similar in utility as compared to quadruple representation but requires less space than it. Temporaries are implicit and easier to rearrange code.

**Problems in Note Copy**

1. **Which function creates space on the stack for each element?**

**int f(int n)**

**{**

**int t,s;**

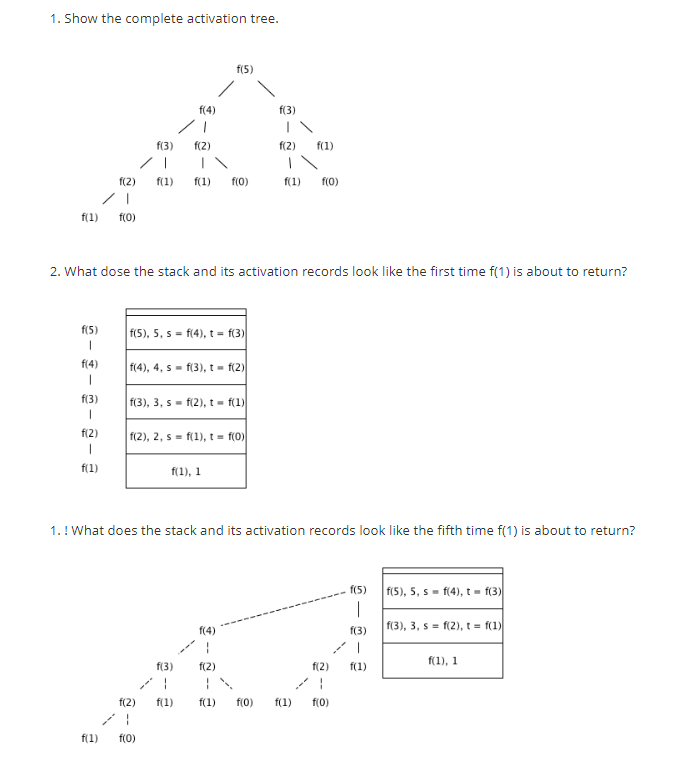
**if( n <2 ) return 1;**

**s= f(n-1);**

**t=f(n-2);**

**return s+t;**

**}**

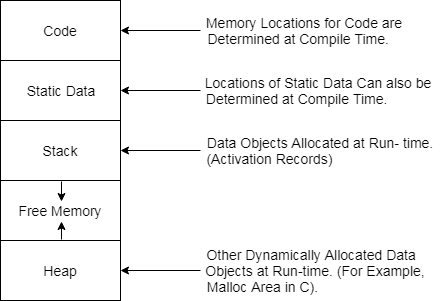


1. **How does the compiler manage and organize the logical address space? With a block diagram show how typical subdivision of run-time memory into code and data areas.**

When the target program executes then it runs in its own logical address space in which the value of each program has a location.

The logical address space is shared among the compiler, operating system and target machine for management and organization. The operating system is used to map the logical address into physical address which is usually spread throughout the memory.

Subdivision of Run-time Memory:



* Runtime storage comes into blocks, where a byte is used to show the smallest unit of addressable memory. Using the four bytes a machine word can form. Object of multibyte is stored in consecutive bytes and gives the first byte address.
* Run-time storage can be subdivide to hold the different components of an executing program:

1. Generated executable code
2. Static data objects
3. Dynamic data-object- heap
4. Automatic data objects- stack
5. **Prove that substantial improvement in the running time of code merely by performing local optimization within each basic block. By assuming a basic block and do the possible optimizations and then produce the improved code.**

**Optimization of Basic Blocks**

*1 The DAG Representation of Basic Blocks*

*2 Finding Local Common Subexpressions*

*3 Dead Code Elimination*

*4 The Use of Algebraic Identities*

*5 Representation of Array References*

*6 Pointer Assignments and Procedure Calls*

*7 Reassembling Basic Blocks From DAG's*

We can often obtain a substantial improvement in the running time of code merely by performing *local* optimization within each basic block by itself. More thorough *global* optimization, which looks at how information flows among the basic blocks of a program. It is a complex subject, with many different techniques to consider.

**1. The DAG Representation of Basic Blocks**

Many important techniques for local optimization begin by transforming a basic block into a DAG (directed acyclic graph).

There is a node in the DAG for each of the initial values of the variables appearing in the basic block.

We can eliminate *local common subexpressions,* that is, instructions that compute a value that has already been computed.

We can apply algebraic laws to reorder operands of three-address instruc-tions, and sometimes thereby simplify the computation.

**2. Finding Local Common Subexpressions**

Common subexpressions can be detected by noticing, as a new node *M* is about to be added, whether there is an existing node *N* with the same children, in the same order, and with the same operator.

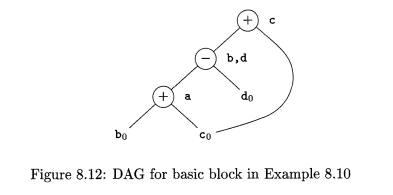
Example 8.10  :  A DAG for the block

**a**= b +**c**

b = **a** - d

**c**= b +**c**

d = **a** – d



**3. Dead Code Elimination**

he operation on DAG's that corresponds to dead-code elimination can be im-plemented as follows. We delete from a DAG any root (node with no ancestors) that has no live variables attached. Repeated application of this transformation will remove all nodes from the DAG that correspond to dead code.

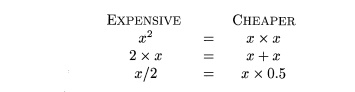
**4. The Use of Algebraic Identities**

Algebraic identities represent another important class of optimizations on basic blocks. For example, we may apply arithmetic identities, such as



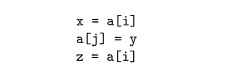
to eliminate computations from a basic block.

Another class of algebraic optimizations includes local *reduction in strength,* that is, replacing a more expensive operator by a cheaper one as in:



**5. Representation of Array References**

At first glance, it might appear that the array-indexing instructions can be treated like any other operator. Consider for instance the sequence of three-address statements:



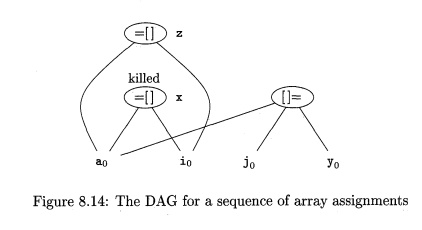
**Example 8 . 1 3 :**The DAG for the basic block

x = a [ i ]

a [ j ] =  y

z = a [ i ]

is shown in Fig- 8.14. The node N for x is created first, but when the node labeled [ ] = is created, N is killed.  Thus, when the node for z is created, it cannot be identified with N, and a new node with the same operands a0    and i0    must be created instead.



**6. Pointer Assignments and Procedure Calls**

When we assign indirectly through a pointer, as in the assignments

x = \*p

\*q = y

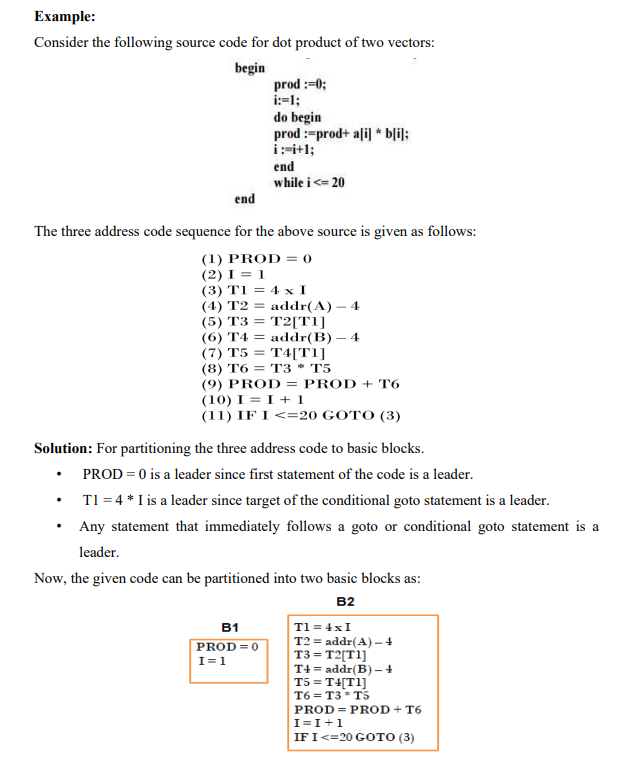
we do not know what p or q point to. In effect, x = \*p is a use of every variable whatsoever, and \*q = y is a possible assignment to every variable. As a consequence, the operator =\* must take all nodes that are currently associated with identifiers as arguments, which is relevant for dead-code elimination. More importantly, the \*= operator kills all other nodes so far constructed in the DAG.

**7. Reassembling Basic Blocks From DAG's**

After we perform whatever optimizations are possible while constructing the DAG or by manipulating the DAG once constructed, we may reconstitute the three-address code for the basic block from which we built the DAG.

If the node has more than one live variable attached, then we have to in-troduce copy statements to give the correct value to each of those variables. Sometimes, global optimization can eliminate those copies, if we can arrange to use one of two variables in place of the other.

**Additional Questions:**



**Explain SDD for Boolean expression with and without back patching.**

Backpatching is basically a process of fulfilling unspecified information. This information is of labels. It basically uses the appropriate semantic actions during the process of code generation. It may indicate the address of the Label in goto statements while producing TACs for the given expressions. Here basically two passes are used because assigning the positions of these label statements in one pass is quite challenging. It can leave these addresses unidentified in the first pass and then populate them in the second round. Backpatching is the process of filling up gaps in incomplete transformations and information.

Backpatching is mainly used for two purposes:

* 1. Boolean expression:
  2. Flow of Control statements

**Boolean expression:**

Boolean expressions are statements whose results can be either true or false. A boolean expression which is named for mathematician George Boole is an expression that evaluates to either true or false. Let’s look at some common language examples:

My favorite color is blue. → true

I am afraid of mathematics. → false

2 is greater than 5. → false

**Backpatching for Boolean Expressions:**

Using a translation technique, it can create code for Boolean expressions during [bottom-up parsing](https://www.geeksforgeeks.org/bottom-up-or-shift-reduce-parsers-set-2/). In grammar, a non-terminal marker M creates a semantic action that picks up the index of the next instruction to be created at the proper time.

*Consider the Boolean expression “a < b or c < d and e < f”. To generate three-address code for this, we have already incorporated semantic rules in the previous module. In backpatching the same code is generated in two passes. In the first pass, the following would be generated:*

*100: if a < b goto \_*

*101: goto \_*

*102: if c < d goto \_*

*103: goto \_*

*104: if e < f goto \_*

*105: goto \_*

*In the second pass, the same code is re-run to generate the true, false labels by incorporating*

*short circuit information.*

*100: if a < b goto TRUE*

*101: goto 102*

*102: if c < d goto 104*

*103: goto FALSE*

*104: if e < f goto TRUE*

*105: goto FALSE*

*In this module, we will write semantic rules to generate three-address code based in two passes using the backpatching functions discussed already.*

**Write the advantages and disadvantages of heap storage allocation strategies.**

Heap storage is used to allocate storage that has a lifetime that is not related to the execution of the current routine. The storage is shared among all program units and all threads in an enclave.

### Advantages & Disadvantages of Heap Memory

Below are the advantages and disadvantages of heap Memory:

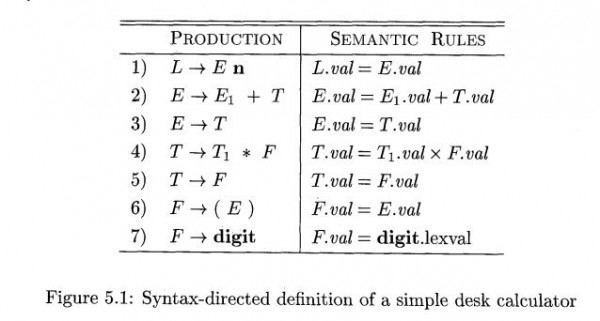
**Advantage:**

1. Heap finds the greatest and minimum number.
2. Heap method used in the Priority Queue.
3. Heap allows accessing variables globally.
4. In heap memory, the Garbage collection feature helps to free the memory used by the object.

**Disadvantages:**

1. It can provide the maximum memory an OS can provide.
2. It takes more time to compute.
3. In heap memory, Memory management is more complicated.
4. Execution time is more in heap memory.

**Analyze the grammar and syntax-directed translation for desk calculator and show the annotated parse tree for the expression (3 + 4) \* (5 + 6).**

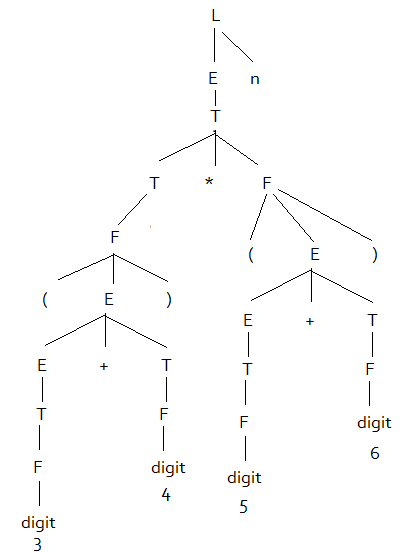


Solution:

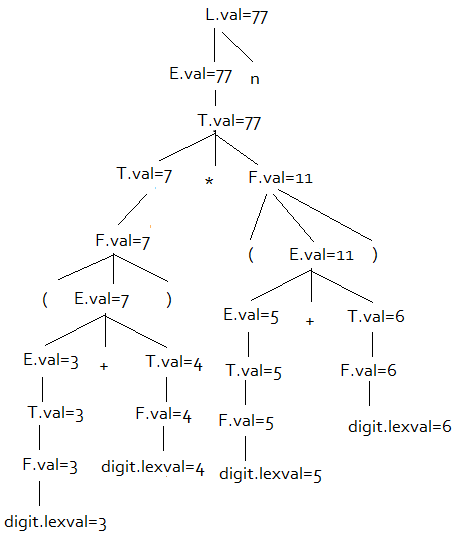
Here all the attributes are synthesized attributes. The attribute value of digit is provided by lexical analyzer since it is a terminal

1. (3+4)∗(5+6)n

Parse Tree



Annotated Parse Tree



**Write a quick sort algorithm and construct an activation tree for this algorithm.**

**QuickSort**is a[Divide and Conquer algorithm](https://www.geeksforgeeks.org/divide-and-conquer-algorithm-introduction/). It picks an element as pivot and partitions the given array around the picked pivot. There are many different versions of quickSort that pick pivot in different ways.

* Always pick first element as pivot.
* Always pick last element as pivot (implemented below)
* Pick a random element as pivot.
* Pick median as pivot.

The key process in **quickSort**is partition(). Target of partitions is, given an array and an element x of array as pivot, put x at its correct position in sorted array and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x. All this should be done in linear time.

Consider the following program of Quicksort

main() {

Int n;

readarray();

quicksort(1,n);

}

quicksort(int m, int n) {

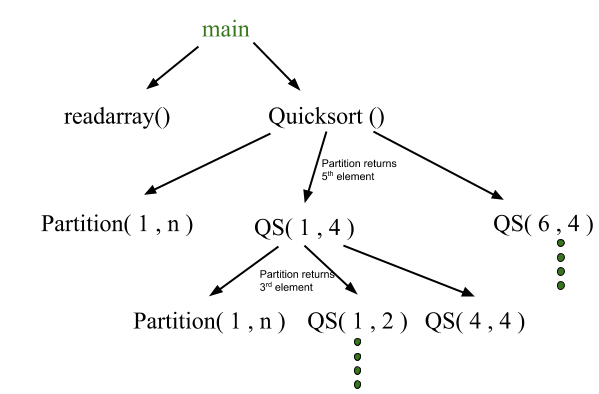
Int i= partition(m,n);

quicksort(m,i-1);

quicksort(i+1,n);

}

The activation tree for this program will be:



First main function as root then main calls readarray and quicksort. Quicksort in turn calls partition and quicksort again. The flow of control in a program corresponds to the depth first traversal of activation tree which starts at the root.

**Eliminate the local sub expression from the basic block Bx.**

**t6 = 4 \* i**

**x = a[t6]**

**t7 = 4 \* i**

**t8 = 4 \* i**

**t9 = a[t8]**

**a[t7] = t9**

**t10 = 4 \* j**

**a[t10] = x**

**goto By**

Solution

**After Local common subexpression elimination**

t6 := 4 \* i

x := a[t6]

t7 := t6

t8 := 4\*j

t9 := a[t8]

a[t7]:= t9

t10 := t8

a[t10]:= x

goto By

**After Global common subexpression elimination**

x := t3

t9 := t5

a[t2]:= t5

a[t4]:= x

goto By

**How do you relate run-time environments to compiler?**

A translation needs to relate the static source text of a program to the dynamic actions that must occur at runtime to implement the program. The program consists of names for procedures, identifiers etc., that require mapping with the actual memory location at runtime.

By runtime, we mean a program in execution. Runtime environment is a state of the target machine, which may include software libraries, environment variables, etc., to provide services to the processes running in the system.

Runtime support system is a package, mostly generated with the executable program itself and facilitates the process communication between the process and the runtime environment. It takes care of memory allocation and de-allocation while the program is being executed.

**What are the issues in the design of a code generator? Explain briefly**

Code generator converts the intermediate representation of source code into a form that can be readily executed by the machine. A code generator is expected to generate the correct code. Designing of code generator should be done in such a way so that it can be easily implemented, tested and maintained.

**The following issue arises during the code generation phase:**

1. **Input to code generator –**  
   The input to code generator is the intermediate code generated by the front end, along with information in the symbol table that determines the run-time addresses of the data-objects denoted by the names in the intermediate representation. Intermediate codes may be represented mostly in quadruples, triples, indirect triples, Postfix notation, syntax trees, DAG’s, etc. The code generation phase just proceeds on an assumption that the input are free from all of syntactic and state semantic errors, the necessary type checking has taken place and the type-conversion operators have been inserted wherever necessary.
2. **Target program –**  
   The target program is the output of the code generator. The output may be absolute machine language, relocatable machine language, assembly language.
   * Absolute machine language as output has advantages that it can be placed in a fixed memory location and can be immediately executed.
   * Relocatable machine language as an output allows subprograms and subroutines to be compiled separately. Relocatable object modules can be linked together and loaded by linking loader. But there is added expense of linking and loading.
   * Assembly language as output makes the code generation easier. We can generate symbolic instructions and use macro-facilities of assembler in generating code. And we need an additional assembly step after code generation.
3. **Memory Management –**  
   Mapping the names in the source program to the addresses of data objects is done by the front end and the code generator. A name in the three address statements refers to the symbol table entry for name. Then from the symbol table entry, a relative address can be determined for the name.
4. **Instruction selection –**  
   Selecting the best instructions will improve the efficiency of the program. It includes the instructions that should be complete and uniform. Instruction speeds and machine idioms also plays a major role when efficiency is considered. But if we do not care about the efficiency of the target program then instruction selection is straight-forward.
5. **Register allocation issues –**  
   Use of registers make the computations faster in comparison to that of memory, so efficient utilization of registers is important. The use of registers are subdivided into two subproblems:
   * During **Register allocation –** we select only those set of variables that will reside in the registers at each point in the program.
   * During a subsequent **Register assignment** phase, the specific register is picked to access the variable.
6. **Evaluation order –**  
   The code generator decides the order in which the instruction will be executed. The order of computations affects the efficiency of the target code. Among many computational orders, some will require only fewer registers to hold the intermediate results. However, picking the best order in the general case is a difficult NP-complete problem.
7. **Approaches to code generation issues:**Code generator must always generate the correct code. It is essential because of the number of special cases that a code generator might face. Some of the design goals of code generator are:
   * Correct
   * Easily maintainable
   * Testable
   * Efficient

**Explain in detail 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.

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.

1. **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.

1. **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

1. **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.

1. **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.

1. **Loop Optimizations:**

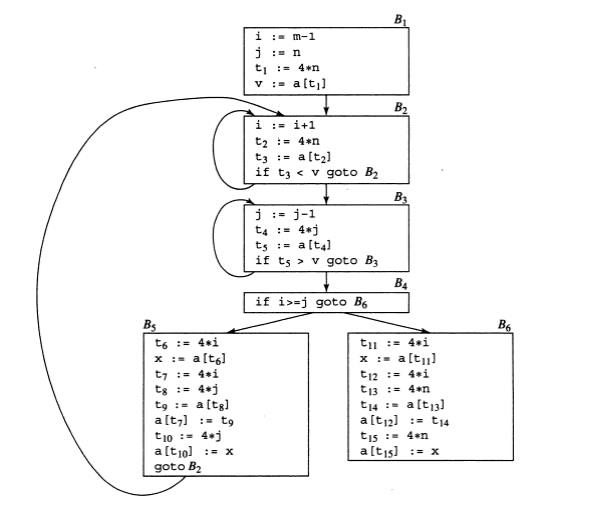
In loops, especially in the inner loops, programs tend to spend the bulk of their time. The running time of a program may be improved if the number of instructions in an inner loop is decreased, even if we increase the amount of code outside that loop.

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.



1. **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.

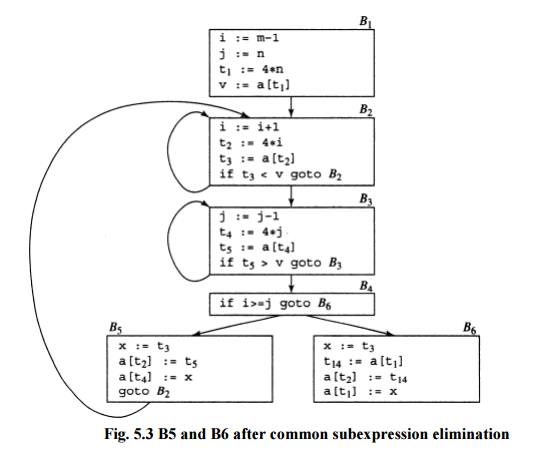
1. **Induction Variables :**

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.

1. **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, x² 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**

**Discuss global data flow analysis with a suitable example.**

# **Global data flow analysis**

* To efficiently optimize the code compiler collects all the information about the program and distribute this information to each block of the flow graph. This process is known as data-flow graph analysis.
* Certain optimization can only be achieved by examining the entire program. It can't be achieve by examining just a portion of the program.
* For this kind of optimization user defined chaining is one particular problem.
* Here using the value of the variable, we try to find out that which definition of a variable is applicable in a statement.

Based on the local information a compiler can perform some optimizations. For example, consider the following code:

1. x = a + b;
2. x = 6 \* 3

* In this code, the first assignment of x is useless. The value computer for x is never used in the program.
* At compile time the expression 6\*3 will be computed, simplifying the second assignment statement to x = 18;

Some optimization needs more global information. For example, consider the following code:

a = 1;

 b = 2;

   c = 3;

**if** (....) x = a + 5;

**else** x = b + 4;

   c = x + 1;

In this code, at line 3 the initial assignment is useless and x +1 expression can be simplified as 7.

But it is less obvious that how a compiler can discover these facts by looking only at one or two consecutive statements. A more global analysis is required so that the compiler knows the following things at each point in the program:

* Which variables are guaranteed to have constant values
* Which variables will be used before being redefined

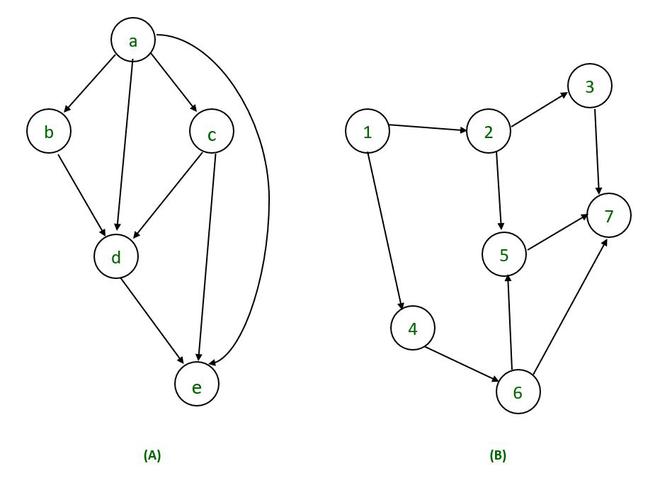
Data flow analysis is used to discover this kind of property. The data flow analysis can be performed on the program's control flow graph (CFG).

The control flow graph of a program is used to determine those parts of a program to which a particular value assigned to a variable might propagate.

**What are the conditions that need to be satisfied by the type expressions if they are represented by graphs?**Directed Acyclic Graph :  
The Directed Acyclic Graph (DAG) is used to represent the structure of [basic blocks](https://www.geeksforgeeks.org/basic-blocks-in-compiler-design/), to visualize the flow of values between basic blocks, and to provide optimization techniques in the basic block. To apply an optimization technique to a basic block, a DAG is a three-address code that is generated as the result of an intermediate code generation.

* Directed acyclic graphs are a type of data structure and they are used to apply transformations to basic blocks.
* The Directed Acyclic Graph (DAG) facilitates the transformation of basic blocks.
* DAG is an efficient method for identifying common sub-expressions.
* It demonstrates how the statement’s computed value is used in subsequent statements.

**Examples of directed acyclic graph :**



**Directed Acyclic Graph Characteristics :**  
A Directed Acyclic Graph for [Basic Block](https://www.geeksforgeeks.org/basic-blocks-in-compiler-design/) is a directed acyclic graph with the following labels on nodes.

* The graph’s leaves each have a unique identifier, which can be variable names or constants.
* The interior nodes of the graph are labelled with an operator symbol.
* In addition, nodes are given a string of identifiers to use as labels for storing the computed value.
* Directed Acyclic Graphs have their own definitions for transitive closure and transitive reduction.
* Directed Acyclic Graphs have [topological orderings](https://www.geeksforgeeks.org/topological-sorting/) defined.

**Algorithm for construction of Directed Acyclic Graph :**  
There are three possible scenarios for building a DAG on three address codes:

**Case 1 –**  x = y op z  
**Case 2 –** x = op y  
**Case 3  –**  x = y

Directed Acyclic Graph for the above cases can be built as follows :

**Step 1 –**

* If the y operand is not defined, then create a node (y).
* If the z operand is not defined, create a node for case(1) as node(z).

**Step 2 –**

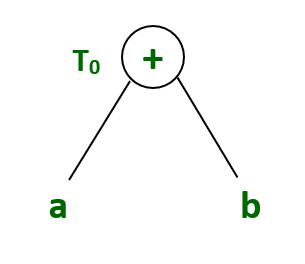
* Create node(OP) for case(1), with node(z) as its right child and node(OP) as its left child (y).
* For the case (2), see if there is a node operator (OP) with one child node (y).
* Node n will be node(y)  in case (3).

**Step 3 –**  
Remove x from the list of node identifiers. Step 2: Add x to the list of attached identifiers for node n.

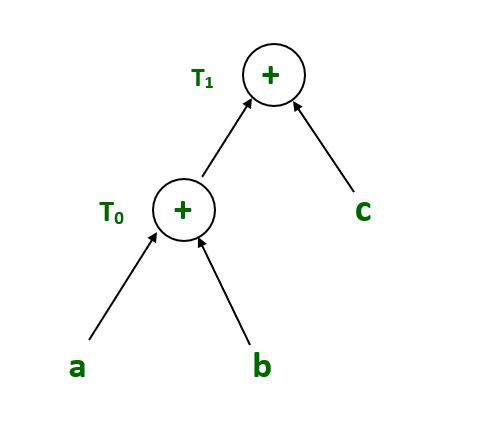
**Example :**

*T0 = a + b         —Expression 1  
T1 = T0 + c       —-Expression 2  
d = T0 + T1—–Expression 3*

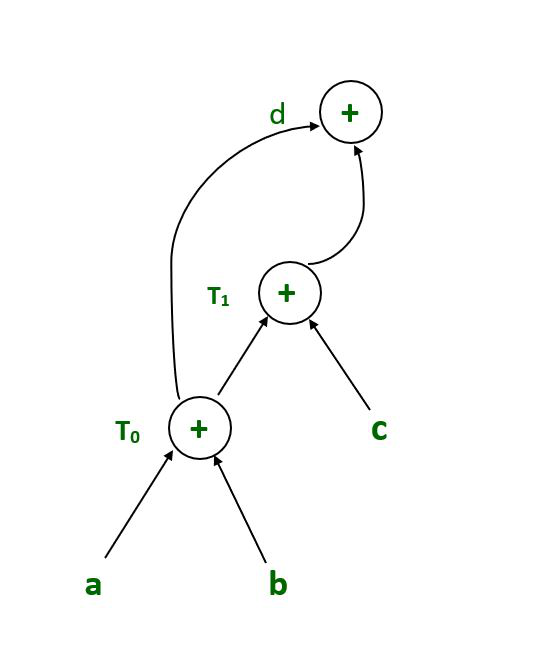
**Expression 1 :                   T0 = a + b**



**Expression 2:                    T1 = T0 + c**



**Expression 3 :                          d = T0 + T1**



**What are the structure preserving transformations? Explain with suitable examples.**

# **Optimization of Basic Blocks:**

Optimization process can be applied on a basic block. While optimization, we don't need to change the set of expressions computed by the block.

There are two type of basic block optimization. These are as follows:

1. Structure-Preserving Transformations
2. Algebraic Transformations

## 1. Structure preserving transformations:

The primary Structure-Preserving Transformation on basic blocks is as follows:

* Common sub-expression elimination
* Dead code elimination
* Renaming of temporary variables
* Interchange of two independent adjacent statements

### **(a) Common sub-expression elimination:**

In the common sub-expression, you don't need to be computed it over and over again. Instead of this you can compute it once and kept in store from where it's referenced when encountered again.

a : = b + c

b : = a - d

c : = b + c

d : = a - d

In the above expression, the second and forth expression computed the same expression. So the block can be transformed as follows:

a : = b + c

b : = a - d

c : = b + c

d : = b

### **(b) Dead-code elimination**

* It is possible that a program contains a large amount of dead code.
* This can be caused when once declared and defined once and forget to remove them in this case they serve no purpose.
* Suppose the statement x:= y + z appears in a block and x is dead symbol that means it will never subsequently used. Then without changing the value of the basic block you can safely remove this statement.

AD

### **(c) Renaming temporary variables**

A statement t:= b + c can be changed to u:= b + c where t is a temporary variable and u is a new temporary variable. All the instance of t can be replaced with the u without changing the basic block value.

### **(d) Interchange of statement**

Suppose a block has the following two adjacent statements:

t1 : = b + c

t2 : = x + y

These two statements can be interchanged without affecting the value of block when value of t1 does not affect the value of t2.

## 2. Algebraic transformations:

* In the algebraic transformation, we can change the set of expression into an algebraically equivalent set. Thus the expression x:= x + 0 or x:= x \*1 can be eliminated from a basic block without changing the set of expression.
* Constant folding is a class of related optimization. Here at compile time, we evaluate constant expressions and replace the constant expression by their values. Thus the expression 5\*2.7 would be replaced by13.5.
* Sometimes the unexpected common sub expression is generated by the relational operators like <=, >=, <, >, +, = etc.
* Sometimes associative expression is applied to expose common sub expression without changing the basic block value. if the source code has the assignments

a:= b + c

             e:= c +d +b

The following intermediate code may be generated:

a:= b + c

t:= c +d

e:= t + b

**Explain how “Redundant sub-expression Elimination” can be done at global level in a given program.**

Let‘s consider a global common sub-expression elimination optimization as our example.

Careful 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]** = ∩ **exit[x]: x ∈ pred[B]** (i.e. B has available the intersection of the exit of its predecessors)

**killed[B]** = set of the expressions killed in B

**defined[B]** = set of expressions defined in B

**exit[B] = avail[B] - killed[B] + defined[B]**

**avail[B] = ∩ (avail[x] - killed[x] + defined[x]) : x ∈ 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

redecessors or successors).

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

a) Identify each statement s 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 prior to s.

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.

**Lets try an example to make things clearer:**

main:

BeginFunc 28;

b = a + 2 ;

c = 4 \* b ;

tmp1 = b < c;

ifNZ tmp1 goto L1 ;

b = 1 ;

L1:

d = a + 2 ;

EndFunc ;

First, divide the code above into basic blocks. Now calculate the available expressions

for each block. Then find an expression available in a block and perform step 2c above.

What common subexpression can you share between the two blocks? What if the above

code were:

main:

BeginFunc 28;

b = a + 2 ;

c = 4 \* b ;

tmp1 = b < c ;

IfNZ tmp1 Goto L1 ;

b = 1 ;

z = a + 2 ; <========= an additional line here

L1:

d = a + 2 ;

EndFunc ;