UNIT V

SYLLABUS:

Run-time Environment: StorageOrganization,Stack Allocationstrategies: Static,Stack,Heapallocation, Activation record.

Code optimization: Introduction, Principal sources of optimization, Flowgraphs, Techniques in global and local optimization.

Code Generation: Issues in code generation, DAG, Simple code generator.

-

RUN-TIME ENVIRONMENT

The final phase in the compiler model is the code generator. It takes as input an intermediaterepresentation of the source program and produces as output an equivalent target program. Thecode generation techniques presented below can be used whether or not an optimizing phaseoccursbeforecodegeneration.

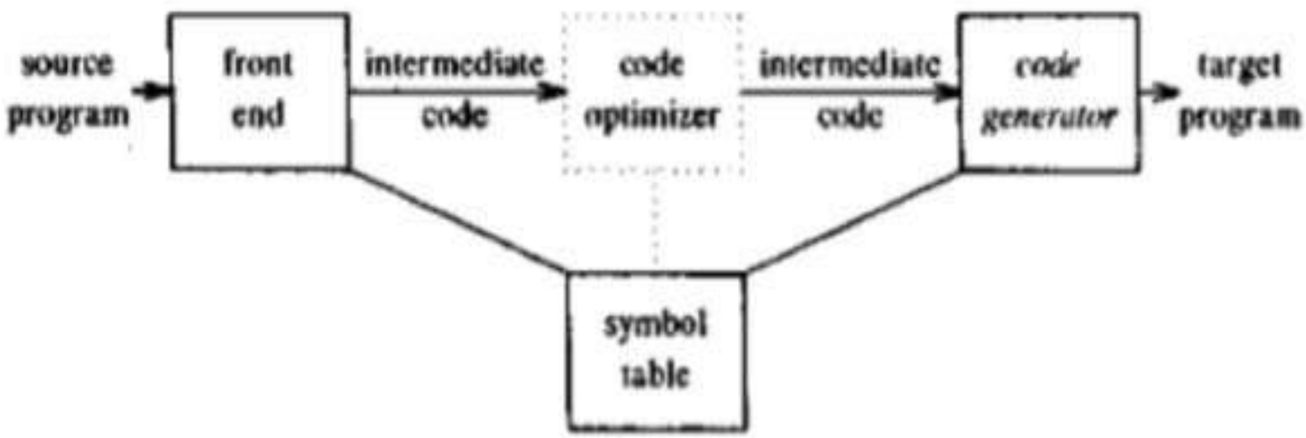


Figure 5.1 Position of code generator

STORAGEORGANISATION

- Theexecuting target program runs in its own logical address space in which each programvaluehas alocation.
- Themanagement and organization of this logical address space is shared between the complier,operatingsystemandtargetmachine.Theoperatingsystemmapsthelogicaladdressintophysicaladdresses,whichare usuallyspreadthroughoutmemory.

Typicalsubdivision ofrun-timememory:

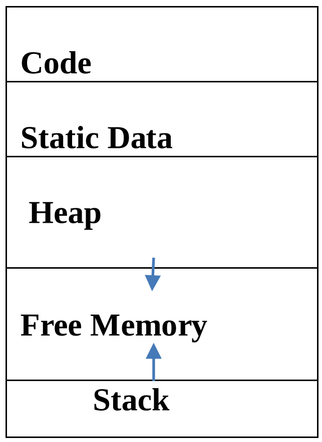


Figure 5.2: Typical subdivision of run-time memory:

- Run-time storage comes in blocks, where a byte is the smallest unitof addressable memory.Four bytes form a machine word.Multi-byte objects are stored in consecutive bytes and given the address of first byte.

  This run-time storage might be subdivided to hold:

  1. The generated target code,
  2. Data objects,and
  3. A counterpart of the control stack to keep track of procedure activations.

- The storage layout for data objects is strongly influenced by the addressing constraints of the target machine.
- A character array of length 10 needs only enough bytes to hold 10 characters, a compiler may allocate 12 bytes to get alignment,leaving 2bytes unused.
- This unused space due to alignment considerations is referred to as padding.
- The size of some program objects maybe known at runtime and maybe placed in an area called static.
- The dynamic areas used to maximize the utilization of space at runtime are stack and heap.

ACTIVATION RECORDS:

- Procedure calls and returns are usually managed by a runtime stack called the *control stack.*
- Each live activation has an activation record on the control stack, with the root of the activation tree at the bottom, the latter activation has its record at the top of the stack.
- The contents of the activation record vary with the language being implemented. The diagram below shows the contents of activation record.

Actual Parameters
Return Value
Control Link
Access Link
Machine Status
Local Data
Temporaries

Figure 5.3: General activation record.

- Temporary values such as those arising from the evaluation of expressions.
- Localdatabelongingto theprocedurewhoseactivationrecordthisis.
- Asavedmachinestatus,withinformationaboutthestateofthemachinejustbeforethecalltoprocedures.
- Anaccesslinkmaybeneededtolocatedataneededbythecalledprocedurebutfoundelsewhere.
- Acontrollinkpointing to the activationrecordofthe caller.
- Space for the return value of the called functions, if any. Again, notall called procedures return avalue,andifonedoes,wemayprefertoplacethatvalueinaregisterforefficiency.
- The actual parameters used by the calling procedure. These are not placed in activation record butratherinregisters,whenpossible,forgreaterefficiency.

STORAGE ALLOCATION STRATEGIES

The different storage allocation strategies are :

1. **Static allocation – lays out storage for all data objects at compile time**
2. **Stack allocation – manages the run-time storage as a stack.**
3. **Heap allocation – allocates and deallocates storage as needed at run time from a data area known as heap**

Staticallocation

- Instaticallocation,namesareboundtostorageastheprogramiscompiled,sothereisnoneedfora run-timesupportpackage.
- Sincethebindingsdonotchangeatrun-time,every timeaprocedureisactivated,itsnamesare boundtothesamestorage locations.
- Thereforevalues of local names are *retained* across activations of a procedure. That is, whencontrol returns to a procedure the values of the locals are the same as they were whencontrol leftthe lasttime.
- Fromthe type of a name, the compiler decides the amount of storage for the name and decideswhere the activation records go. At compile time, we can fill in the addresses at whichthe targetcode canfindthedataitoperateson.

Somelimitationsofusingstaticallocation:

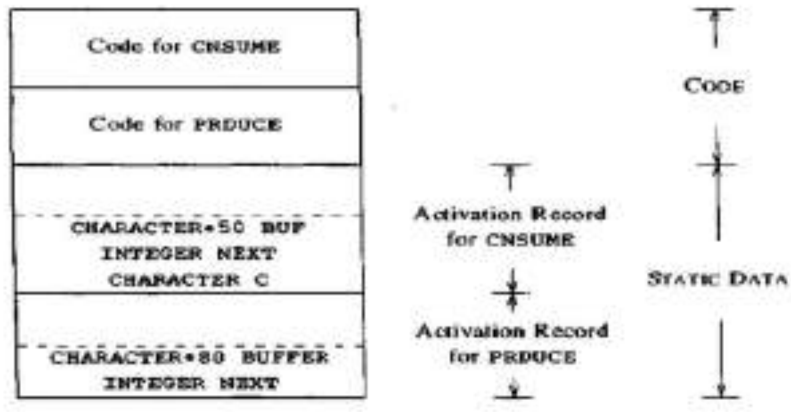
1. Thesizeofadataobjectandconstraintsonitspositioninmemorymustbeknownatcompiletime.

2. Recursive procedures are restricted, because all activations of a procedure use the samebindingsforlocal names.

3. Datastructurescannot becreateddynamically,sincethereisnomechanismforstorageallocationatruntime.

FORTRANusesstaticstorageallocation

- Consider the program
- Program CNSUME
- BUF : Array(1..80) of char
- NEXT : int
- C : char
- ....
- End
- Function PRDUCE
- BUFFER : Array(1..80) of char
- NEXT : int
- ....
- end



Stackallocation

- Allcompilersforlanguagesthatuseprocedures,functionsormethodsasunitsofuser-definedactionsmanage atleastpartoftheirrun-timememoryasastack.
- Eachtimeaprocedureis called,spaceforits localvariablesis pushedontoastack,andwhentheprocedureterminates,thatspaceis poppedoff thestack.

Callingsequences:

- Procedurescalledareimplementedinwhatiscalledascallingsequence,whichconsistsofcodethat allocatesanactivationrecordonthe stack andentersinformation intoitsfields.
- Areturnsequenceissimilartocodetorestorethestateofmachinesothecallingprocedurecancontin ueits executionafterthecall.
- Thecodein callingsequenceisoften dividedbetween thecallingprocedure(caller)andtheprocedureitcalls (callee).
- Whendesigningcallingsequencesandthelayoutofactivationrecords,thefollowingprinciplesare helpful:
  - ➢ Valuescommunicatedbetweencaller andcalleearegenerallyplacedatthebeginningofthe callee’s activation record, so they are as close as possible to the caller’s activationrecord.
- ➢ Fixedlengthitemsaregenerallyplacedinthemiddle.Suchitemstypicallyincludethecontrollink,the accesslink,andthemachinestatusfields.
- ➢ Itemswhosesizemay notbeknownearlyenoughareplaced at theend oftheactivationrecord.The most common example is dynamically sized array, where the value of one of the callee’sparametersdetermines thelengthofthearray.
- ➢ We must locate the top-of-stack pointer judiciously. A common approach is to have it point to theend offixed-length fieldsintheactivationrecord.Fixed-lengthdatacanthenbeaccessed by fixedoffsets,knowntotheintermediate-code generator,relative tothetop-of-stackpoint

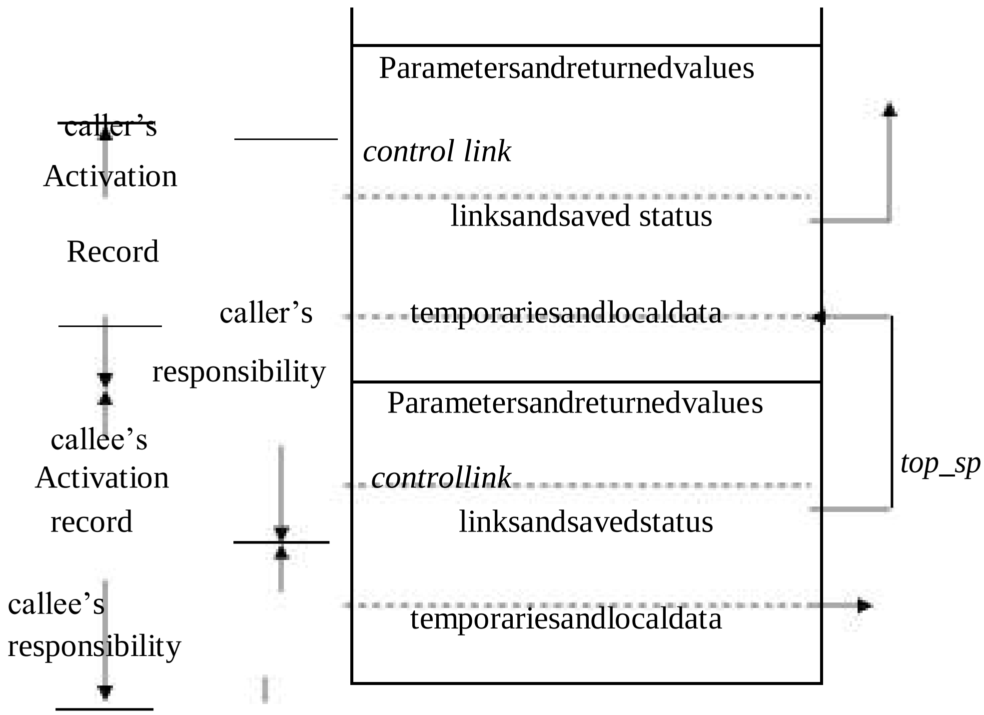


Figure 5.5:Division oftasksbetween callerand callee

- Thecallingsequenceanditsdivisionbetweencaller andcalleeareasfollows.
  - Thecallerevaluatestheactualparameters.
  - The caller stores a return address and the old value of *top_sp*into the callee's activationrecord.The callerthen incrementsthe*top_sp*tothe respectivepositions.
  - Thecalleesavestheregistervaluesandotherstatusinformation.
  - Thecalleeinitializes itslocaldataandbeginsexecution.

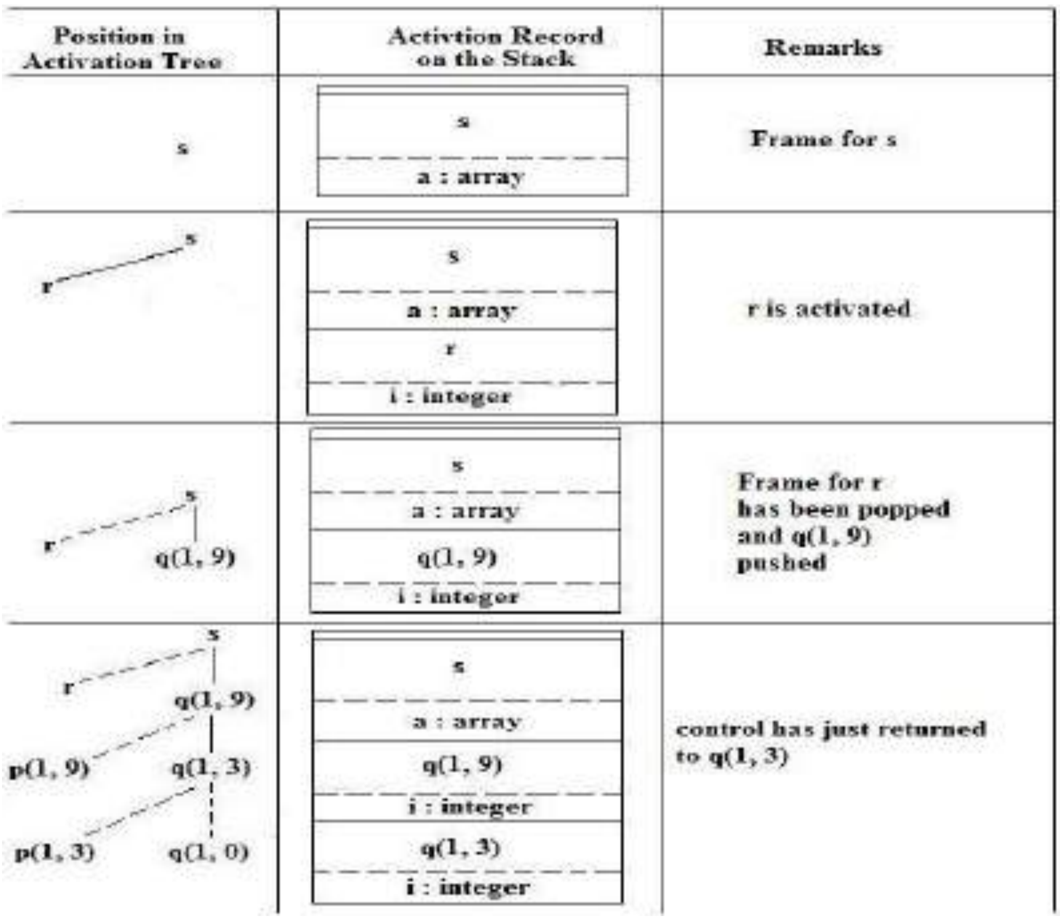
- Asuitable,correspondingreturnsequence is:
  - Thecalleeplacesthereturnvaluenexttotheparameters.
  - Usingtheinformationinthemachine-statusfield,thecalleerestores*top_sp*andotherregisters,and then branchestothereturnaddressthat thecallerplacedinthestatusfield.
  - Although*top_sp*hasbeendecremented,thecallerknowswherethereturnvalueis,relativetothecurrentv alueof*top_sp*;thecaller thereforemayusethatvalue.

Variablelength dataonstack:

- The run-time memory management system must deal frequently with the allocation of space forobjects, the sizes of which are not known at the compile time, but which are local to a procedureandthusmaybeallocatedonthestack.
- Thereasontopreferplacingobjectsonthestackis thatweavoidtheexpenseof garbagecollecting

theirspace.

- The same scheme works for objects of any type if they are local to the procedure called and have asize thatdepends onthe parameters ofthecall.



Heapallocation

Stackallocationstrategycannotbeusedifeither ofthefollowingispossible:

1. Thevaluesoflocalnames mustberetainedwhenanactivationends.
2. Acalledactivationoutlivesthecaller.

- Heapallocation parcels outpieces of contiguous storage, as neededforactivation records orotherobjects.
- Piecesmaybedeallocatedinanyorder,sooverthetimetheheapwillconsistofalternateareasthatarefree andinuse

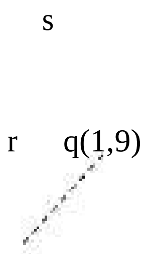
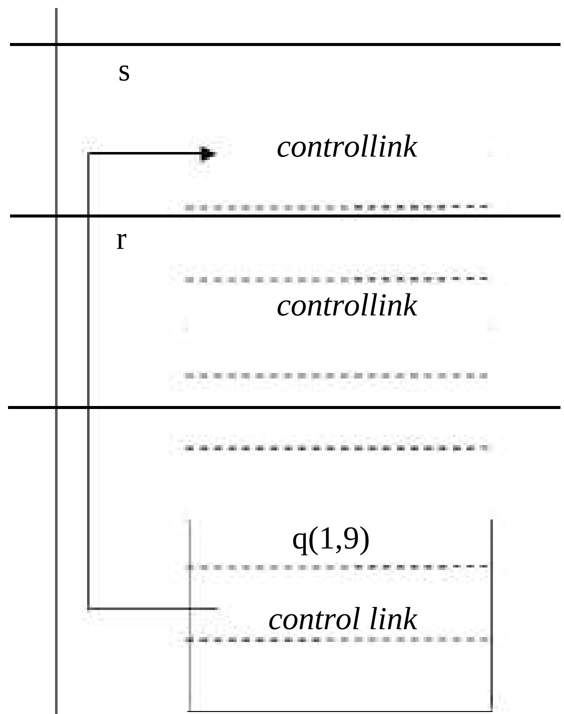
Positioninthe activationtree	Activationrecordsintheheap	Remarks
		Retainedactivation record forr

Figure 5.6: Heap allocation

- Therecordforanactivationofprocedurerisretainedwhentheactivationends.
- Therefore,therecordforthenewactivationq(1,9)cannotfollowthatfor sphysically.
- Iftheretainedactivationrecordforrisdeallocated,therewillbefreespaceintheheapbetweentheactivationrecords fors andq.
- Forlargeblocksofstorageusetheheapmanager. Thisapproachresultsinfastallocation anddeallocationofsmallamountsofstorage,sincetakingandreturningablockfromlinkedlistare efficientoperations.

CODE GENERATION

ISSUESINTHEDESIGNOFACODEGENERATOR

Thefollowingissuesariseduringthecodegenerationphase:

1. Input tocode generator
2. Targetprogram
3. Memorymanagement

4. Instructionselection
5. Registerallocation
6. Evaluationorder

1. Input tocodegenerator:

- The inputto the code generation consists of the intermediate representation of the source programproduced by front end , together with information in the symbol table to determinerun-timeaddressesofthedataobjectsdenotedbythenamesintheintermediate representation.

- Intermediaterepresentationcanbe:
  a. Linear representationsuchaspostfix notation
  b. Threeaddressrepresentationsuchasquadruples
  c. Virtualmachinerepresentationsuchasstackmachinecode
  d. Graphicalrepresentationssuchassyntaxtreesanddags.

- Prior to code generation, the front end must be scanned, parsed and translated into intermediaterepresentation along with necessary type checking. Therefore, input to code generation is assumedtobeerror-free.

2. Targetprogram:

- Theoutputofthecodegeneratoristhetargetprogram.Theoutputmaybe:
  a. Absolutemachinelanguage
     - Itcanbeplacedinafixedmemorylocationandcanbeexecutedimmediately.

  b. Relocatablemachinelanguage
     - Itallowssubprogramstobecompiledseparately.

  c. Assemblylanguage
     - Codegenerationismadeeasier.

3. Memorymanagement:

- Namesinthesourceprogramaremappedtoaddressesofdataobjectsinrun-timememorybythe frontendandcodegenerator.

- Itmakesuseofsymboltable,thatis,anameinathree-addressstatementreferstoasymbol-table entryforthename.

- Labels in three-address statements have to be converted to addresses of instructions. Forexample,

  j:**goto***i*generatesjumpinstructionasfollows:

  - if$i<j$, a backward jump instruction with target address equal to location of code forquadruple*i*is generated.
  - if$i>j$, the jump is forward. We must store on a list for quadruple*i*thelocation of thefirst machine instruction generated for quadruple*j*. When *i*isprocessed, the machinelocationsforallinstructionsthatforwardjumpsto*i*are filled.

4. **Instructionselection:**

- Theinstructionsoftargetmachineshouldbecompleteanduniform.isconsidered.

- Thequalityofthe generated codeisdeterminedbyitsspeedandsize.

- Theformerstatementcanbetranslatedintothelatterstatementasshownbelow:

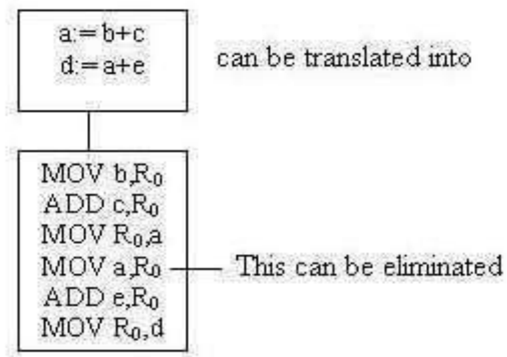


Figure 5.3: Code Translation

5. **Registerallocation**

- Instructionsinvolvingregisteroperandsareshorterandfasterthanthoseinvolvingoperandsinmemory.

- Theuseofregistersissubdividedintotwosubproblems:

Registerallocation–thesetofvariables thatwillresideinregistersat apointintheprogramisselected.

  - ***Registerassignment***–thespecificregisterthatavariablewillresidein ispicked.

- Certainmachinerequireseven-odd*registerpairs*forsomeoperandsandresults.Forexample ,considerthedivisioninstructionoftheform:

  Dx,y

  where,x–dividendevenregister ineven/oddregisterpairy–
  divisor
  even register holds the remainder
  oddregisterholdsthe quotient

6. **Evaluationorder**

- Theorderinwhichthecomputationsareperformedcanaffecttheefficiencyofthetargetcode.Somecom putationordersrequirefewerregisterstoholdintermediateresultsthanothers.

<u>ASIMPLECODEGENERATOR</u>

- Acodegeneratorgeneratestargetcodeforasequenceofthree-addressstatementsandeffectivelyusesregisterstostoreoperands ofthestatements.

For example:considerthethree-addressstatement**a:=b+c**

Itcanhavethefollowingsequenceofcodes:

ADDR_j,R_i Cost=1 //if$R_i$containsband$R_j$containsc(or)
ADDc,R_i Cost=2 //ifcisinamemorylocation

(or)

MOVc,R_j Cost=3 //movecfrommemorytoRjandadd
ADDR_j, R_i

RegisterandAddressDescriptors:

- Aregisterdescriptorisusedtokeeptrackofwhatiscurrentlyineachregisters.Theregisterdescriptorsshowthatinitiallyalltheregistersareempty.
- Anaddressdescriptorstoresthelocationwherethecurrentvalueofthenamecanbefoundatruntime.

Acode-generationalgorithm:

Thealgorithmtakesas inputasequenceofthree-addressstatementsconstitutingabasicblock. For eachthree-addressstatementoftheformx:=yopz**,**performthefollowingactions:

1. Invokeafunction*getreg*todeterminethelocationLwheretheresultofthecomputationyopzshouldbe stored.
2. Consulttheaddressdescriptorforytodeterminey’, thecurrentlocationofy. Prefertheregisterfory’ if the value of y is currently both in memory and a register. If the value of y is not already in L,generate theinstruction **MOVy’,L**toplacea copyofyinL.
3. Generatetheinstruction**OPz’,L**wherez’isacurrentlocationofz.Preferaregistertoamemorylocationif zisinboth.Updatetheaddressdescriptor of xtoindicatethatxisinlocation L.IfxisinL,updateitsdescriptorand removexfromallotherdescriptors.
4. If the current values of y or z have no next uses, are not live on exit from the block, and are inregisters,altertheregisterdescriptorto indicatethat,afterexecutionofx:=yopz,those registerswillnolongercontainyorz.

Thealgorithmicsequenceof*getreg*functioncanbe,

1. ifxvalueisinregisterthatregisterisreturned.
2. If(1) fails,newregister isreturned.

3. If(2)fails,andtheoperationneedsaspecialregister,thatregistervalueistemporarilymovedtothe memoryandtheregisteris returned.
4. If(3)fails,finallymemorylocationisreturned.

GeneratingCodeforAssignment Statements:

- Theassignment d : = (a-b) + (a-c) + (a-c) might be translated into the following three-addresscodesequence:

  t : = a – b u :
  =a – c v : = t
  + ud:=v+u

withdliveattheend.

Codesequencefortheexampleis:

Statements	CodeGenerated	Registerdescriptor	Addressdescriptor
		Registerempty	
t:=a-b	MOVa,R0 SUBb,R0	R0containst	tinR0
u:=a-c	MOVa ,R1 SUBc,R1	R0 containst R1containsu	tinR0 uinR1
v:=t+u	ADDR1,R0	R0 containsv R1 containsu	uinR1 vinR0
d:=v+u	ADDR1,R0 MOV R0,d	R0 containsd	dinR0 dinR0andmemory

GeneratingCodeforIndexedAssignments

Thetableshowsthecodesequencesgenerated for theindexed assignmentstatements**a:=b[i]** and**a[i]:=b**

Statements	CodeGenerated
a:=b[i]	MOVb(Ri),R
a[i]:= b	MOVb,a(Ri)

GeneratingCodeforPointerAssignments

Thetableshowsthecodesequencesgenerated forthe pointerassignments**a :=*p**and ***p :=a**

Statements	CodeGenerated
a:=*p	MOV*Rp,a
*p:=a	MOVa,*Rp

GeneratingCode forConditionalStatements:

Statement	Code
if x < y goto z	**CMP x, y** **CJ<z** **/* jump to z if condition code is negative */**
x : = y +z if x <0 goto z MOV y, R0	**ADD z, R0** **MOV R0,x** **CJ<Z**

DAG:

THE DAG REPRESENTATION FOR BASIC BLOCKS

- A DAG for a basic block is a directed acyclic graph with the following labels on nodes:
  1. Leaves are labeled by unique identifiers, either variable names or constants.
  2. Interior nodes are labeled by an operator symbol.
  3. Nodes are also optionally given a sequence of identifiers for labels to store the computed values.
- DAGs are useful data structures for implementing transformations on basic blocks.
- It gives a picture of how the value computed by a statement is used in subsequent statements.
- It provides a good way of determining common sub - expressions.

Algorithm for construction of DAG

Input: Abasicblock

Output:ADAGforthebasicblockcontainingthefollowinginformation:

1. A label for each node. For leaves, the label is an identifier. For interior nodes, anoperator symbol.
2. Foreachnodealistofattachedidentifierstoholdthecomputedvalues. Case

(i) x : = y OP z

Case(ii)x:=OPy Case

(iii) x : = y

Method:

Step1: Ifyisundefinedthencreate node(y).

Ifzisundefined,createnode(z)forcase(i).

Step2: Forthecase(i),createanode(OP)whoseleftchildisnode(y)andrightchildis node(z). (

Checking for common sub expression). Let n be this node.

For case(ii), determine whether there is node(OP)with onechild node(y). Ifnotcreate such a node.

Forcase(iii),nodenwillbenode(y).

Step3: Deletexfromthelistofidentifiersfornode(x).Appendxtothelistofattached identifiers for

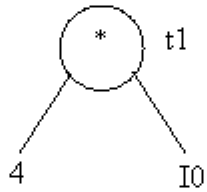
the node n found in step 2 and set node(x) to n.

Example: Consider the block of three- address statements:

1. t_1:=4*i
2. t_2:=a[t_1]
3. t_3:=4*i
4. t_4:=b[t_3]
5. t_5:=t_2*t_4
6. t_6:=prod+t_5
7. prod:=t_6
8. t_7:=i+1
9. i:=t_7
10. ifi<=20goto(1)

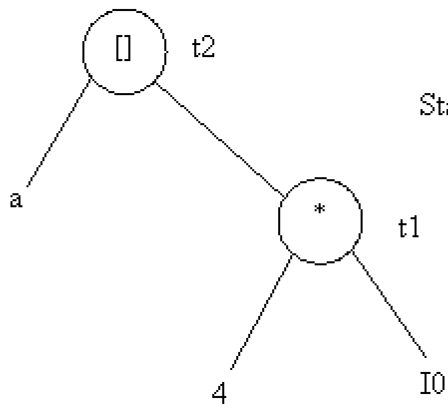
Stages in DAG Construction

(a)



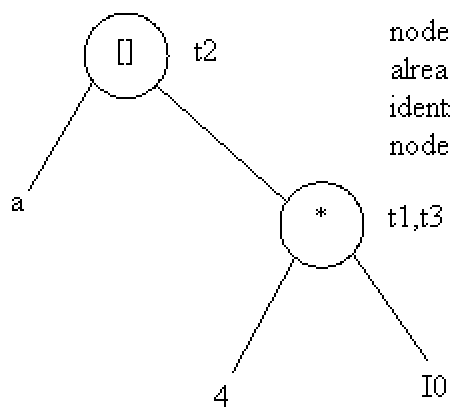
Statement (1)

(b)



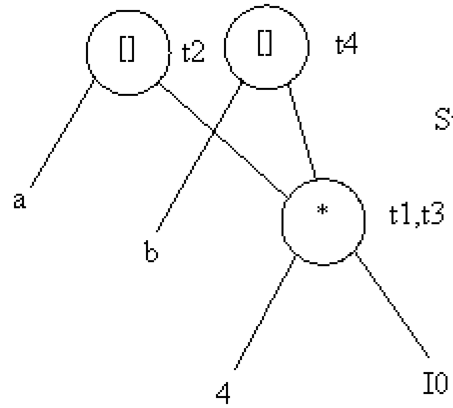
Statement (2)

(c)



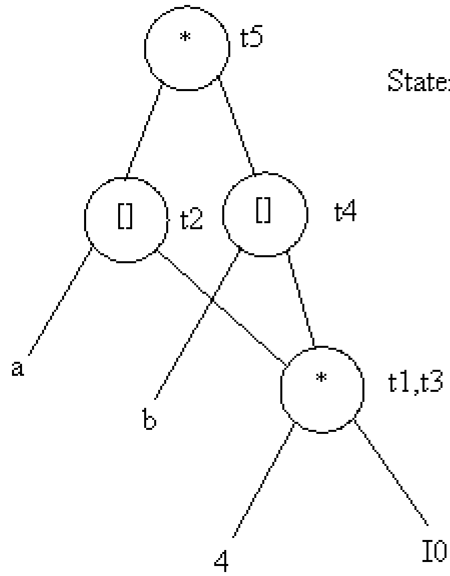
node for 4*I0 exist already, hence attach identifier t3 to the existing node for Statement (3)

(d)



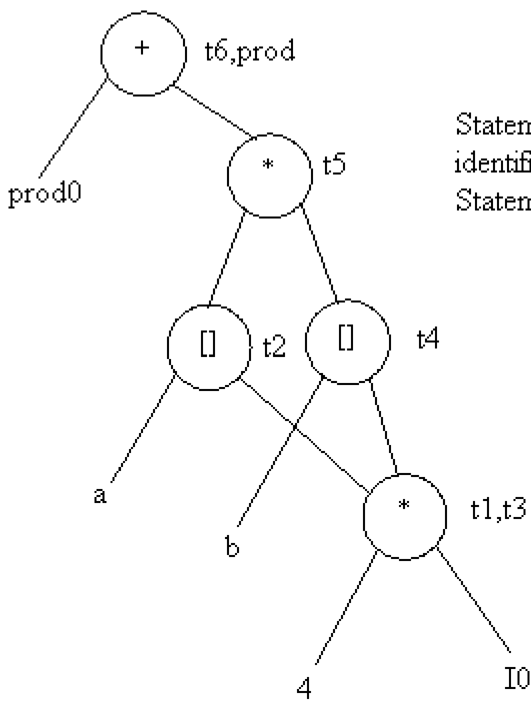
Statement (4)

(e)



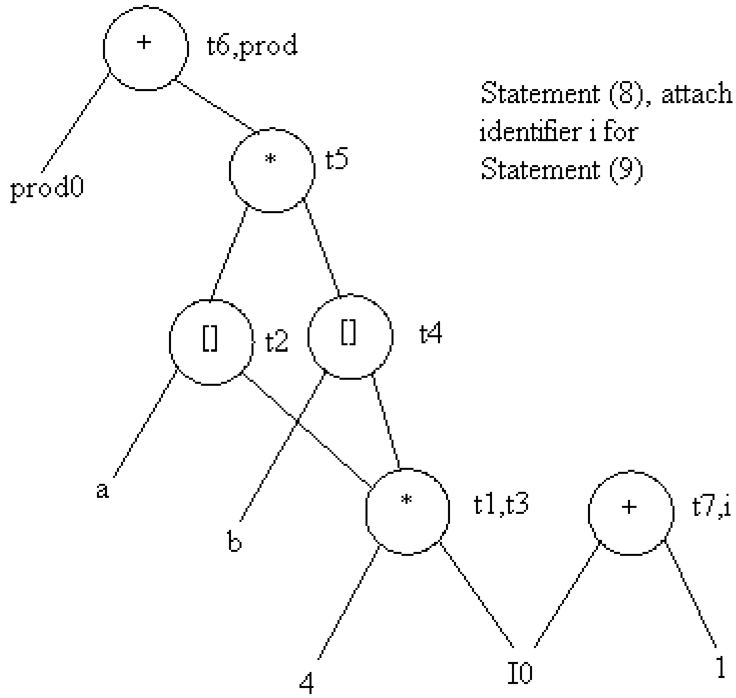
Statement (5)

(f)

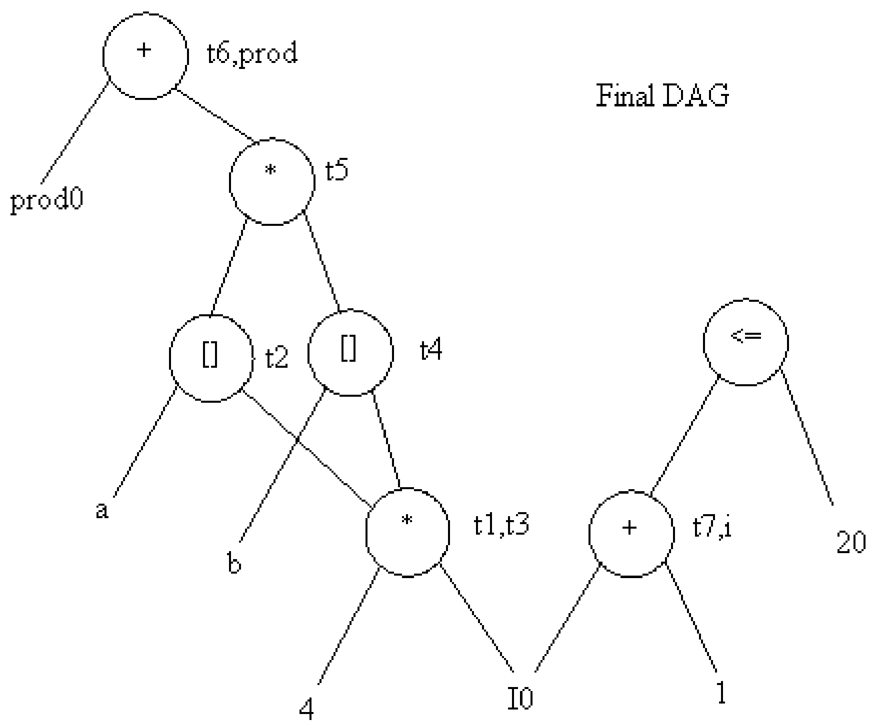


Statement (6), attach identifier prod for Statement (7)

(g)



(h)



Application of DAGs:

1. We can automatically detect common sub expressions.
2. We can determine which identifiers have their values used in the block.
3. We can determine which statements compute values that could be used outside the block.

GENERATING CODE FROM DAGs

The advantage of generating code for a basic block from its dag representation is that, from a dag we can easily see how to rearrange the order of the final computation sequence than we can starting from a linear sequence of three-address statements or quadruples.

Rearranging the order

The order in which computations are done can affect the cost of resulting object code.

For example, consider the following basic block: $t_1 : = a + b$

$t_2 : = c + d$

$t_3 : = e - t_2$

$t_4 : = t_1 - t_3$

Generated code sequence for basic block:

MOV a , R_0

ADD b , R_0

MOV c , R_1

ADD d , R_1

MOV R_0 , t_1

MOV e , R_0

SUB R_1 , R_0

MOV t_1 , R_1

SUB R_0 , R_1

MOV R_1 , t_4

Rearranged basic block:

Now t1 occurs immediately before t4.

$t_2 : = c + d$

$t_3 : = e - t_2$

$t_1 : = a + b$

$t_4 : = t_1 - t_3$

Revised code sequence:

MOV c , R_0

ADD d , R_0

MOV a , R_0

SUB R_0 , R_1

MOV a , R_0

ADD b , R_0

SUB R_1 , R_0

MOV R_0 , t_4

In this order, two instructions MOV R_0 , t_1 and MOV t_1 , R_1 have been saved.

A Heuristic ordering for Dags:

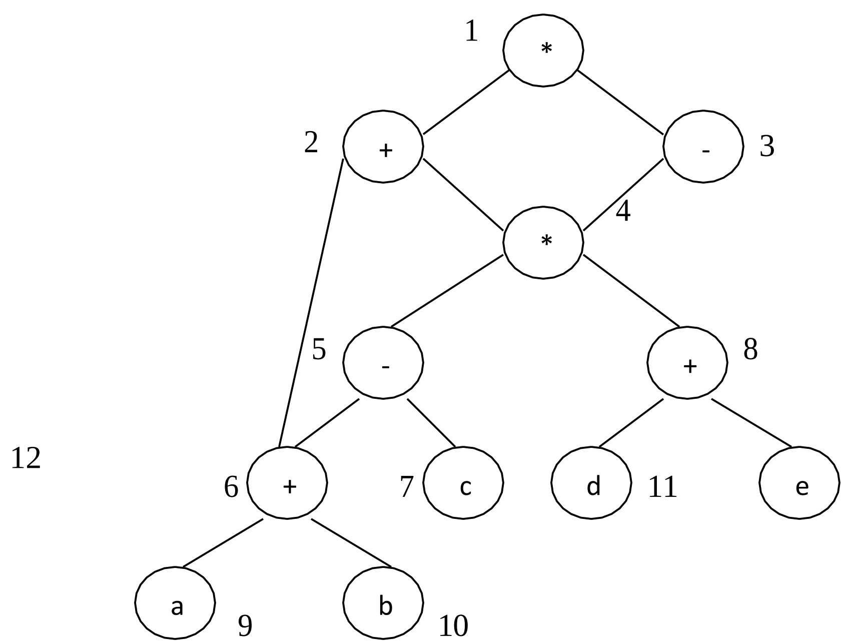
The heuristic ordering algorithm attempts to make the evaluation of a node immediately follow the evaluation of its leftmost argument.
The algorithm shown below produces the ordering in reverse.

Algorithm:

1) while unlisted interior nodes remain do begin
2) select an unlisted node n, all of whose parents have been listed;
3) list n;
4) while the leftmost child m of n has no unlisted parents and is not a leaf do begin
5) list m;
6) n : = m

endend

Example: Consider the DAG shown below:



Initially, the only node with no unlisted parents is 1 so set n=1 at line (2) and list 1 at line (3).

Now, the left argument of 1, which is 2, has its parents listed, so we list 2 and set n=2 at line (6).

Now, at line (4) we find the leftmost child of 2, which is 6, has an unlisted parent 5. Thus we select a new n at line (2), and node 3 is the only candidate. We list 3 and proceed down its left chain, listing 4, 5 and 6. This leaves only 8 among the interior nodes so we list that.
The resulting list is 1234568 and the order of evaluation is 8654321.

Code sequence:

$t_8 := d + e$
$t_6 := a + b$
$t_5 := t_6 - c$
$t_4 := t_5 * t_8$
$t_3 := t4 - e$
$t_2 := t_6 + t_4$
$t_1 := t_2 * t_3$

This will yield an optimal code for the DAG on machine whatever be the number of registers.

CODE OPTIMIZATION

INTRODUCTION: The code produced by the straight forward compiling algorithms can often be made to run faster or take less space, or both. This improvement is achieved by program transformations that are traditionally called optimizations. Compilers that apply code-improving transformations are called optimizing compilers.

Optimizations are classified into two categories. They are

- Machine independent optimizations:
- Machine dependent optimizations:

Machine independent optimizations:

- Machine independent optimizations are program transformations that improve the target code without taking into consideration any properties of the target machine.

Machine dependent optimizations:

- Machine dependent optimizations are based on register allocation and utilization of special machine-instruction sequences.

The criteria for code improvement transformations:

- ✓ Simply stated, the best program transformations are those that yield the most benefit for the least effort.

- ✓ The transformation must preserve the meaning of programs. That is, the optimization must not change the output produced by a program for a given input, or cause an error such as division by zero, that was not present in the original source program. At all times we take the “safe” approach of missing an opportunity to apply a transformation rather than risk changing what the program does.

- ✓ A transformation must, on the average, speed up programs by a measurable amount. We are also interested in reducing the size of the compiled code although the size of the code has less importance than it once had. Not every transformation succeeds in improving every program, occasionally an “optimization” may slow down a program slightly.

- ✓ The transformation must be worth the effort. It does not make sense for a compiler writer to expend the intellectual effort to implement a code improving transformation and to have the compiler expend the additional time compiling source programs if this effort is not repaid when the target programs are executed. “Peephole” transformations of this kind are simple enough and beneficial enough to be included in any compiler.

Organization for an Optimizing Compiler:

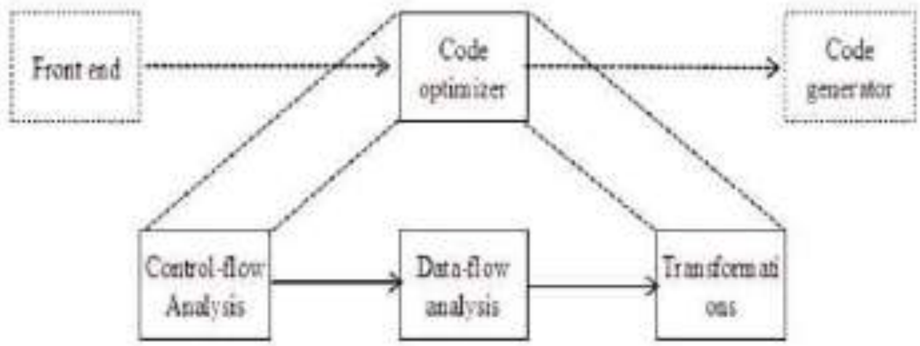


Figure 5.7: Organization for an Optimizing Compiler

- Flow analysis is a fundamental prerequisite for many important types of code improvement.
- Generally control flow analysis precedes data flow analysis.
- Control flow analysis (CFA) represents flow of control usually in form of graphs, CFA constructs such as:
  - Control flow graph
  - Call graph
- Data flow analysis (DFA) is the process of asserting and collecting information prior to program execution about the possible modification, preservation, and use of certain entities (such as values or attributes of variables) in a computer program.

PRINCIPALSOURCESOFOPTIMISATION

- Atransformationofaprogramiscalledlocalifitcanbeperformedbylookingonlyatthe 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-PreservingTransformations

- There are a number of ways in which a compiler can improve a program without changing the function it computes.
- Thetransformations:
  - ✓ Commonsubexpressionelimination,
  - ✓ Copypropagation,
  - ✓ Dead-codeelimination,and
  - ✓ Constantfolding

Arecommonexamplesofsuchfunction-preservingtransformations.Theother transformations come up primarily when global optimizations are performed.

- Frequently, a program will include several calculations of the same value, such as an 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.

- <u>CommonSubexpressionselimination:</u>

  - AnoccurrenceofanexpressionEiscalledacommonsub-expressionifEwaspreviously 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.
  - Forexample

t_1: = 4*i
t_2:=a[t1]
t_3: = 4*j
t_4: = 4*i
t_5: = n
t_6:=b[t_4]+t_5

Theabovecodecanbeoptimizedusingthecommonsub-expressioneliminationas t_1: = 4*i
t_2:=a[t_1]
t_3: = 4*j
t_5: = n
t_6:=b[t_1]+t_5

The common sub expression t_4: =4*i iseliminatedasitscomputationisalready in t_1. And value of i is not been changed from definition to use.

- <u>CopyPropagation:</u>

  - 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. Thismaynot appear to be an improvement, but as we shall see it gives us an opportunity to eliminate x.
  - Forexample:

x=Pi;
A=x*r*r;

Theoptimizationusingcopypropagationcanbedoneasfollows: A=Pi*r*r;
Herethevariablexiseliminated

- <u>Dead-Code Eliminations</u>:

  - Avariableisliveatapointinaprogramifitsvaluecanbeusedsubsequently;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,itmayappearastheresultofprevioustransformations. Anoptimizationcan be done by eliminating dead code.
Example:

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

Here,'if'statementisdeadcodebecausethisconditionwillnevergetsatisfied.

- <u>Constantfolding:</u>

  - We can eliminate both the test and printing from the object code. More generally, deducing at compile time that the value of an expression is a constant and using the constant instead is known as constant folding.

  - Oneadvantageofcopypropagationisthatitoftenturnsthecopystatementintodead code.
  - Forexample,

a=3.14157/2canbereplacedby
a=1.570therebyeliminatingadivisionoperation.

- <u>LoopOptimizations:</u>

  - We now give a brief introduction to a very important place for optimizations, namely loops,especiallytheinnerloopswhereprogramstendtospend thebulkoftheirtime.The running time of a program maybe improved if we decrease the number of instructions in an inner loop, even if we increase the amount of code outside that loop.
  - Threetechniquesareimportantforloopoptimization:
  - codemotion,whichmovescodeoutsidealoop;
  - Induction-variableelimination,whichweapplytoreplacevariablesfrominnerloop.
  - Reduction instrength,whichreplaces andexpensiveoperation bya cheaperone, suchas a multiplication by an addition.

- <u>CodeMotion:</u>

  - 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)                    /*statementdoesnotchangelimit*/
```

Code motion will result in the equivalent of

t=limit-2;
while(i<=t)   /*statementdoesnotchangelimitort*/

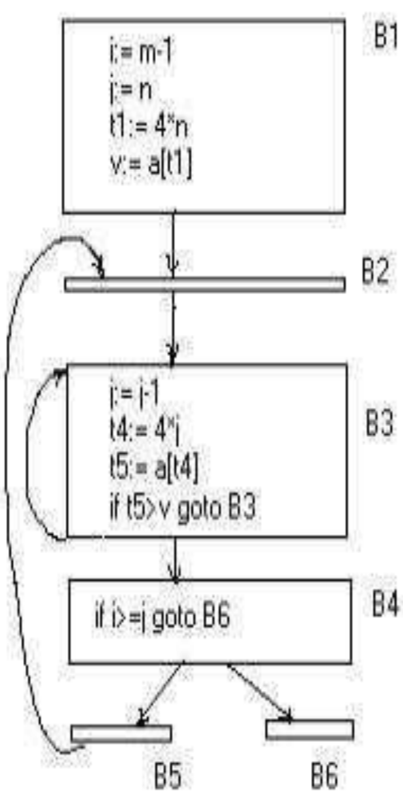
➢ InductionVariables:

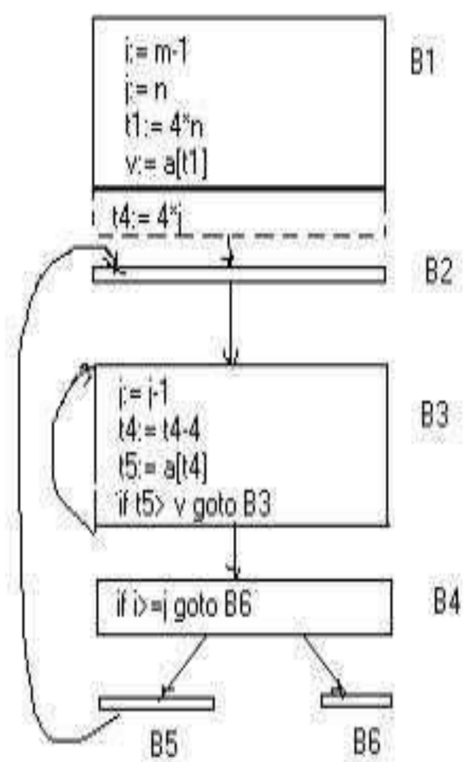
- LoopsareusuallyprocessedinsideoutForexampleconsiderthelooparoundB3.
- Note that the values of jand t_4 remain in lock-step; everytime the value of j decreases by 1, that of t_4 decreases by 4 because 4*j is assigned to t_4. Such identifiers are called induction variables.
- Whentherearetwoor moreinduction variables inaloop,itmaybe possibletogetridof all but one, by the process of induction-variable elimination. For the inner loop around B3 in Fig. we cannot get rid of either j or t_4 completely; t_4 is used in B3 and j in B4. However, we can illustrate reduction in strength and illustrate a part of the process of induction-variableelimination.EventuallyjwillbeeliminatedwhentheouterloopofB2

-B5isconsidered.

Example:

As the relationship t_4:=4*j surelyholds after such an assignment to t_4 in Fig. and t_4 is not changed elsewhere in the inner loop around B3, it follows that just after the statement j:=j-1therelationshipt_4:=4*j-4musthold.Wemaythereforereplacetheassignmentt_4:= 4*jbyt_4:=t_4-4.Theonlyproblemis that$t_4$doesnothaveavaluewhenweenterblock B3 forthefirsttime.Sincewemustmaintaintherelationshipt_4=4*jonentrytotheblockB3, we place an initializations of t_4 at the end of the block where j itself is





before   after

Initialized,shownbythedashedadditiontoblockB1insecondFig.

- Thereplacementofamultiplicationbyasubtractionwillspeeduptheobjectcodeif multiplication takes more time than addition or subtraction, as is the case on many machines.

➢ ReductionInStrength:

- Reduction in strength replaces expensive operations by equivalent cheaper ones on the target machine. Certain machine instructions are considerably cheaper than others andcan 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 cheapertoimplementas ashift.Floating-pointdivisionbyaconstantcanbeimplemented as multiplication by a constant, which may be cheaper.

OPTIMIZATIONOFBASICBLOCKS

Therearetwotypesofbasicblockoptimizations.Theyare:

- Structure-PreservingTransformations
- AlgebraicTransformations

Structure-PreservingTransformations:

TheprimaryStructure-PreservingTransformationonbasicblocksare:

- Commonsub-expressionelimination
- Deadcodeelimination
- Renamingoftemporaryvariables
- Interchangeoftwoindependentadjacentstatements.

➢ Commonsub-expressionelimination:

Common sub expressions need not be computed over and over again. Instead they can be computedonceand kept in store from where it's referencedwhenencounteredagain – of course providing the variable values in the expression still remain constant.

Example:

a:=b+c
b: =a-d
c:=b+c
d: =a-d

The2ndand4thstatementscomputethesameexpression: b+canda-d Basic block can be transformed to
a:=b+c
b: = a-d
c: = a
d:=b

- Deadcodeelimination:

It’s possible that a large amount of dead (useless) code may exist in the program. This might be especially caused when introducing variables and procedures as part of construction or error-correction of a program – once declared and defined, one forgets to remove them in case they serve no purpose. Eliminating these will definitely optimize the code.

- Renamingoftemporaryvariables:

  - A statement t:=b+c where t is a temporary name can be changed to u:=b+c where u is another temporary name, and change all uses of t to u.
  - Inthiswecantransformabasicblocktoitsequivalentblockcallednormal-formblock.

- Interchangeoftwoindependentadjacentstatements:

  - Twostatements
    t_1:=b+c

t_2:=x+y

Canbeinterchangedorreorderedinitscomputationinthebasicblockwhenvalueof$t_1$doesnotaffectthevalueoft_2.

AlgebraicTransformations:

- Algebraic identities represent another important class of optimizations on basic blocks. This includes simplifying expressions or replacing expensive operation by cheaper ones

i.e.reductioninstrength.

- Another class of related optimizations is constant folding. Here we evaluate constant expressions at compile time and replace the constant expressions by their values. Thusthe expression 2*3.14 would be replaced by 6.28.
- The relational operators <=, >=, <, >, + and = sometimes generate unexpected common sub expressions.
- Associativelawsmayalsobeappliedto exposecommonsubexpressions.Forexample,if the source code has the assignments

a:=b+c
e:=c+d+b
thefollowingintermediatecodemaybegenerated:
a :=b+c
t:=c+d
e:=t+b

Example:

x:=x+0canberemoved

x:=y**2canbereplacedbyacheaperstatementx:=y*y

- The compiler writer should examine the language carefully to determine what rearrangementsofcomputationsarepermitted,sincecomputerarithmeticdoesnotalways obey the algebraic identities of mathematics. Thus, a compiler may evaluate x*y-x*z as x*(y-z) but it may not evaluate a+(b-c) as (a+b)-c.

FLOWGRAPH

A compiler first converts the source code of any programming language into an intermediate code. It is then converted into basic blocks. After dividing an intermediate code in basic blocks, the flow of control among basic blocks is represented by a flow graph.

Properties of Flow Graphs

1. The control flow graph is process-oriented.
2. A control flow graph shows how program control is parsed among the blocks.
3. The control flow graph depicts all of the paths that can be traversed during the execution of a program.
4. It can be used in software optimization to find unwanted loops.

Representation of Flow Graphs

Flow graphs are directed graphs. The nodes/bocks of the control flow graph are the basic blocks of the program. There are two designated blocks in Control Flow Graph:

1. Entry Block: The entry block allows the control to enter in the control flow graph.
2. Exit Block: Control flow leaves through the exit block.

An edge can flow from one block A to another block B if:

1. the first instruction of the B's block immediately follows the last instruction of the A's block.
2. there is a conditional/unconditional jump from A's end to the starting of B.
3. B follows X in the original order of the three-address code, and A does not end in an unconditional jump.

Let's see an example,

Consider the source code for converting a 10 x 10 matrix to an identity matrix.

```
for r from 1 to 10 do
    for c from 1 to 10 do
        a [ r, c ] = 0.0;

for r from 1 to 10 do
      a [ r, c ] = 1.0;
```

The following are the three address codes for the above source code:

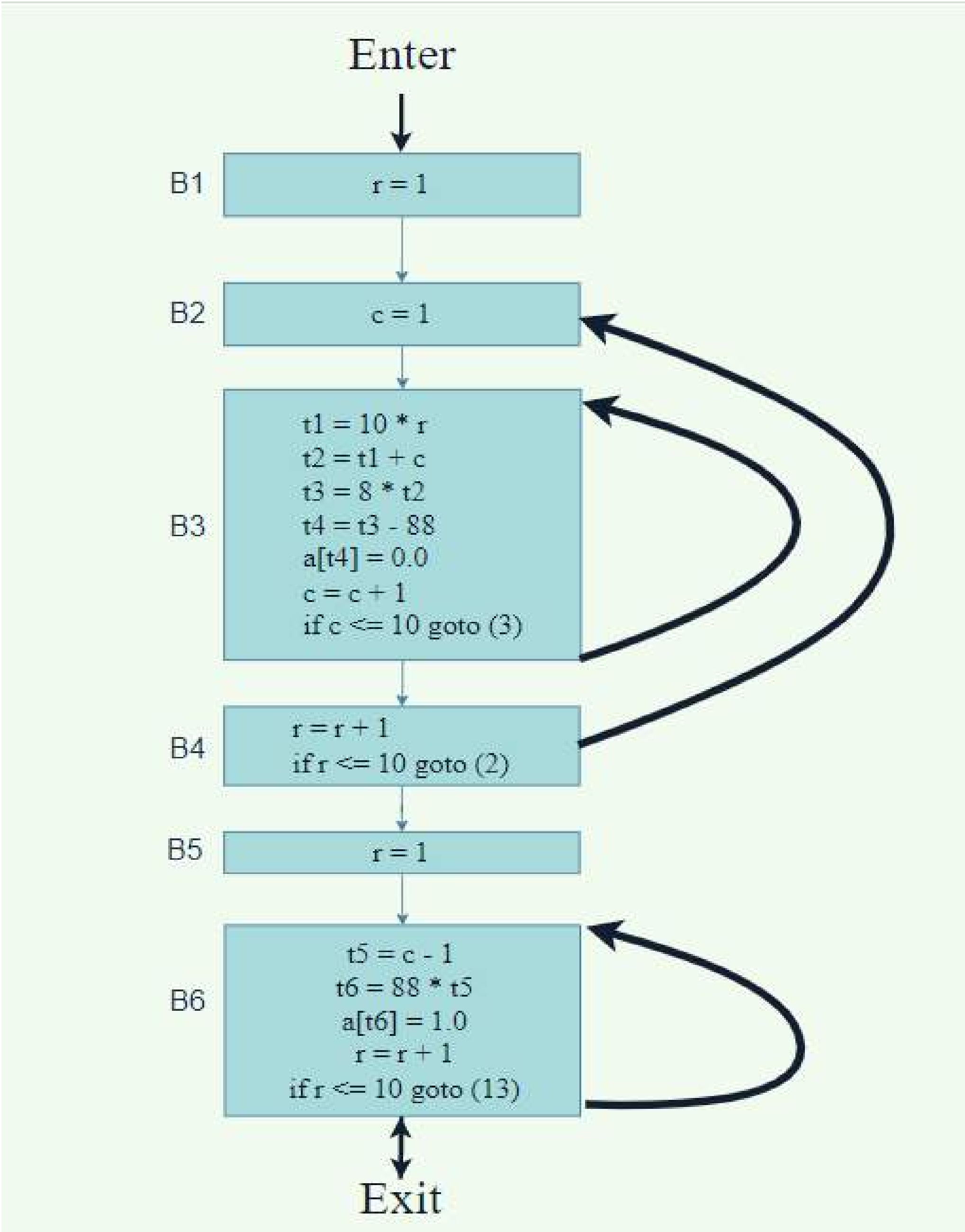
```
1) r = 1
 2) c = 1
 3) t1 = 10 * r
 4) t2 = t1 + c
 5) t3 = 8 * t2
 6) t4 = t3 - 88
 7) a[t4] = 0.0
 8) c = c + 1
 9) if c <= 10 goto (3)
 10) r = r + 1
 11) if r <= 10 goto (2)
```

12) r = 1
13) t5 = c - 1
14) t6 = 88 * t5
15) a[t6] = 1.0
16) r = r + 1
17) if r <= 10 goto (13)

There are six basic blocks for the above-given code, which are:

- B1 for statement 1
- B2 for statement 2
- B3 for statements 3-9
- B4 for statements 10-11
- B5 for statement 12
- B6 for statements 13-17.

The control flow graph of the above-given basic blocks is:



Explanation:

- B1 is the start point for the control flow graph because B1 contains the starting instructions.
- Because B1 does not end with unconditional jumps, and the B2 block's leader immediately follows B1's leader, B2 is the only successor of B1.
- There are two successors to the B3 block. The conditional jump in the last instruction of block B3 is targeted at the first instruction of the B3 block; therefore, one is block B3 itself. Another is block B4 due to conditional jump at the end of the B3 block.
- The last block, B6, is the exit point of the control flow graph.

<u>LOOPSINFLOWGRAPH</u>

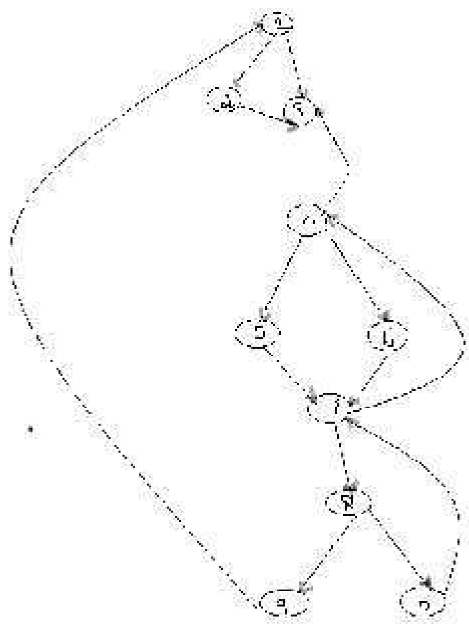
A graph representation of three-address statements, called a flow graph, is useful for understanding code-generation algorithms, even if the graph is not explicitly constructed by a code-generation algorithm. Nodes in the flow graph represent computations, and the edges represent the flow of control.

Dominators:

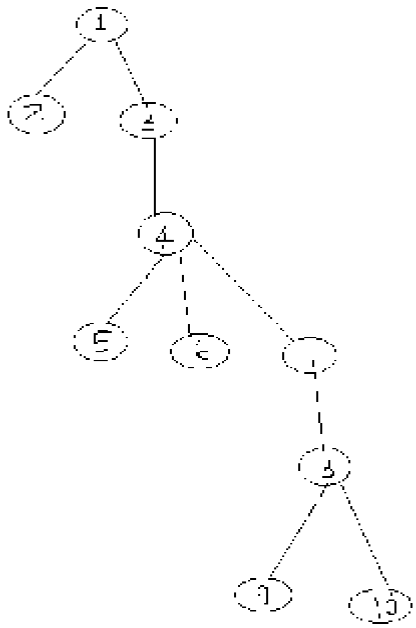
In a flow graph, a node d dominates node n, if every path from initial node of the flow graph to n goes through d. This will be denoted by *d dom n*. Every initial node dominates all the remaining nodes in the flow graph and the entry of a loop dominates all nodes in the loop. Similarly every node dominates itself.

Example:

*Intheflowgraphbelow,
*Initialnode,node1dominateseverynode.
*node2dominatesitself
*node3dominatesallbut1and2.
*node4dominatesallbut1,2and3.
*node5and6dominatesonlythemselves,sinceflowofcontrolcanskiparoundeitherbygoin through the other.
*node7dominates7,8,9and10.
*node8dominates8,9and10.
*node9and10dominatesonlythemselves.



- Thewayofpresentingdominatorinformationisinatree,calledthedominatortreein which the initial node is the root.
- Theparentofeachothernodeisitsimmediatedominator.
- Eachnodeddominatesonlyitsdescendentsinthetree.
- The existence of dominator tree follows from a property of dominators; each node has a unique immediate dominator in that is the last dominator of n on anypath from the initial node to n.
- In terms of the dom relation, the immediate dominator m has the property is d=!n and d dom n, then d dom m.



D(1)={1}

D(2)={1,2}

D(3)={1,3}

D(4)={1,3,4}

D(5)={1,3,4,5}

D(6)={1,3,4,6}

D(7)={1,3,4,7}

D(8)={1,3,4,7,8}

D(9)={1,3,4,7,8,9}

D(10)={1,3,4,7,8,10}

NaturalLoop:

- Oneapplicationofdominatorinformationisindeterminingtheloopsofaflowgraphsuitable for improvement.

- Thepropertiesofloopsare

  - A loop must have a single entry point, called the header. This entry point-dominates all nodes in the loop, or it would not be the sole entry to the loop.
  - Theremustbeatleastonewaytoiteratetheloop(i.e.)atleastonepathbacktotheheader.

- One way to find all the loops in a flow graph is to search for edges in the flow graph whose heads dominate their tails. If a→b is an edge, b is the head and a is the tail. These types of edges are called as back edges.

Example:

Intheabovegraph,

7→4 4DOM7

10→7 7DOM10

4→3

8→3

2 →1

- Theaboveedgeswillformloopinflowgraph.
- Givenaback edgen → d,wedefinethenaturalloopoftheedgetobedplustheset ofnodes that can reach n without going through d. Node d is the header of the loop.

  - Algorithm:Constructingthenaturalloopofabackedge.

Input:AflowgraphGandabackedgen→d.
Output:Thesetloopconsistingofallnodesinthenaturalloopn→d.

Method: Beginning with node n, we consider each node m*d that we know is in loop, to make surethatm'spredecessorsare alsoplacedinloop. Eachnodeinloop, exceptford,isplacedonce onstack, so itspredecessorswillbe examined. Notethatbecause dis putin theloopinitially, we never examine its predecessors, and thus find only those nodes that reach n without going through d.

Procedureinsert(m);
ifmisnotin*loop*thenbegin *loop* := *loop* U {m}; push *m* onto *stack*
end;

stack:=empty;

loop:={*d*};
insert(*n*);
while*stack*isnotemptydobegin
pop*m*,thefirstelementof*stack*,off*stack*;
foreachpredecessor*p*of*m*do*insert(p)*
end

Innerloop:

- If we use the natural loops as “the loops”, then we have the useful property that unless two loops have the same header, they are either disjointed or one is entirely contained in theother.Thus,neglectingloops withthesameheaderforthemoment,wehaveanatural notion of inner loop: one that contains no other loop.
- When two natural loopshavethesameheader, but neitheris nested withinthe other, they are combined and treated as a single loop.

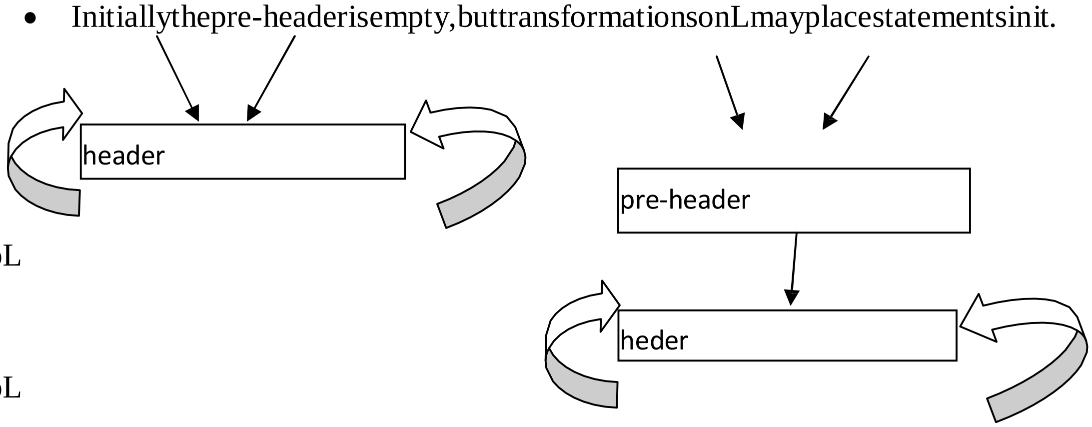
Pre-Headers:

- Several transformations require us to move statements “before the header”. Therefore begin treatment of a loop L by creating a new block, called the preheater.

- The pre-header has only the header as successor, and all edges which formerly enteredthe header of L from outside L instead enter the pre-header.

- EdgesfrominsideloopLtotheheaderarenotchanged.

- Initiallythepre-headerisempty,buttransformationsonLmayplacestatementsinit.



loopL

loopL

(a)Before (b)After

<u>Reducibleflowgraphs:</u>

- Reducible flow graphs are special flow graphs, for which several code optimization transformations are especially easy to perform, loops are unambiguously defined, dominators can be easily calculated, data flow analysis problems can also be solved efficiently.

- Exclusive use of structured flow-of-control statements such as if-then-else, while-do, continue, and break statements produces programs whose flow graphs are always reducible.

- The most important properties of reducible flow graphs are that there are no jumps into the middle of loops from outside; the only entry to a loop is through its header.

- *Definition*:

AflowgraphGisreducibleifandonlyifwecanpartitiontheedgesintotwodisjoint groups, *forward* edges and *back* edges, with the following properties.

- ➢ Theforwardedgesfrom anacyclicgraphinwhicheverynodecanbereachedfrominitial node of G.

- ➢ Thebackedgesconsistonlyofedgeswhereheadsdominatetheirstails.

- ➢ Example:Theaboveflowgraphisreducible.

- If we know the relation DOM for a flow graph, we can find and remove all the back edges.

- Theremainingedgesareforwardedges.

- Iftheforwardedgesformanacyclic graph,thenwecansaytheflowgraphreducible.

- In the above example remove the five back edges 4→3, 7→4, 8→3, 9→1 and 10→7 whose heads dominate their tails, the remaining graph is acyclic.

- The key property of reducible flow graphs for loop analysis is that in such flow graphs every set of nodes that we would informally regard as a loop must contain a back edge.

PEEPHOLE OPTIMIZATION

• A statement-by-statement code-generations strategy often produce target code that contains redundant instructions and suboptimal constructs .The quality of such target code can be improved by applying "optimizing" transformations to the target program.

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

• The peephole is a small, moving window on the target program. The code in the peephole need not contiguous, although some implementations do require this.it is characteristic of peephole optimization that each improvement may spawn opportunities for additional improvements.

• We shall give the following examples of program transformations that are characteristic of peephole optimizations:

✓ Redundant-instructions elimination
✓ Flow-of-control optimizations
✓ Algebraic simplifications
✓ Use of machine idioms

✓ Unreachable Code

Redundant Loads and Stores:

If we see the instructions sequence

(1) MOV R_0,a

(2) MOV a,R_0
we can delete instructions (2) because whenever (2) is executed. (1) will ensure that the value of a is already in register R_0.If (2) had a label we could not be sure that (1) was always executed immediately before (2) and so we could not remove (2).

Unreachable Code:

• Another opportunity for peephole optimizations is the removal of unreachable instructions. An unlabeled instruction immediately following an unconditional jump may be removed. This operation can be repeated to eliminate a sequence of instructions. For example, for debugging purposes, a large program may have within it certain segments that are executed only if a variable debug is 1. In C, the source code might look like:

```
#define debug 0
….
If ( debug ) {
Print debugging information
}
```

• In the intermediate representations the if-statement may be translated as:

```
If debug =1 goto L2 goto L2
L1: print debugging information
L2: ............................................................................................ (a)
```

• One obvious peephole optimization is to eliminate jumps over jumps .Thus no matter what the value of debug; (a) can be replaced by:

```
If debug ≠1 goto L2
Print debugging information
L2: ............................................................................................ (b)
```

• As the argument of the statement of (b) evaluates to a constant true it can be replaced by

36

```
If debug ≠0 goto L2
Print debugging information
L2: ............................................................................................ (c)
```

• As the argument of the first statement of (c) evaluates to a constant true, it can be replaced by goto L2. Then all the statement that print debugging aids are manifestly unreachable and can be eliminated one at a time.

Flows-Of-Control Optimizations:

• The unnecessary jumps can be eliminated in either the intermediate code or the target code by the following types of peephole optimizations. We can replace the jump sequence

```
goto L1
….
L1: gotoL2 by the
sequence
goto L2
….
L1: goto L2
```

• If there are now no jumps to L1, then it may be possible to eliminate the statement L1:goto L2 provided it is preceded by an unconditional jump .Similarly, the sequence

```
if a < b goto L1
….
L1: goto L2 can be
```

replaced by
If a < b goto L2
….
L1: goto L2
• Finally, suppose there is only one jump to L1 and L1 is preceded by an unconditional goto. Then the sequence goto L1
……..
37
L1: if a < b goto L2
L3: .... (1)
• May be replaced by
If a < b goto L2 goto L3
…….
L3: .... (2)
• While the number of instructions in (1) and (2) is the same, we sometimes skip the unconditional jump in (2), but never in (1).Thus (2) is superior to (1) in execution time

Algebraic Simplification:
• There is no end to the amount of algebraic simplification that can be attempted through peephole optimization. Only a few algebraic identities occur frequently enough that it is worth considering implementing them .For example, statements such as
x := x+0 Or
x := x * 1
• Are often produced by straightforward intermediate code-generation algorithms, and they can be eliminated easily through peephole optimization.

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.
$X_2 \rightarrow X*X$

Use of Machine Idioms:
• The target machine may have hardware instructions to implement certain specific operations efficiently. For example, some machines have auto-increment and auto-decrement addressing modes. These add or subtract one from an operand before or after using its value.
• The use of these modes greatly improves the quality of code when pushing or popping a stack, as in parameter passing. These modes can also be used in code for statements like
i : =i+1.

i:=i+1 → i++
i:=i-1 →i- -

Techniques in local loops and global optimization:

Loop Optimization is the process of increasing execution speed and reducing the overheads associated with loops. It plays an important role in improving cache performance and making effective use of parallel processing capabilities. Most execution time of a scientific program is spent on loops.
Loop Optimization is a machine independent optimization.
Decreasing the number of instructions in an inner loop improves the running time of a program even if the amount of code outside that loop is increased.

Loop Optimization Techniques

In the compiler, we have various loop optimization techniques, which are as follows:

1. Code Motion (Frequency Reduction)
In frequency reduction, the amount of code in the loop is decreased. A statement or expression, which can be moved outside the loop body without affecting the semantics of the program, is moved outside the loop.
Example:
Before optimization:
```
while(i<100)
{
 a = Sin(x)/Cos(x) + i;
i++;
}
```

After optimization:
```
t = Sin(x)/Cos(x);
while(i<100)
{
 a = t + i;
i++;
}
```

2. Induction Variable Elimination
If the value of any variable in any loop gets changed every time, then such a variable is known as an induction variable. With each iteration, its value either gets incremented or decremented by some constant value.

Example:
Before optimization:
```
B1
i:= i+1
x:= 3*i
y:= a[x]
if y< 15, goto B2
```
In the above example, i and x are locked, if i is incremented by 1 then x is incremented by 3. So, i and x are induction variables.
After optimization:
```
B1
i:= i+1
x:= x+4
y:= a[x]
```

```
if y< 15, goto B2
```

3. Strength Reduction

Strength reduction deals with replacing expensive operations with cheaper ones like multiplication is costlier than addition, so multiplication can be replaced by addition in the loop.

Example:

Before optimization:

```
while (x<10)
   {
y := 3 * x+1;
a[y] := a[y]-2;
x := x+2;
    }
```

After optimization:

```
t= 3 * x+1;
while (x<10)
         {
            y=t;
a[y]= a[y]-2;
            x=x+2;
            t=t+6;
         }
```

4. Loop Invariant Method

In the loop invariant method, the expression with computation is avoided inside the loop. That computation is performed outside the loop as computing the same expression each time was overhead to the system, and this reduces computation overhead and hence optimizes the code.

Example:

Before optimization:

```
for (inti=0; i<10;i++)
t= i+(x/y);
...
end;
```

After optimization:

```
s = x/y;
for (inti=0; i<10;i++)
t= i+ s;
...
end;
```

5. Loop Unrolling

Loop unrolling is a loop transformation technique that helps to optimize the execution time of a program. We basically remove or reduce iterations. Loop unrolling increases the program's speed by eliminating loop control instruction and loop test instructions.

Example:

Before optimization:

```
for (inti=0; i<5; i++)
printf("Pankaj\n");
```

After optimization:

```
printf("Pankaj\n");
printf("Pankaj\n");
printf("Pankaj\n");
printf("Pankaj\n");
printf("Pankaj\n");
```

6. Loop Jamming

Loop jamming is combining two or more loops in a single loop. It reduces the time taken to compile the many loops.

Example:
Before optimization:

```
for(inti=0; i<5; i++)
    a = i + 5;
for(inti=0; i<5; i++)
    b = i + 10;
```

After optimization:

```
for(inti=0; i<5; i++)
{
 a = i + 5;
 b = i + 10;
}
```

7. Loop Fission

Loop fission improves the locality of reference, in loop fission a single loop is divided into multiple loops over the same index range, and each divided loop contains a particular part of the original loop

Example:
Before optimization:

```
for(x=0;x<10;x++)
{
a[x]=…
b[x]=…
}
```

After optimization:

```
for(x=0;x<10;x++)
a[x]=…
for(x=0;x<10;x++)
b[x]=…
```

8. Loop Interchange
In loop interchange, inner loops are exchanged with outer loops. This optimization technique also improves the locality of reference.
Example:
Before optimization:

```
for(x=0;x<10;x++)
for(y=0;y<10;y++)
a[y][x]=…
```

After optimization:
```
for(y=0;y<10;y++)
for(x=0;x<10;x++)
a[y][x]=…
```

9. Loop Reversal
Loop reversal reverses the order of values that are assigned to index variables. This help in removing dependencies.

Example:
Before optimization:

```
for(x=0;x<10;x++)
a[9-x]=…
```

After optimization:

```
for(x=9;x>=0;x--)
a[x]=…
```

10. Loop Splitting
Loop Splitting simplifies a loop by dividing it into numerous loops, and all the loops have some bodies but they will iterate over different index ranges. Loop splitting helps in reducing dependencies and hence making code more optimized.

Example:
Before optimization
```
for(x=0;x<10;x++)
if(x<5)
a[x]=…
else
b[x]=…
```

After optimization:

```
for(x=0;x<5;x++)
a[x]=…
for(;x<10;x++)
b[x]=…
```

11. Loop Peeling

Loop peeling is a special case of loop splitting, in which a loop with problematic iteration is resolved separately before entering the loop.

Before optimization:

```
for(x=0;x<10;x++)
if(x==0)
a[x]=…
else
b[x]=…
```

After optimization:

```
a[0]=…
for(x=1;x<100;x++)
b[x]=…
```

12. Unswitching

Unswitching moves a condition out from inside the loop, this is done by duplicating loop and placing each of its versions inside each conditional clause.

Before optimization:

```
for(x=0;x<10;x++)
if(s>t)
a[x]=…
else
b[x]=…
```

After optimization:

```
if(s>t)
for(x=0;x<10;x++)
a[x]=…
else
for(x=0;x<10;x++)
b[x]=…
```