UNIT–IV

RuntimeEnvironments,Stackallocationofspace,accesstoNonLocaldateonthestackHeapManagement code generation–Issues in design of code generation the target Language Address in thetargetcodeBasicblocksandFlowgraphs.ASimpleCodegeneration.

UNIT 4RUNTIMEENVIRONMENT

By **runtime**, we mean a program in execution. **Runtime environment** is a state of the targetmachine, which may include software libraries, **environment** variables, etc., to provide services to theprocesses running in thesystem.

StorageOrganization

- Whenthetargetprogramexecutesthenitrunsinitsownlogicaladdressspaceinwhichthevalueofeach program has alocation.
- The logical address space is shared among the compiler, operating system and target machineformanagementand organization.Theoperatingsystemisused tomapthelogicaladdressintophysicaladdress which is usually spread throughout thememory.

Therun-timerepresentationof anobjectprogramin thelogicaladdress spaceconsistsofdataandprogramareas as shown in Fig. 5.1

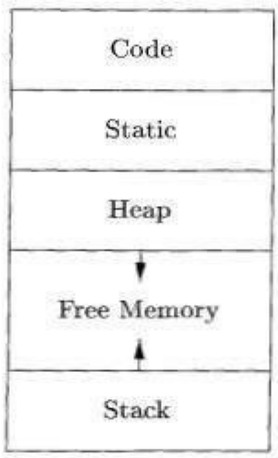


Figure 7.1: Typical subdivision of run-time memory into code and data areas

Storageneeded for anameis determinedfrom its type.

- Runtime storage comes into blocks, where a byte is used to show the smallest unit ofaddressable memory. Using the four bytes a machine word can form. Object of multibyte isstoredin consecutivebytes and gives thefirst byteaddress.
- Run-timestoragecanbe subdividetohold thedifferentcomponentsofanexecuting program:

1. Generatedexecutable code
2. Staticdataobjects
3. Dynamicdata-object-heap
4. Automaticdataobjects-stack

Two areas, *Stack* and *Heap*, are at the opposite ends of the remainder of the address space. These areasare dynamic; their size can change as the program executes. Stack to support call/return policy forprocedures.Heaptostoredatathatcanoutliveacalltoaprocedure..Theheapisusedtomanageallocateand deallocatedata.

StaticVersusDynamicStorageAllocation

The layout and allocation of data to memory locations in the run-time environment are key issues instorage management. The two terms *static* and *dynamic* distinguish between compile time and run time,respectively.Wesay thatastorage-allocation decision is

Static:-if it can be made by the compiler looking only at the text of the program, not at whattheprogram does when it executes.
Dynamic:-if it can bedecidedonly while theprogram is running.

Compilersusefollowingtwostrategiesfordynamicstorageallocation:

Stack storage. Names local to a procedure are allocated space on a stack.stack supports the normalcall/returnpolicy for procedures.

Heap storage. Data that may outlive the call to the procedure that created it is usually allocated on a"heap" of reusable storage.The heap is an area of virtual memory that allows objects or other dataelementstoobtain storagewhen they are createdandto returnthat storage when theyareinvalidated.

Stackallocationofspace

1 ActivationTrees

2 ActivationRecords

3 CallingSequences

4 Variable-LengthData ontheStack

Each time a procedure is called, space for its local variables is pushed onto a stack, and when theprocedureterminates, that spaceis popped off the stack.

1 ActivationTrees

Stackallocationisavalidallocationforproceduressinceprocedurecallsarenested

Example:quicksort algorithm

Themainfunctionhasthreetasks.Itcalls*readArray*,setsthesentinels,andthencalls*quicksort*ontheentiredat aarray.

Procedure activations are nested in time. If an activation of procedure *p* calls procedure *q*, thenthatactivation of*q* must end before theactivation of *p* can end.

```
int a[11];
void readArray() { /* Reads 9 integers into a[1], ..., a[9]. */
    int i;
    ...
}
int partition(int m, int n) {
    /* Picks a separator value v, and partitions a[m..n] so that
       a[m..p-1] are less than v, a[p] = v, and a[p+1..n] are
       equal to or greater than v. Returns p. */
    ...
}
void quicksort(int m, int n) {
    int i;
    if (n > m) {
        i = partition(m, n);
        quicksort(m, i-1);
        quicksort(i+1, n);
    }
}
main() {
    readArray();
    a[0] = -9999;
    a[10] = 9999;
    quicksort(1,9);
}
```

Figure 7.2: Sketch of a quicksort program

Representtheactivationsofproceduresduringthe runningofanentireprogrambya tree,calledan activation tree. Each node corresponds to one activation, and the root is the activation of the "main"procedure that initiates execution of the program. At a node for an activation of procedure p, thechildrencorrespond to activations of theprocedurescalled by this activation of p.

```
enter main()
    enter readArray()
    leave readArray()
    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()
```

Figure 7.3: Possible activations for the program of Fig. 7.2

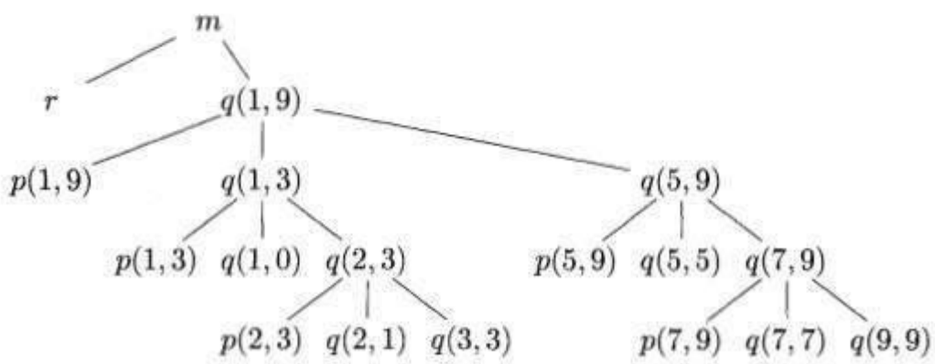


Figure 7.4: Activation tree representing calls during an execution of *quicksort*

2 ActivationRecords

a. Procedure calls and returns are usually managed by a run-time stack called the controlstack.
b. Eachliveactivationhasanactivationrecord(sometimescalledaframe)
c. Theroot ofactivation treeis at thebottomof thestack
d. Thecurrentexecution pathspecifies thecontent ofthestackwith thelast
e. Activationhasrecordinthetopofthestack.

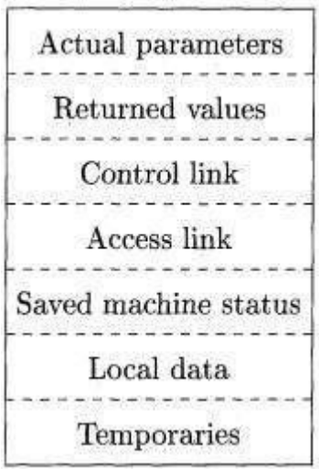


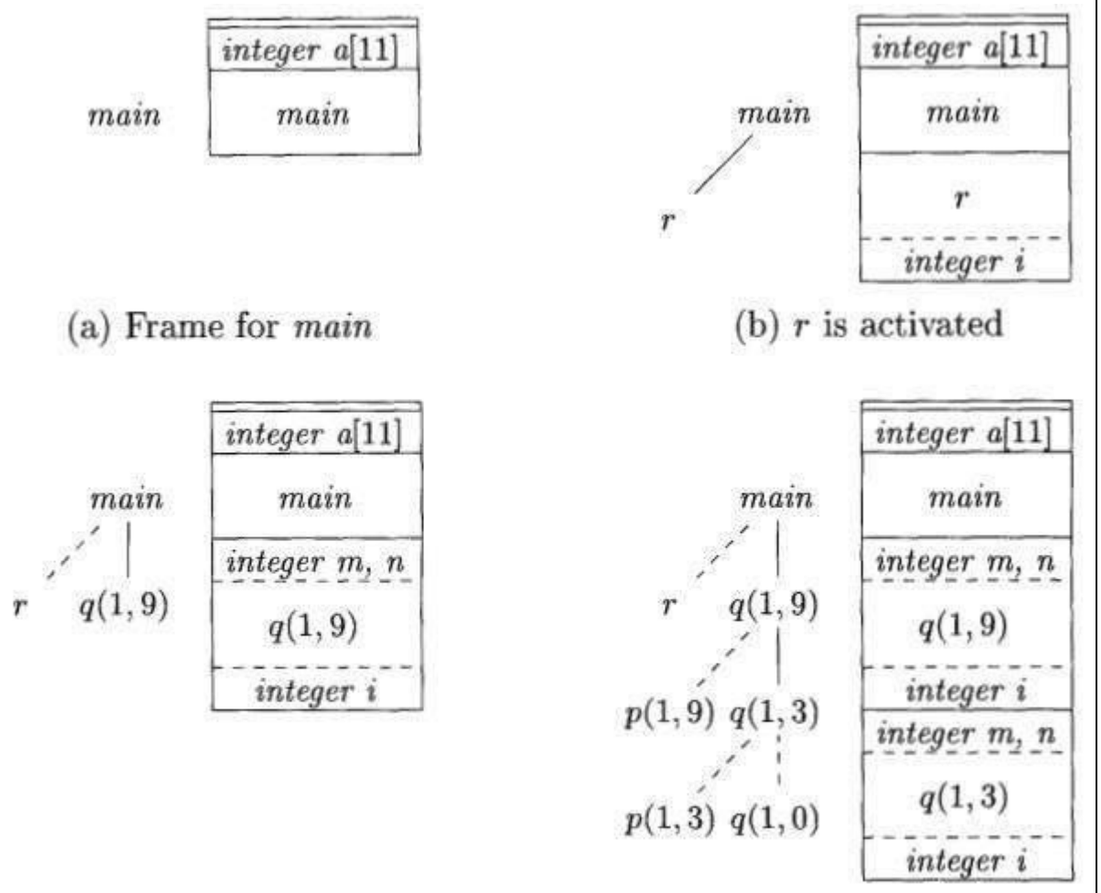
Figure7.5:Ageneralactivationrecord

An activation record is used to store information about the status of the machine, such as the valueof the program counter and machine registers, when a procedure call occurs. When control returns fromthe call, the activation of the calling procedure can be restarted after restoring the values of relevantregisters and setting the program counter to the point immediately after the call. Data objects whoselifetimesarecontainedinthatofanactivationcanbeallocatedonthestackalongwithotherinformationass ociated with the activation.

Anactivationrecordcontainsallthenecessaryinformationrequiredtocallaprocedure.Anactivationrecordmay contain the following units(depending upon thesourcelanguageused).

Temporaries	Storestemporaryandintermediatevaluesofanexpression.
LocalData	Storeslocaldataofthecalledprocedure.
MachineStatus	StoresmachinestatussuchasRegisters,ProgramCounteretc.,beforetheproce dureis called.
Control Link	Storestheaddressof activationrecordofthecallerprocedure.
AccessLink	Storestheinformationof datawhichisoutsidethe local scope.
ActualParameters	Storesactualparameters,i.e.,parameterswhich areusedtosendinputtothecalledprocedure.

ReturnValue	Storesreturnvalues.



(a) Frame for *main*

(b) r is activated

(c) r has been popped and $q(1,9)$ pushed

(d) Control returns to $q(1,3)$

Figure 7.6: Downward-growing stack of activation records

3 CallingSequences

Designingcallingsequencesandthelayoutofactivationrecords,thefollowing

1. Valuescommunicatedbetweencallerandcalleearegenerallyplacedatthe beginningofcallee'sactivationrecord
2. Fixed-length items: are generally placed at the middle. such items typically include the controllink,the access link, and the machine status fields.
3. Itemswhosesizemay notbe knownearlyenough:areplaced atthe endofactivationrecord4.Wemustlocatethetop-of-stackpointerjudiciously: acommonapproachistohave itpointtothe end offixed lengthfields in theactivation record.

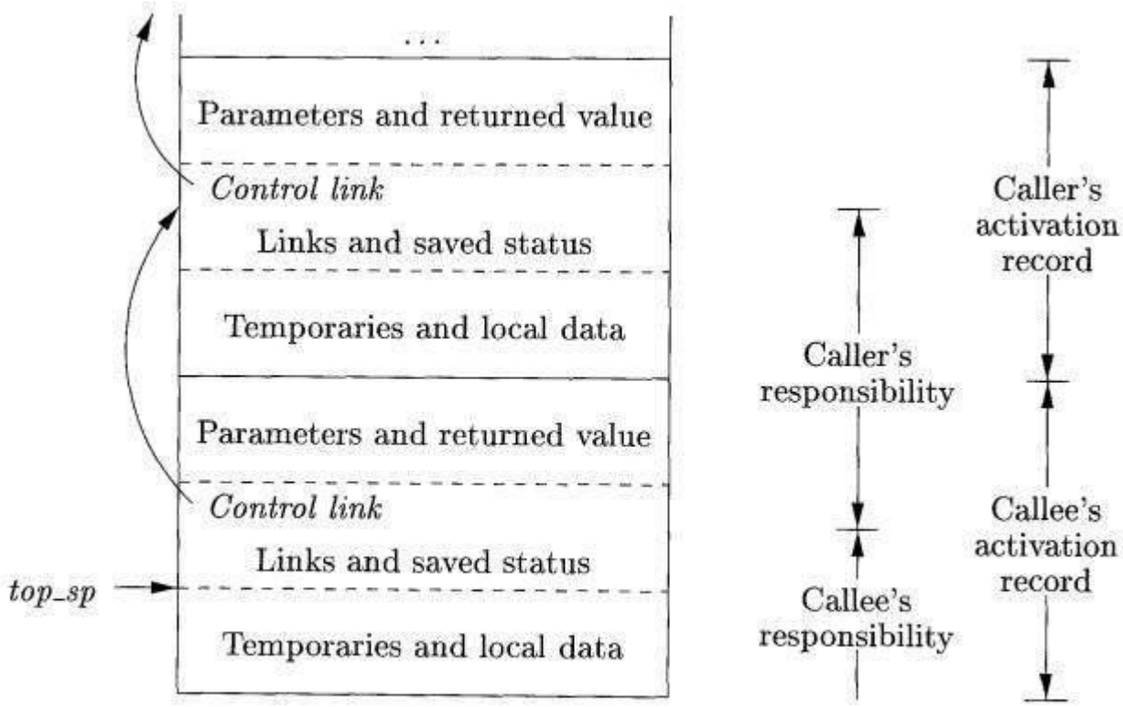


Figure 7.7: Division of tasks between caller and callee

A register topsp points to the end of the machine-status field in the current top activation record.This position within the callee's activation record is known to the caller, so the caller can be maderesponsible for settingtopsp before control is passed to the callee.The calling sequence anditsdivisionbetween caller and calleeis as follows:

1. Thecallerevaluatestheactualparameters.

The caller stores a return address and the old value of *topsp* into the callee's activation record.The caller then increments *topsp* to the position shown in Fig. 7.7. That is, *topsp* is moved past thecaller'slocal dataand temporariesand thecallee'sparameters and status fields.

Thecalleesavestheregistervaluesandotherstatusinformation.Thec

alleeinitializes its local dataand begins execution.

Asuitable,corresponding **returnsequence**is:

1. Thecalleeplaces thereturnvalue nexttothe parameters, asin Fig.7.5.
2. Using information in the machine-status field, the callee restores *topsp* and other registers,andthen branchestothe returnaddress thatthecaller placedin thestatusfield.
3. Although*topsp* hasbeen decremented,thecallerknows wherethereturnvalue is,relativeto

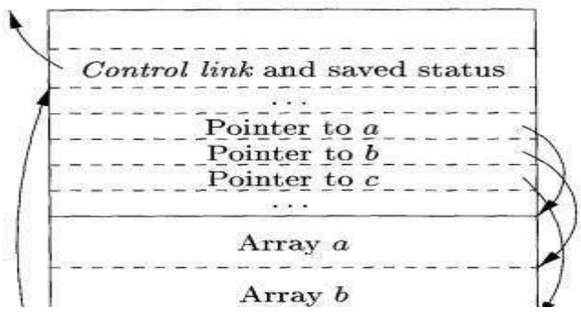
thecurrentvalueof*topsp;*thecaller thereforemayusethatvalue.

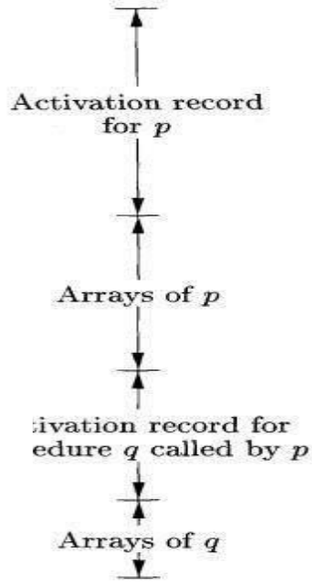
4. Variable-LengthDataontheStack

The run-time memory-management system must deal frequently with the allocation of space forobjects the sizes of which are not known at compile time, but which are local to a procedure and thusmaybeallocatedon the stack.

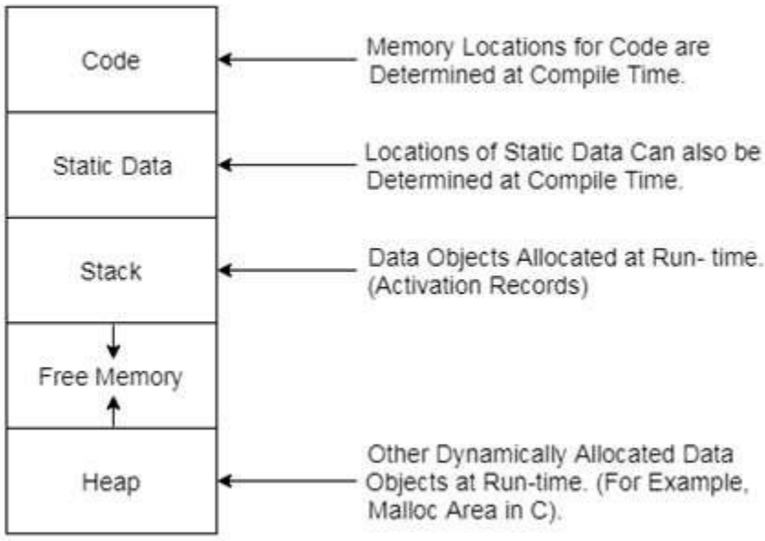
it ispossible to allocate objects, arrays, or other structures of unknown size on the stack. Thereason to prefer placingobjects on the stack if possible is that we avoid the expense of garbagecollecting their space. Note that the stack can be used only for an object if it is local to a procedure andbecomesinaccessible when the procedure returns.

A common strategy for allocating variable-length arrays (i.e., arrays whose size depends on thevalue of one or more parameters of the called procedure) is shown in Fig. 7.8. The same scheme worksfor objects of any type if they are local to the procedure called and have a size that depends on theparametersofthecall.





ted arrays



ed by p. The activation record for are located beyond that . Access

AccesstoNonLocaldataonthestack

1DataAccessWithoutNestedProcedures2Is

sues WithNested Procedures

3ALanguageWithNestedProcedureDeclarations4N

esting Depth

5 AccessLinks

6 ManipulatingAccessLinks

7 AccessLinksforProcedureParameters8

Displays

Consider how procedures access their data. Especially im-portant is the mechanism for findingdataused within aprocedure*p* but that does not belong to*p*

1DataAccessWithoutNestedProcedures

Namesareeither localto theprocedureinquestion oraredeclaredglobally.

1. For global names the address is known statically at compile time providing there is only onesourcefile.Ifmultiplesourcefiles,thelinker knows.Ineithercasenoreferencetotheactivationrecordis needed; theaddresses areknow priorto execution.
2. For names local to the current procedure, the address needed is in the AR at a known-at-compile-timeconstantoffsetfromthesp.Inthecaseofvariablesize arrays,theconstantoffsetrefersto apointer to theactual storage.

2IssuesWithNestedProcedures

Access becomes far more complicated when a language allows procedure dec-larations to benested .The reason is that knowing at compile time thatthe declaration of *p* is immediately nestedwithin*q* doesnottellustherelativepositionsoftheiractivationrecordsatruntime.Infact,sinceeither*p* or*q* orbothmayberecursive,theremaybeseveralactivationrecordsof*p* and/or*q*onthestack.

Finding the declaration that applies to a nonlocal name *x in a* nested pro-cedure *p* is a staticdecision; it can be done by an extension of the static-scope rule for blocks. Suppose *x* is declared in theenclosingprocedure *q*. Findingtherelevantactivationof *q* fromanactivationof *p* isadynamicdecision; it re-quires additional run-time information about activations. One possible solution is to useaccesslinks.

3. ALanguageWithNestedProcedureDeclarations

Invariouslanguageswith nested procedures, one of themostinfluential is ML.

ML is a *functional language*, meaning that variables, once declared and initialized, are notchanged. There are only a few exceptions, such as the array, whose elements can be changed by specialfunctioncalls.

• Variablesaredefined,andhavetheirunchangeablevaluesinitialized,

val (name)=(expression)

• Functionsaredefinedusingthesyntax:

fun(name)((arguments))= (body)

• Forfunction bodies,uselet-statementsof the form:

let (list of definitions) in (statements) endThe definitions are normally v a l or fun statements.Thescopeofeachsuchdefinitionconsistsofallfollowingdefinitions,uptothein,andallthestatements up to the end. Most importantly, function definitions can be nested. For example, the body ofa function p can contain a let-statement that includes the definition of another (nested) function q.Similarly, q can have function definitions within its own body, leading to arbitrarily deep nesting offunction

4. NestingDepth

Nesting depth is 1 to procedures that are not nested within any other procedure. For example, allC functions are at nesting depth 1. However, if a procedure *p* is defined immediately within a procedureat nesting depth*i*, then give *p* the nesting depth*i*

```
1) fun sort(inputFile, outputFile) =
       let
2)         val a = array(11,0);
3)         fun readArray(inputFile) = ··· ;
4)               ··· a ··· ;
5)         fun exchange(i,j) =
6)               ··· a ··· ;
7)         fun quicksort(m,n) =
               let
8)                 val v = ··· ;
9)                 fun partition(y,z) =
10)                      ··· a ··· v ··· exchange ···
               in
11)                ··· a ··· v ··· partition ··· quicksort
               end
       in
12)        ··· a ··· readArray ··· quicksort ···
       end;
```

Figure 7.10: A version of quicksort, in ML style, using nested functions

+1

5. AccessLinks

A direct implementation of the normal static scope rule for nested functions is obtained byadding a pointer called the *access link* to each activation record. If procedure *p* is nested immediatelywithin procedure *q* in the source code, then the access link in any activation of *p* points to the mostrecentactivationof *q*.Note thatthe nestingdepthof*q*mustbe exactlyonelessthan thenestingdepthof p. Access links form a chain from the activation record at the top of the stack to a sequence ofactivationsat progressively lower nesting depths.

Figure 7.11 shows a sequence of stacks that might result from execution of the function *sort* ofFig.7.10.InFig.7.11(a),weseethesituationafter*sort*hascalled*readArray* toloadinputintothearray *a* and then called *quicksort(l,* 9) to sort the array. The access link from *quicksort(l,* 9) points to theactivation record for *sort,* not because *sort* called *quicksort* but because *sort* is the most closely nestedfunctionsurrounding *quicksort*in theprogram.

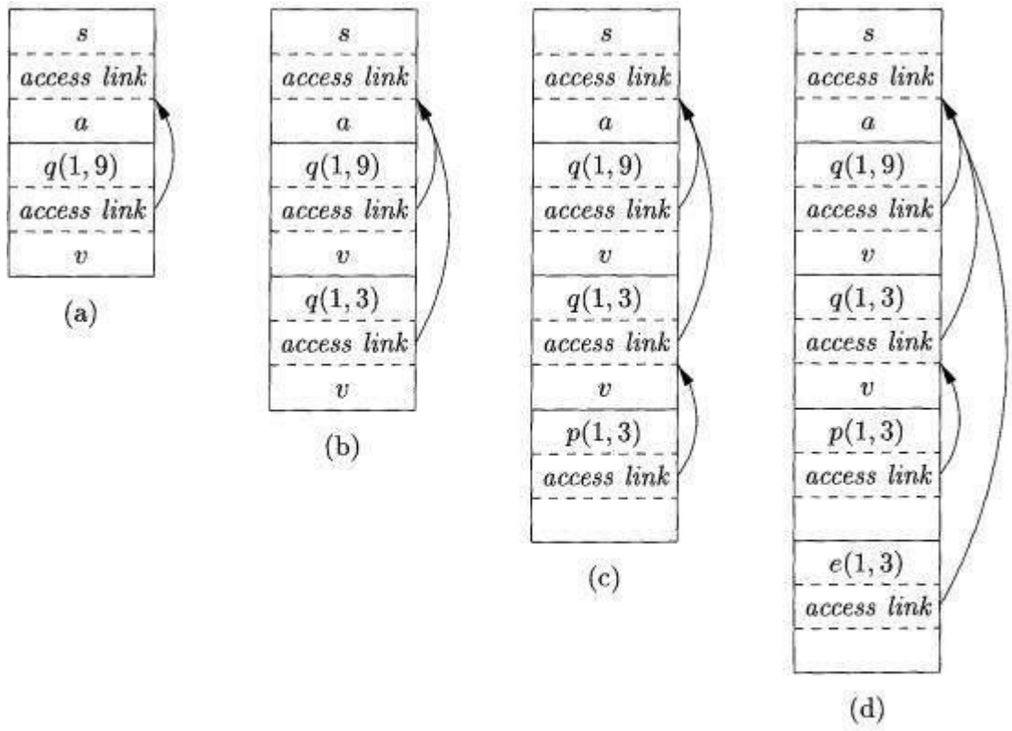


Figure 7.11: Access links for finding nonlocal data

seearecursivecallto*quicksort(l,*3), followedbyacallto*partition,*whichcalls*exchange.*Noticethat*quicksort(l,* 3)'saccess link pointsto*sort,* forthe same reasonthat*quicksort(l,* 9)'s does.

6. ManipulatingAccessLinks

Thehardercaseiswhenthecallistoaprocedure-parameter;inthatcase,theparticularprocedurebeingcalledisnotknownuntilruntime,andthenestingdepthofthecalledproceduremay

differindifferentexecutionsof the call. consider situation when a procedure *q* callsprocedure*p*, explicitly. Therearethreecases:

1. Procedure *p* isatahighernestingdepththan *q*. Thenpmustbedefinedimmediatelywithin *q*, orthecallby *q* wouldnotbeatapositionthatiswithinthescopeoftheprocedurename *p*. Thus,thenestingdepthof *p* isexactlyonegreaterthanthatof *q*, andtheaccesslinkfrom *p* must lead to *q*. It is a simple matter for the calling sequence to include a step that places in theaccesslink for*p* apointer to theactivation recordof *q*.

2. The call is recursive, that is, *p* = *q*.Then the access link for the new activation record is thesameas that of theactivation record below it.

3. Thenestingdepthnpofpislessthanthenestingdepthnqofq.Inorderforthecallwithinqtobe inthe scopeof name p, procedure qmustbe nestedwithin some procedure r,while pisaprocedure defined immediately within r. The top activation record for r can therefore be found byfollowing the chain of access links, starting in the activation record for q, for nq — np + 1 hops. Then,theaccess link forp must go to this activation ofr.

7. AccessLinksforProcedureParameters

Whenaprocedure *p* ispassedtoanotherprocedure *q* asaparameter,and *q* thencallsitsparameter (and therefore calls *p* in this activation of *q),* it is possible that *q* does not know the context inwhich *p* appears in the program. If so, it is impossible for *q* to know how to set the access link for *p*. Thesolution to this is, when procedures are used as parameters, the caller needs to pass, along with the nameof the procedure-parameter, the proper access link for that parameter. The caller always knows the link,since if *p* is passed by procedure *r* as an actual parameter, then *p* must be a name accessible to r, andtherefore,rcandeterminetheaccesslinkfor*p* exactlyasif*p* werebeingcalledby*r*directly.

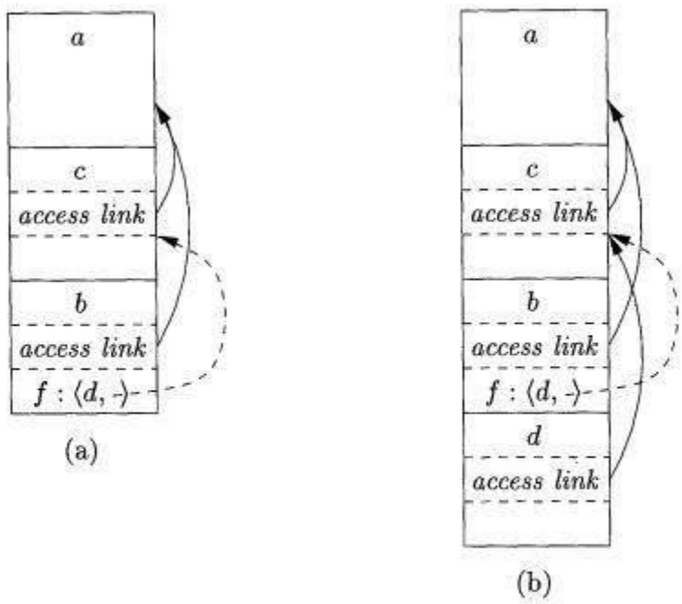


Figure 7.13: Actual parameters carry their access link with them

8. Displays

One problem with the access-link approach to nonlocal data is that if the nesting depth gets large, wemayhavetofollowlongchainsoflinkstoreachthedataweneed.Amoreefficientimplementationuses an auxiliary array *d*, called the *display*, which consists of one pointer for each nesting depth. Wearrange that, at all times, *d[i]* is a pointer to the highest activation record on the stack for any procedureatnesting depth *i*. Examples of adisplay areshown in Fig. 7.14.

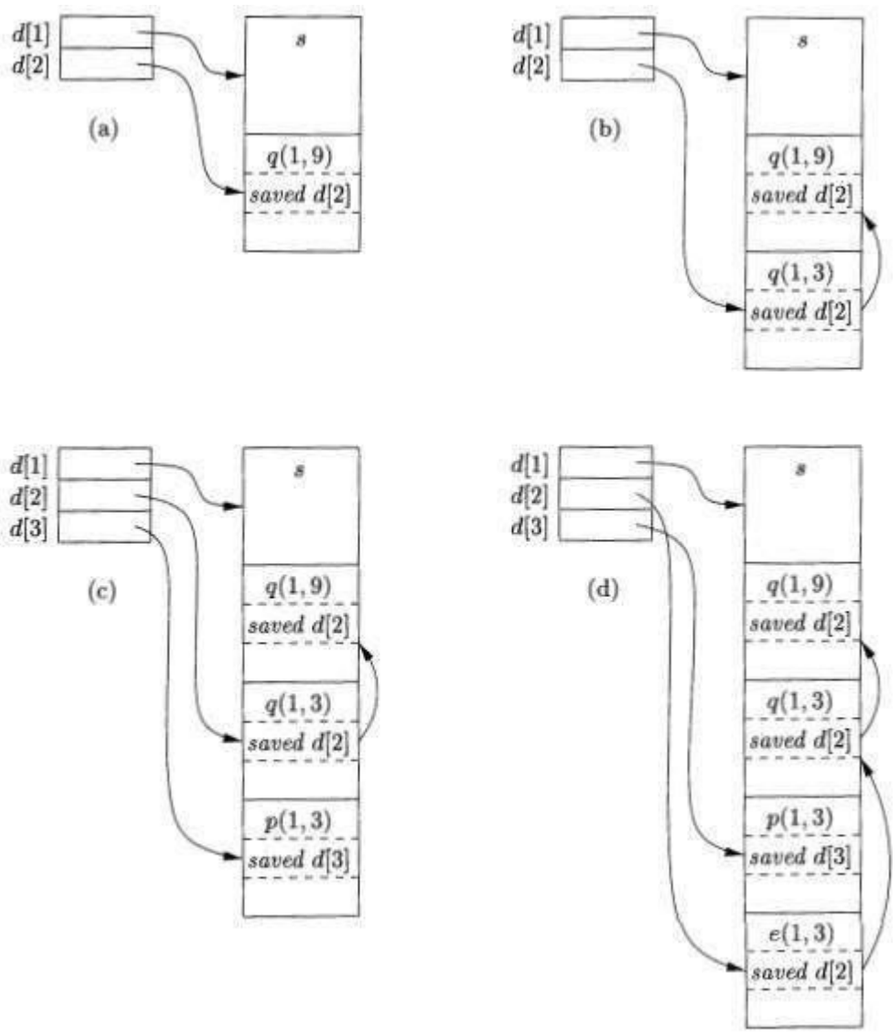


Figure 7.14: Maintaining the display

Inordertomaintainthedisplaycorrectly,weneedtosavepreviousvaluesofdisplayentriesinnewactivation records.

HeapManagement

The heap is the portion of the store that is used for data that lives indefinitely, or until theprogramexplicitly deletes it.

1 TheMemoryManager

2 TheMemoryHierarchyofaComputer3L

ocality in Programs

4 ReducingFragmentation

5 ManualDeallocationRequests

1 TheMemoryManager

It performstwobasicfunctions:

• **Allocation**. Whenaprogramrequestsmemoryforavariableorobject,[3] thememorymanagerproduces a chunk of contiguous heap memory of the requested size. If possible, it satisfies an allocationrequest using free space in the heap; if no chunk of the needed size is available, it seeks to increase theheapstoragespacebygettingconsecutivebytesofvirtualmemoryfromtheoperatingsystem.Ifspaceisexhausted, thememorymanager passesthat informationback to theapplication program.

• **Deallocation**. The memory manager returns deallocated space to the pool of free space, so it can reusethe space to satisfy other allocation requests. Memory managers typically do not return memory to theoperatingsys-tem, even if theprogram's heap usagedrops.

Thus,thememorymanagermustbepreparedtoservice,inanyorder,allo-cationanddeallocation requests of any size, ranging fromone byte to as large as the program's entire addressspace.

Herearetheproperties wedesireofmemory managers:

• **SpaceEfficiency**.Amemorymanagershouldminimizethetotalheapspaceneededbyaprogram.Larger programs to run in afixed virtualaddress space..

• **Program Efficiency.** A memory manager should make good use of the memory subsystem to allowprogramsto runfaster.

• **Low Overhead**. Because memory allocations and deallocations are fre-quent operations in manyprograms, it is important that these operations be as efficient as possible. That is, we wish to minimizethe*overhead*

2. TheMemoryHierarchyofaComputer

The efficiency of a program is determined not just by the number of instructions executed, butalso by how long it takes to execute each of these instructions. The time taken to execute an instructioncanvarysignificantly,sincethetimetakentoaccessdifferentpartsofmemorycanvaryfrom

nanosecondstomilliseconds.Data-intensiveprogramscanthereforebenefitsignificantlyfromoptimizationsthat makegood useof thememorysubsystem.

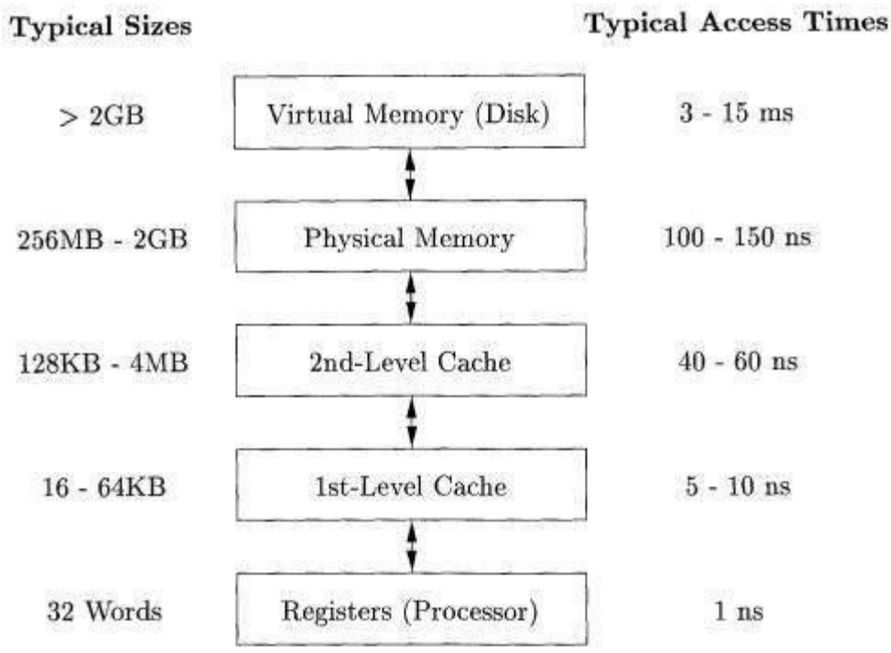


Figure 7.16: Typical Memory Hierarchy Configurations

3. LocalityinPrograms

Most programs exhibit a high degree of locality;that is, they spend most of their time executinga relatively small fraction of the code and touching only a small fraction of the data. We say thataprogram hastemporal locality if the memory locations it accesses are likely to be accessed again withina short period of time. We say that a program has *spatial locality* if memory locations close to thelocationaccessed arelikely also to beaccessed within a short period oftime.

Programs spend 90% of their time executing 10% of the code. Programs often contain manyinstructions that are never executed. Programs built with components and libraries use only a smallfractionofthe provided functionality.

The typicalprogramspendsmostofits timeexecutinginnermostloopsandtight recursivecycles in a program. By placing the most common instructions and data in the fast-but-small storage,while leaving the rest in the slow-but-large storage. Average memory-access time of a program can beloweredsignificantly.

4. ReducingFragmentation

Tobeginwiththewhole heapisasinglechunkofsize

500KbytesAfterafewallocations anddeallocations,therareholes

Intheabove picture,it isnot possibletoallocate100Kor 150Keventhoughtotal freememoryis150K

With each deallocation request, the freed chunks of memory are added back to the pool of freespace.We coalesce contiguous holes into larger holes, as the holes can only get smaller otherwise.Ifwe are not careful, the memory may end up getting fragmented, consisting of large numbers of small,noncontiguous holes. It is then possible that no hole is large enough to satisfy a future request, eventhoughtheremay besufficient aggregate freespace.

Best -FitandNext- FitObject Placement

We reduce fragmentation by controlling how the memory manager places new objects in theheap. It has been found empirically that a good strategy for minimizing fragmentation for real lifeprogramsistoallocatetherequestedmemoryinthesmallestavailableholethatislargeenough.This *best-fit* algorithmtendstosparethelargeholestosatisfysubsequent,largerrequests.Analternative, called *first-fit,* where an object is placed in the first (lowest-address) hole in which it fits,takesless timetoplaceobjects,but has beenfound inferiortobest-fit in overallperformance.

To implement best-fit placement more efficiently, we can separate free space into *bins,* according totheirsizes.Binning makes it easy to find thebest-fit chunk.

Man ag in g and CoalescingFreeSpace

When anobject is deallocated manually, the memory manager must make its chunk free, so itcan be allocated again. In some circumstances, it may also be possible to combine *(coalesce)* that chunkwith adjacent chunks of the heap, to form a larger chunk. There is an advantage to doing so, since wecan always use a large chunk to do the work of small chunks of equal total size, but many small chunkscannothold onelargeobject, as thecombinedchunk could.

Automatic garbage collection can eliminate fragmentation altogether if it moves all the allocatedobjectsto contiguous storage.

5. ManualDeallocationRequests

Inmanualmemorymanagement,wheretheprogrammermustexplicitlyarrangeforthedeallocation of data, as in C and C + + . Ideally, any storage that will no longer be accessed should bedeleted.

ProblemswithManualDeallocation

1. Memoryleaks‰

   Failingtodeletedatathatcannotbereferenced‰Impor

   tantinlongrunning ornonstopprograms ,,

2. Danglingpointerdereferencing‰R

   eferencingdeleteddata,,

Bothareseriousand hardtodebug

GarbageCollection,,

1. Reclamationofchunksof storageholding objectsthat canno longerbeaccessedby aprogram,,
2. GCshouldbeableto determinetypesofobjects‰

   Then,sizeandpointerfieldsofobjects canbedeterminedbythe GC‰

   Languagesinwhich typesofobjects canbedeterminedatcompiletimeorrun-time aretype safe,,

   Javaistypesafe,,

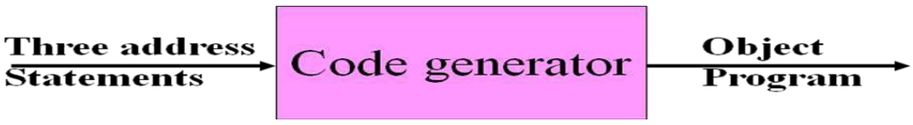
   Cand C++arenottypesafebecausethey permittypecasting,which createsnewpointers

”

   Thus,anymemorylocationcanbe(theoretically)accessedatanytimeandhencecannot beconsidered inaccessible

CodeGeneration

Ittakesasinputtheintermediaterepresentation(IR)producedbythe front endofthecompiler,along with relevant symbol table information, and produces as output a semantically equivalent targetprogram



Themostimportantcriterionforacodegenerator is thatit producecorrectcode.

Thefollowingissuearisesduringthecodegenerationphase:

1InputtotheCodeGenerator2T
heTarget Program
3 InstructionSelection
4 RegisterAllocation
5.EvaluationOrder

Inputtocodegenerator

Theinputtocodegeneratoristheintermediatecodegeneratedbythefrontend,alongwithinformation in the symbol table that determines the run-time addresses of the data-objects denotedby the names in the intermediate representation. Intermediate codes may be represented mostly inquadruples, triples, indirect triples, Postfix notation, syntax trees, DAG's, etc. The code generationphase just proceeds on an assumption that the input are free from all of syntactic and state semanticerrors, the necessary type checking has taken place and the type-conversion operators have beeninsertedwherevernecessary

Targetprogram
Thetargetprogramistheoutputofthecodegenerator.Theoutputmaybeabsolutemachinelanguage ,relocatablemachinelanguage,assemblylanguage.

- **Absolute**machinelanguageas outputhasadvantagesthatitcanbeplacedinafixedmemorylocationandcanbeimmediatelyex ecuted.
- **Relocatable** machine language as an output allows subprograms and subroutines tobe compiled separately. Relocatable object modules can be linked together and loadedbylinkingloader.Butthereisaddedexpenseoflinkingandloading.
- **Assembly** language as output makes the code generation easier. We can generatesymbolic instructions and use macro-facilities of assembler in generating code. And weneedanadditionalassemblystepafter codegeneration.

.

Instructionselection

Selectingthebestinstructionswillimprovetheefficiencyoftheprogram.Itincludestheinstructionsthat should be complete and uniform. Instruction speeds and machine idioms also plays a major rolewhen efficiency is considered. But if we do not care about the efficiency of the target program theninstructionselection is straight-forward.

Forexample,three-addressstatements wouldbetranslated intothelattercodesequenceasshownbelow:

```
P:=Q+RS:=
P+TMOV
Q, R0ADD
R, R0MOV
R0, PMOV
P,
R0ADDT,R
0
```

Herethe fourth statementis redundantas thevalue of theP isloaded againin thatstatementthat justhasbeen stored in the previous statement. It leads to an inefficient code sequence. A given intermediaterepresentation can be translated into many code sequences, with significant cost differences between thedifferent implementations. A prior knowledge of instruction cost is needed in order to design goodsequences,butaccuratecost information is difficultto predict.

Registerallocationissues

Useofregistersmakethe computationsfasterin comparisontothatofmemory,soefficientutilizationofregisters is important.Theuseofregisters aresubdivided into two subproblems:

1. During **Register allocation –** we select only those set of variables that will reside in theregistersat each point inthe program.
2. Duringasubsequent **Registerassignment** phase,thespecificregisteris pickedtoaccessthevariable.

As the number of variables increases, the optimal assignment of registers to variables becomesdifficult. Mathematically, this problem becomes NP-complete. Certain machine requires registerpairsconsist of an evenand next odd-numberedregister. For example

```
Ma, b
```

Thesetypesofmultiplicativeinstructioninvolveregisterpairswherethemultiplicandisan evenregisterand b, themultiplier is theodd register of theeven/odd register pair.

Evaluationorder –
The code generator decides the order in which the instruction will be executed. The order ofcomputationsaffectsthe efficiencyofthetarget code.Amongmanycomputationalorders,some willrequire only fewer registers to hold the intermediate results. However, picking the best order in thegeneralcaseis a difficultNP-complete problem.

Approachestocodegenerationissues:
Codegeneratormustalwaysgeneratethe correctcode.Itis essentialbecauseofthenumberofsp ecialcasesthatacode generatormightface.Someofthe designgoalsofcodegenerator are:

- Correct
- Easilymaintainable
- Testable
- Efficient

ThetargetLanguage

1ASimpleTargetMachineModel2P

rogramand Instruction Costs

ASimpleTargetMachineModel

opsource,destination

Where,opisusedasanop-codeandsourceanddestinationareusedasadatafield.

- It has the following op-codes:ADD(addsourcetodestination)
  SUB (subtract source from destination)MOV(movesource todestination)
- Thesourceanddestinationof aninstructioncanbespecifiedbythecombinationofregistersand memorylocationwith addressmodes.

MODE	FORM	ADDRESS	EXAMPLE	ADDED COST
Absolute	M	M	AddR0, R1	1
Register	R	R	Addtemp, R1	0
indexed	c(R)	C+contents(R)	ADD100(R2),R1	1
indirectregister	*R	contents(R)	ADD*100	0
indirectindexed	*c(R)	contents(c+ contents(R))	(R2),R1	1
literal	#c	c	ADD#3,R1	1

- Here,cost1meansthatitoccupiesonlyonewordofmemory.
- Eachinstructionhas acost of1plusaddedcostsforthesourceanddestination.
- Instruction cost=1 +costisusedfor sourceanddestinationmode.

2 ProgramandInstructionCosts

Cost of an instruction to be one plus the costs associated with the addressing modes of theoperands . This cost corresponds to the length in words of the instruction. Addressing modes involvingregisters have zero additional cost, while those involving a memory location or constant in them have anadditionalcost of one, becausesuch operands havetobe stored in thewords following theinstruction**.**

Examples:

• TheinstructionLDRO,RlcopiesthecontentsofregisterRlintoregisterRO.Thisinstructionhasacost of onebecauseno additionalmemory words arerequired.

• The instruction LD RO, Mloads the contentsof memory location Minto register RO.T h e costistwosincethe address ofmemory locationM is in theword followingtheinstruction.

• The instruction LDR l , *100(R2)loads into register Rl the value given by contents(contents(100 +contents(K2))).Thecost isthreebecause the constant 100 is stored in the word following theinstruction.

Example:

1. Moveregistertomemory R0→M
   MOVR0, M
   cost=1+1+1 (sinceaddressofmemorylocationM isinword followingtheinstruction)

2. Indirectindexedmode:MOV*4(R0),M
   cost = 1+1+1(since one word **for** memory location M, one wordresultof*4(R0) and one**for**instruction)

3. LiteralMode:
   MOV#1, R0
   cost=1+1+1 =3(oneword **for**constant1 and one**for**instruction)

Addressinthetargetcode

Theinformation whichrequired duringanexecutionofaprocedureis keptin ablock ofstoragecalledanactivationrecord. Theactivationrecordincludes storage fornameslocal totheprocedure. Wecandescribeaddressinthetargetcodeusingthefollowingways:

1. Staticallocation
2. Stackallocation

Instaticallocation,theposition ofanactivationrecordisfixedinmemoryat compiletime.

.
Inthestackallocation,foreachexecutionofaprocedureanewactivationrecord ispushedontothestack. Whenthe activation ends then therecord ispopped.

For the run-time allocation and deallocation of activation records the following three-addressstatementsareassociated:

1. Call
2. Return
3. Halt
4. Action,aplaceholderforotherstatements

Assumethattherun-timememoryisdividedintoareasfor:

1. Code
2. Staticdata
3. Stack

Staticallocation:

1. Implementationofcallstatement:

Thefollowingcodeisneeded toimplement staticallocation:

MOV#here +20, callee.static_area /*itsavesreturnaddress*/</p>

GOTOcallee.code_area /*Ittransferscontroltothetargetcodeforthecalledprocedure*/

Where,

callee.static_areashowstheaddressoftheactivationrecord.

callee.code_areashows theaddressofthefirstinstructionforcalledprocedure.

#here+20literalareusedtoreturnaddressofthe instructionfollowingGOTO.

2. Implementationofreturnstatement:

Thefollowingcodeisneededtoimplementreturn

fromprocedurecallee:GOTO* callee.static_area

Itisusedtotransferthecontroltotheaddressthatissavedatthebeginningoftheactivationrecord.

3. Implementationofactionstatement:

TheACTIONinstructionisusedtoimplementactionstatement.

4. Implementationofhaltstatement:

TheHALTstatementis thefinalinstruction thatisused toreturnthe controltotheoperatingsystem.

Stackallocation

Usingtherelativeaddress,staticallocationcanbecomestackallocationfor storageinactivationrecords.

Instackallocation,registerisusedtostorethepositionofactivationrecord sowordsinactivationrecords canbeaccessed asoffsets fromthevalue inthis register.

Thefollowingcodeis neededtoimplementstackallocation:

1. Initializationofstack:

```
MOV#stackstart , SP     /*initializes
 stack*/HALT              /*terminateexecution*/
```

2. ImplementationofCallstatement:

```
ADD #caller.recordsize, SP/* increment stack pointer
*/MOV#here+16, *SP
                                    /*Savereturnaddress*/G
OTOcallee.code_area
```

Where,

caller.recordsizeis thesizeoftheactivationrecord

#here+ 16is theaddressof theinstruction followingthe**GOTO**

3. ImplementationofReturnstatement:

```
GOTO*0(SP)/*returntothecaller*/
 SUB#caller.recordsize,SP          /*decrementSP andrestoreto previous value*/
```

BasicblocksandFlowgraphs

A graph representation of three-address statements, called a flow graph, is useful forunderstanding code-generation algorithms, even if the graph is not explicitly constructed by acode-generation algorithm. Nodes in the flow graph represent computations, and the edgesrepresent the flow of control. Flow graph of a program can be used as a vehicle to collectinformation about the intermediate program. Some register-assignment algorithms use flowgraphsto find theinnerloopswhere aprogram is expectedto spend most ofits time.

Basicblockcontains asequenceof statement.The flowofcontrol entersatthe beginningofthestatementand leaveattheend without anyhalt (except maybethe last instructionofthe block).

Thefollowing sequence of threeaddressstatements forms abasicblock:

1.t1:=x * x
2. t2:= x *
y3.t3:= 2 *
t24.t4:=t1+t3
5. t5:= y *
y6.t6:=t4+t5

Basicblockconstruction:

Algorithm: Partitionintobasicblocks

Input: Itcontainsthesequenceofthree addressstatements

Output:itcontains alistofbasic blockswith eachthreeaddressstatementin exactlyoneblock

Method: Firstidentifytheleaderinthecode.The rules forfindingleaders areasfollows:

- Thefirst statementis aleader.
- StatementLis aleader if thereisan conditionalorunconditionalgoto statementlike: if..... goto L orgoto L
- InstructionLis aleader ifit immediatelyfollowsagotoor conditionalgoto statementlike:ifgoto Bor goto B

For each leader, its basic block consists of the leader and all statement up to. It doesn't include the nextleaderor end ofthe program.

Considerthefollowingsourcecodefordot product oftwo vectorsaand boflength 10:

```
begin
    prod
     :=0;i:=1;
dobegin
    prod :=prod+ a[i] *
    b[i];i:=i+1;
     end
 while i <=
10end
```

Thethree addresscodefor theabovesourceprogramis givenbelow:

B1

```
(1)prod:=0(
 2)i :=1
```

B2

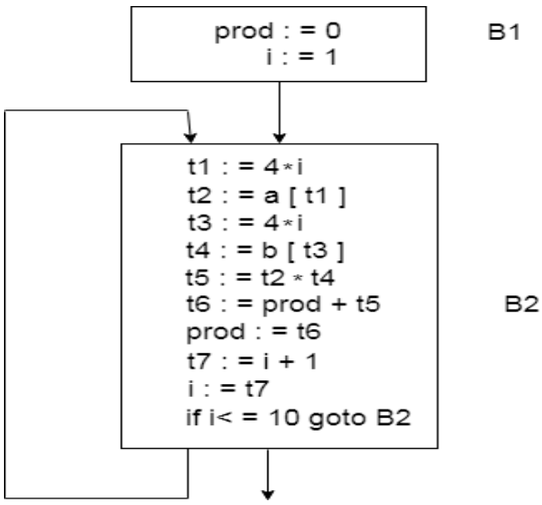
(3) t1 := 4*
i(4)t2:=a[t1]
(5) t3 := 4*
i(6)t4:=b[t3]
(7)t5 :=t2*t4
(8) t6:=prod+t5
(9) prod :=
t6(10) t7:=i+1
(11) i :=t7
(12) ifi<=10 goto(3)

Basic block B1 contains the statement (1) to
(2)BasicblockB2containsthestatement (3)to(12)

FlowGraph

Flowgraphisadirected graph.Itcontainsthe flowofcontrolinformationforthesetofbasic block.

Acontrolflowgraphisusedto depictthathowtheprogramcontrol isbeingparsedamongtheblocks.Itisuseful in theloop optimization.Flow graphfor the vectordot product isgiven as follows:



1.BlockB1 isthe initialnode. BlockB2 immediatelyfollows B1, sofrom B2to B1thereisanedge.

2.Thetarget ofjump fromlast statementofB1is thefirst statementB2, sofrom B1to B2thereisan edge.

ASimpleCodegeneration.

Codegeneratoris usedtoproducethetargetcodeforthree-addressstatements. Itusesregisterstostorethe operands of thethreeaddress statement.

Considerthethree addressstatementx:=y+z.Itcanhavethe followingsequenceof codes:

MOVx, R_0

ADDy, R_0

RegisterandAddressDescriptors:

- Aregisterdescriptorcontainsthetrackof whatiscurrentlyineachregister. Theregisterdescriptorsshow that allthe registers areinitially empty.
- Anaddressdescriptorisusedtostorethe locationwherecurrent valueofthename canbefoundat runtime.

Acode-generationalgorithm:

The algorithm takes a sequence of three-address statements as input. For each three address statement oftheform a:= b opcperform thevarious actions. Theseareas follows:

1. Invokeafunction getregtofind outthe locationLwherethe result ofcomputationb opc shouldbestored.
2. Consulttheaddress descriptionforytodeterminey'.Ifthevalueof ycurrently inmemory andregister both then prefer the register y' . If the value of y is not already in L then generate theinstruction**MOV y' , L**to placeacopy ofyin L.
3. Generate the instruction **OP z' , L** where z' is used to show the current location of z. if z is inboth then prefer a register to a memory location. Update the address descriptor of x to indicatethat x is in location L. If x is in L then update its descriptor and remove x from all otherdescriptor.
4. If the current value of y or z have no next uses or not live on exit from the block or in registerthenaltertheregister descriptortoindicatethat afterexecutionofx :=yopzthoseregisterwillnolongercontain y orz.

GeneratingCodeforAssignment Statements:

Theassignmentstatement d:=(a-b)+(a-c)+(a-c)canbetranslatedinto the following sequenceofthreeaddresscode:

t:=a-b

u:= a-

cv:=t+u

d:=v+u

Codesequenceforthe exampleis as follows:

Statement	CodeGenerated	Registerdescriptor Register empty	Addressdescriptor
t:=a-b	MOVa,R0 SUBb, R0	R0contains t	t in R0
u:= a-c	MOVa,R1 SUBc, R1	R0 contains tR1containsu	t in R0uin R1
v:= t +u	ADDR1, R0	R0 contains vR1containsu	u in R1vin R1
d:= v +u	ADD R1, R0MOVR0, d	R0contains d	d in R0 dinR0and memory