UNIT – V
Runtime Environments, Stack allocation of space, access to Non Local date on the stack Heap Management code generation–Issues in design of code generation the target Language Address in the target code Basic blocks and Flow graphs. A Simple Code generation.

UNIT 5
RUNTIME ENVIRONMENT

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

Storage Organization

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

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

The run-time representation of an object program in the logical address space consists of data and program areas as shown in Fig. 5.1
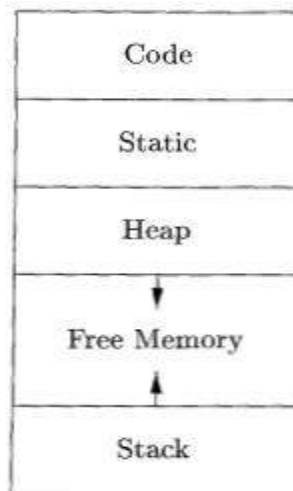

Figure 7.1: Typical subdivision of run-time memory into code and data areas

Storage needed for a name is determined from its type.

o   Runtime storage comes into blocks, where a byte is used to show the smallest unit of addressable memory. Using the four bytes a machine word can form. Object of multibyte is stored in consecutive bytes and gives the first byte address.

o   Run-time storage can be subdivide to hold the different components of an executing program:

1. Generated executable code
2. Static data objects
3. Dynamic data-object- heap
4. Automatic data objects- stack

Two areas, *Stack* and *Heap,* are at the opposite ends of the remainder of the address space. These areas are dynamic; their size can change as the program executes. Stack to support call/return policy for procedures.Heap to store data that can outlive a call to a procedure. . The heap is used to manage allocate and deallocate data.

Static Versus Dynamic Storage Allocation

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

Static:- if it can be made by the compiler looking only at the text of the program, not at what the program does when it executes.

Dynamic:- if it can be decided only while the program is running.

Compilers use following two strategies for dynamic storage allocation:

Stack storage. Names local to a procedure are allocated space on a stack. stack supports the normal call/return policy 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 data elements to obtain storage when they are created and to return that storage when they are invalidated.

Stack allocation of space

1 Activation Trees

2 Activation Records

3 Calling Sequences

4 Variable-Length Data on the Stack

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

1 Activation Trees

Stack allocation is a valid allocation for procedures since procedure calls are nested

Example: quicksort algorithm

The main function has three tasks. It calls *readArray,* sets the sentinels, and then calls *quicksort* on the entire data array.

Procedure activations are nested in time. If an activation of procedure *p* calls procedure *q,* then that activation of *q* must end before the activation 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

Represent the activations of procedures during the running of an entire program by a tree, called an 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, the children correspond to activations of the procedures called 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()
```

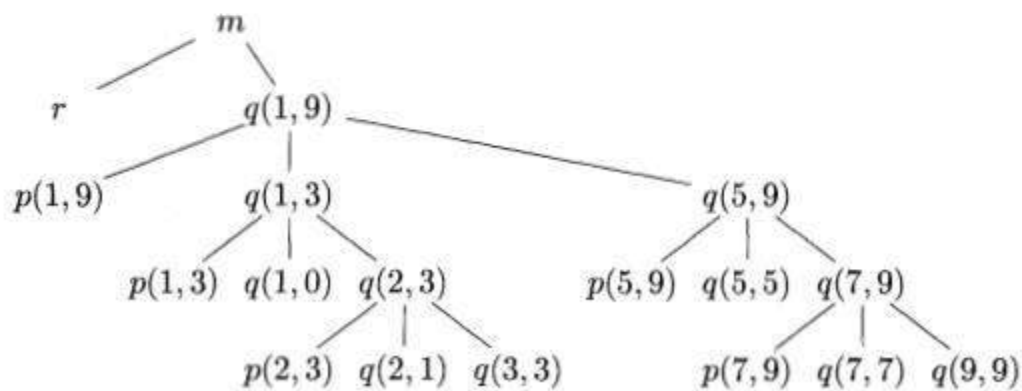Figure 7.3: Possible activations for the program of Fig. 7.2

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

2 Activation Records

    a. Procedure calls and returns are usually managed by a run-time stack called the control stack.

    b. Each live activation has an activation record (sometimes called a frame)

    c. The root of activation tree is at the bottom of the stack

    d. The current execution path specifies the content of the stack with the last

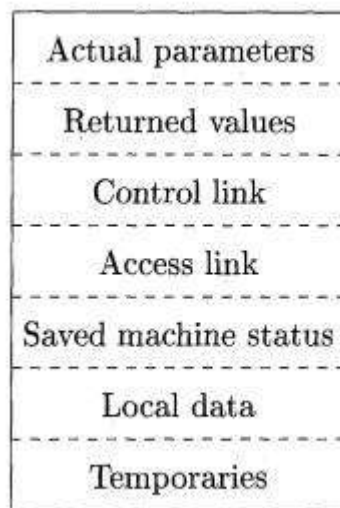   e. Activation has record in the top of the stack.

Figure 7.5: A general activation record

An activation record is used to store information about the status of the machine, such as the value of the program counter and machine registers, when a procedure call occurs. When control returns from the call, the activation of the calling procedure can be restarted after restoring the values of relevant registers and setting the program counter to the point immediately after the call. Data objects whose lifetimes are contained in that of an activation can be allocated on the stack along with other information associated with the activation.

An activation record contains all the necessary information required to call a procedure. An activation record may contain the following units (depending upon the source language used).

Temporaries	Stores temporary and intermediate values of an expression.
Local Data	Stores local data of the called procedure.
Machine Status	Stores machine status such as Registers, Program Counter etc., before the procedure is called.
Control Link	Stores the address of activation record of the caller procedure.
Access Link	Stores the information of data which is outside the local scope.
Actual Parameters	Stores actual parameters, i.e., parameters which are used to send input to the called procedure.

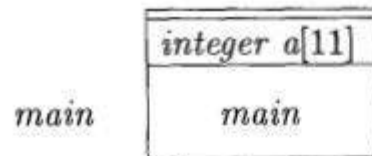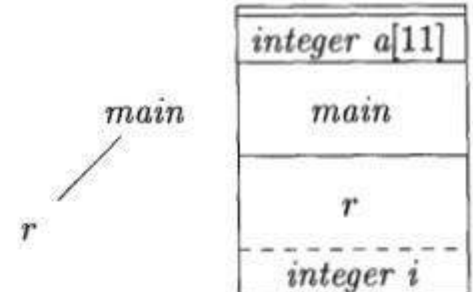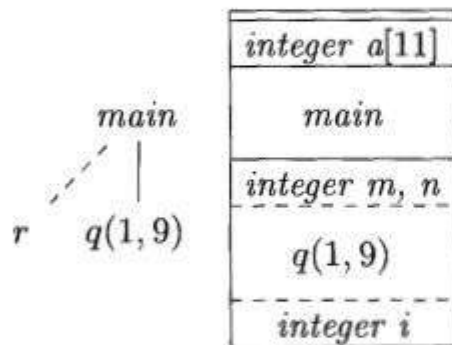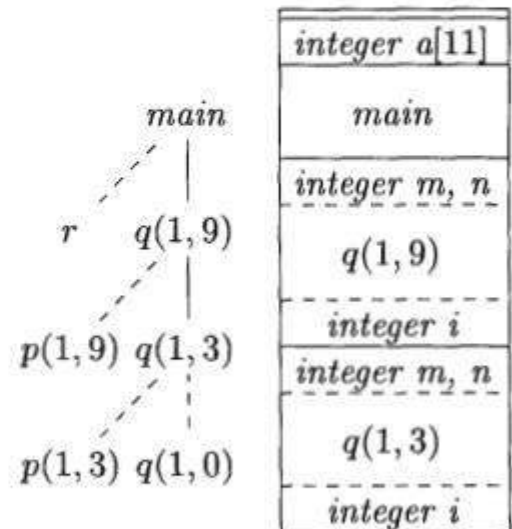| Return Value	Stores return values.

(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 Calling Sequences

Designing calling sequences and the layout of activation records, the following

1. Values communicated between caller and callee are generally placed at the beginning of callee's activation record
2. Fixed-length items: are generally placed at the middle. such items typically include the control link, the access link, and the machine status fields.
3. Items whose size may not be known early enough: are placed at the end of activation record
4. We must locate the top-of-stack pointer judiciously: a common approach is to have it point to the end of fixed length fields in the activation record.
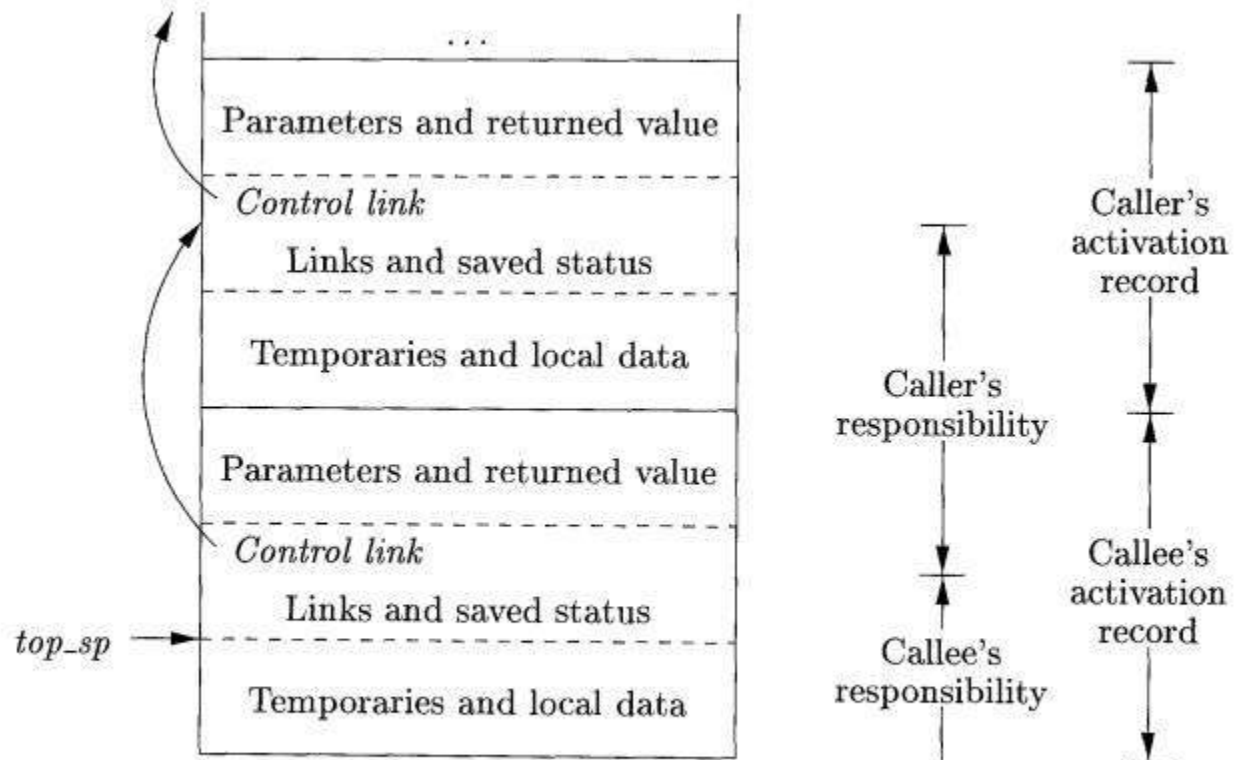
Figure 7.7: Division of tasks between caller and callee

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 made responsible for setting topsp before control is passed to the callee. The calling sequence and its division between caller and callee is as follows:

1. The caller evaluates the actual parameters.

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 the caller's local data and temporaries and the callee's parameters and status fields.

The callee saves the register values and other status information.

The callee initializes its local data and begins execution.

A suitable, corresponding **return sequence** is:

1. The callee places the return value next to the parameters, as in Fig. 7.5.

2. Using information in the machine-status field, the callee restores *topsp* and other registers,

   and then branches to the return address that the caller placed in the status field.

3. Although *topsp* has been decremented, the caller knows where the return value is, relative to
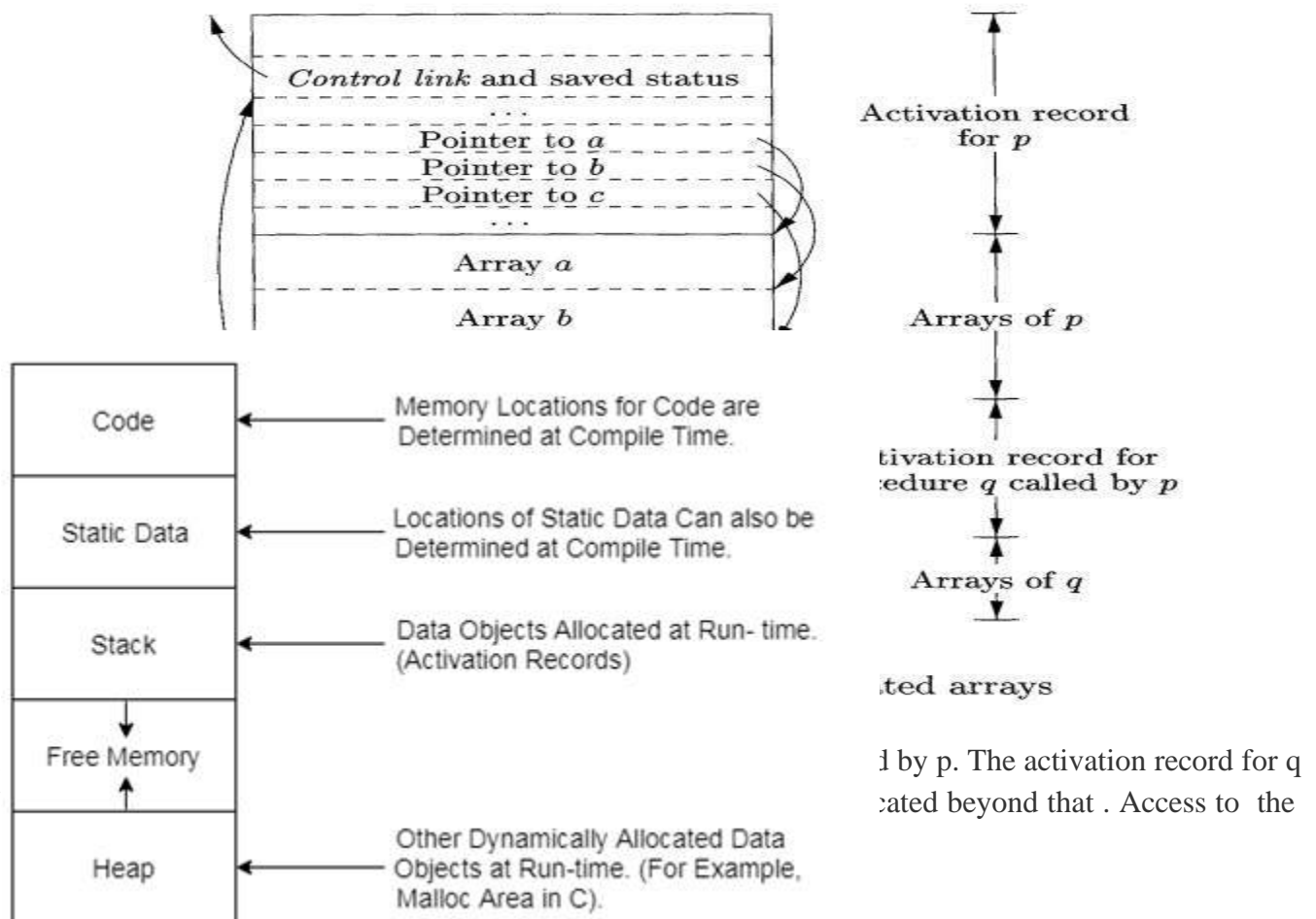
the current value of *topsp;* the caller therefore may use that value.

4. **Variable-Length Data on the Stack**

The run-time memory-management system must deal frequently with the allocation of space for objects the sizes of which are not known at compile time, but which are local to a procedure and thus may be allocated on the stack.

it is possible to allocate objects, arrays, or other structures of unknown size on the stack. The reason to prefer placing objects on the stack if possible is that we avoid the expense of garbage collecting their space. Note that the stack can be used only for an object if it is local to a procedure and becomes inaccessible when the procedure returns.

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

Control link and saved status
. . .
Pointer to *a*
Pointer to *b*
Pointer to *c*
. . .

Array *a*

Array *b*

Activation record
for *p*

Arrays of *p*

tivation record for
:edure *q* called by *p*

Arrays of *q*

Code — Memory Locations for Code are Determined at Compile Time.

Static Data — Locations of Static Data Can also be Determined at Compile Time.

Stack — Data Objects Allocated at Run- time. (Activation Records)

Free Memory

Heap — Other Dynamically Allocated Data Objects at Run-time. (For Example, Malloc Area in C).

.ted arrays

l by p. The activation record for q
:ated beyond that . Access to the

Access to Non Local data on the stack

Consider how procedures access their data. Especially im-portant is the mechanism for finding data used within a procedure p but that does not belong to p

1 Data Access Without Nested Procedures

Names are either local to the procedure in question or are declared globally.

1. For global names the address is known statically at compile time providing there is only one source file. If multiple source files, the linker knows. In either case no reference to the activation record is needed; the addresses are know prior to execution.
2. For names local to the current procedure, the address needed is in the AR at a known-at-compile-time constant offset from the sp. In the case of variable size arrays, the constant offset refers to a pointer to the actual storage.

 2 Issues With Nested Procedures

Access becomes far more complicated when a language allows procedure dec-larations to be nested .The reason is that knowing at compile time that the declaration of p is immediately nested within q does not tell us the relative positions of their activation records at run time. In fact, since either p or q or both may be recursive, there may be several activation records of p and/or q on the stack.

Finding the declaration that applies to a nonlocal name x in a nested pro-cedure p is a static decision; it can be done by an extension of the static-scope rule for blocks. Suppose x is declared in the enclosing procedure $q.$ Finding the relevant activation of q from an activation of p is a dynamic decision; it re-quires additional run-time information about activations. One possible solution is to use access links.

3. A Language With Nested Procedure Declarations

In various languages with nested procedures, one of the most influential is ML.

ML is a *functional language,* meaning that variables, once declared and initialized, are not changed. There are only a few exceptions, such as the array, whose elements can be changed by special function calls.

• Variables are defined, and have their unchangeable values initialized,

$$v \, a \, l \, (name) = (expression)$$

• Functions are defined using the syntax:

$$fun \, (name) \, (\, (arguments) \,) \; = (body)$$

• For function bodies, use let-statements of the form:

let (list of definitions) in (statements) end The definitions are normally v a l or fun statements. The scope of each such definition consists of all following definitions, up to the in, and all the statements up to the end. Most importantly, function definitions can be nested. For example, the body of a 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 of function

4. Nesting Depth

Nesting depth is 1 to procedures that are not nested within any other procedure. For example, all C functions are at nesting depth 1. However, if a procedure *p* is defined immediately within a procedure at 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. Access Links

A direct implementation of the normal static scope rule for nested functions is obtained by adding a pointer called the *access link* to each activation record. If procedure *p* is nested immediately within procedure *q* in the source code, then the access link in any activation of *p* points to the most recent activation of *q*. Note that the nesting depth of *q* must be exactly one less than the nesting depth of p. Access links form a chain from the activation record at the top of the stack to a sequence of activations at progressively lower nesting depths.

Figure 7.11 shows a sequence of stacks that might result from execution of the function *sort* of Fig. 7.10. In Fig. 7.11(a), we see the situation after *sort* has called *readArray* to load input into the array *a* and then called *quicksort(l,* 9) to sort the array. The access link from *quicksort(l,* 9) points to the activation record for *sort,* not because *sort* called *quicksort* but because *sort* is the most closely nested function surrounding *quicksort* in the program.
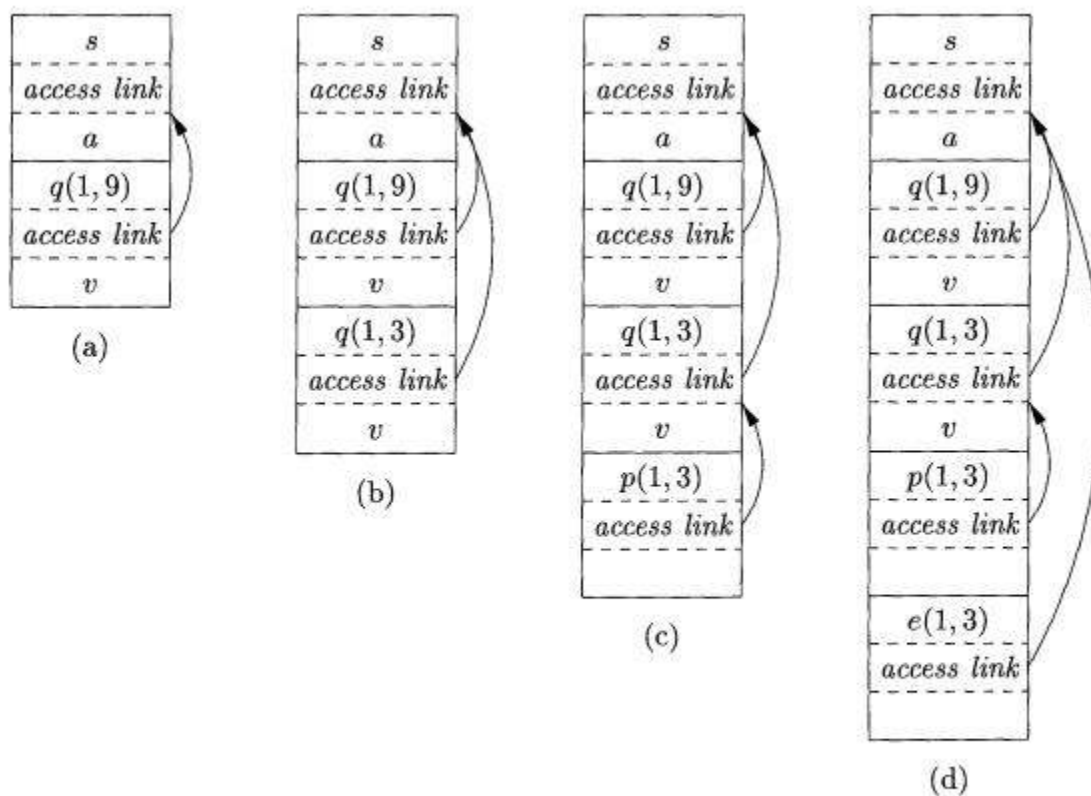

Figure 7.11: Access links for finding nonlocal data

see a recursive call to *quicksort(l,* 3), followed by a call to *partition,* which calls *exchange.* Notice that *quicksort(l,* 3)'s access link points to *sort,* for the same reason that *quicksort(l,* 9)'s does.

6. Manipulating Access Links

The harder case is when the call is to a procedure-parameter; in that case, the particular procedure being called is not known until run time, and the nesting depth of the called procedure may

differ in different executions of the call. consider situation when a procedure q calls procedure p, explicitly. There are three cases:

1. Procedure p is at a higher nesting depth than q. Then p must be defined immediately within q, or the call by q would not be at a position that is within the scope of the procedure name p. Thus, the nesting depth of p is exactly one greater than that of q, and the access link from p must lead to q. It is a simple matter for the calling sequence to include a step that places in the access link for p a pointer to the activation record of q.

2. The call is recursive, that is, $p = q$. Then the access link for the new activation record is the same as that of the activation record below it.

3. The nesting depth np of p is less than the nesting depth nq of q. In order for the call within q to be in the scope of name p, procedure q must be nested within some procedure r, while p is a procedure defined immediately within r. The top activation record for r can therefore be found by following the chain of access links, starting in the activation record for q, for nq — np + 1 hops. Then, the access link for p must go to this activation of r.

7. Access Links for Procedure Parameters

When a procedure p is passed to another procedure q as a parameter, and q then calls its parameter (and therefore calls p in this activation of q), it is possible that q does not know the context in which p appears in the program. If so, it is impossible for q to know how to set the access link for p. The solution to this is, when procedures are used as parameters, the caller needs to pass, along with the name of 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, and therefore, r can determine the access link for p exactly as if p were being called by r directly.
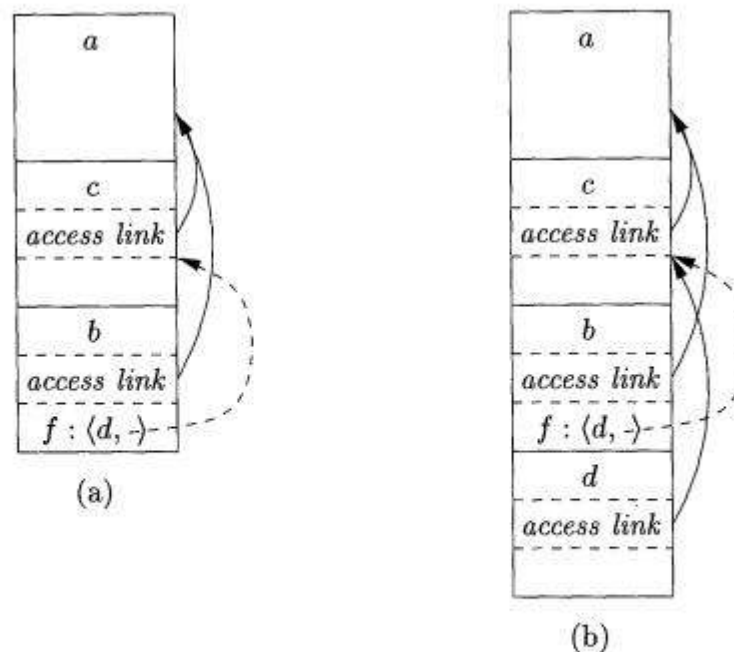

Figure 7.13: Actual parameters carry their access link with them

8. Displays

One problem with the access-link approach to nonlocal data is that if the nesting depth gets large, we may have to follow long chains of links to reach the data we need. A more efficient implementation uses an auxiliary array d, called the *display*, which consists of one pointer for each nesting depth. We arrange that, at all times, $d[i]$ is a pointer to the highest activation record on the stack for any procedure at nesting depth i. Examples of a display are shown in Fig. 7.14.
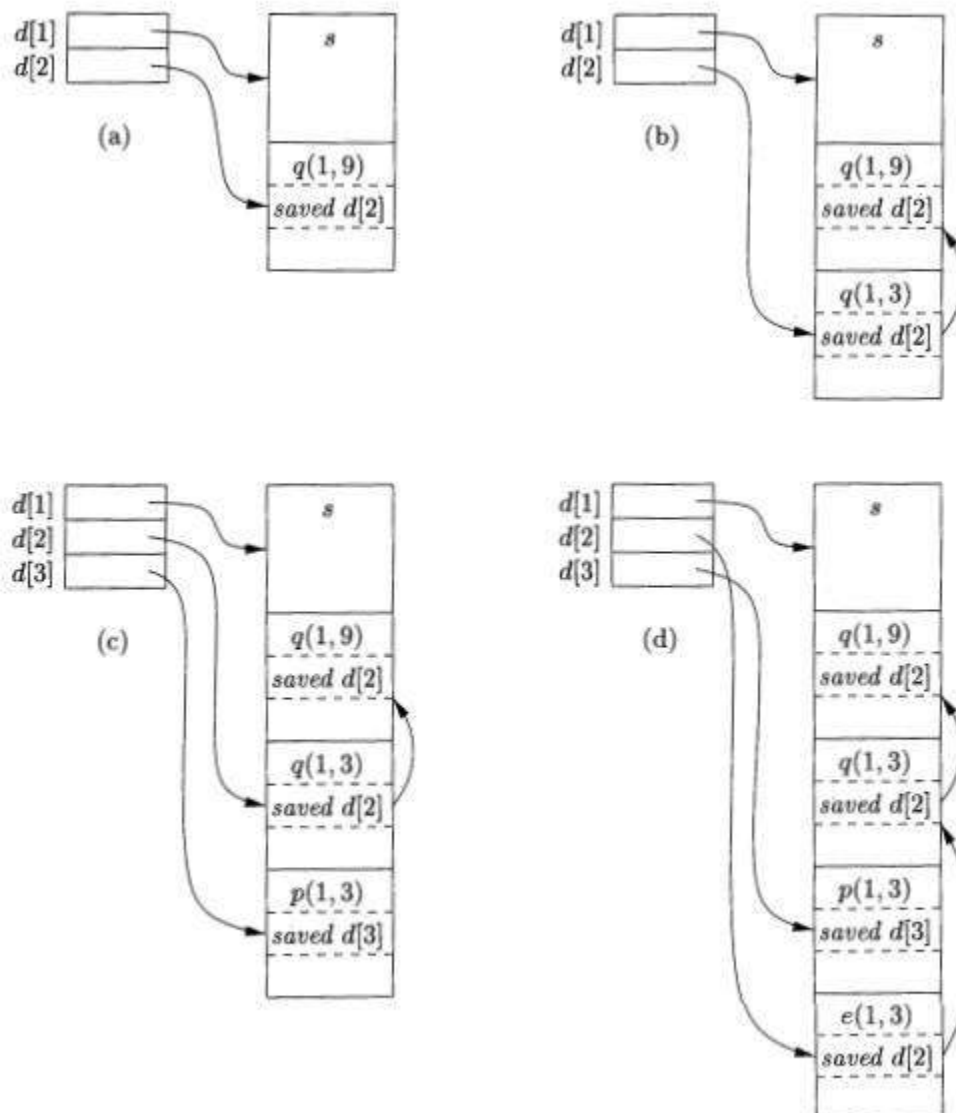

Figure 7.14: Maintaining the display

In order to maintain the display correctly, we need to save previous values of display entries in new activation records.

Heap Management

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

1 The Memory Manager

2 The Memory Hierarchy of a Computer

3 Locality in Programs

4 Reducing Fragmentation

5 Manual Deallocation Requests

1 The Memory Manager

It performs two basic functions:

• **Allocation**. When a program requests memory for a variable or object,[3] the memory manager produces a chunk of contiguous heap memory of the requested size. If possible, it satisfies an allocation request using free space in the heap; if no chunk of the needed size is available, it seeks to increase the heap storage space by getting consecutive bytes of virtual memory from the operating system. If space is exhausted, the memory manager passes that information back to the application program.

• **Deallocation**. The memory manager returns deallocated space to the pool of free space, so it can reuse the space to satisfy other allocation requests. Memory managers typically do not return memory to the operating sys-tem, even if the program's heap usage drops.

Thus, the memory manager must be prepared to service, in any order, allo-cation and deallocation requests of any size, ranging from one byte to as large as the program's entire address space.

Here are the properties we desire of memory managers:

• **Space Efficiency**. A memory manager should minimize the total heap space needed by a program. Larger programs to run in a fixed virtual address space..

• **Program Efficiency.** A memory manager should make good use of the memory subsystem to allow programs to run faster.

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

2. **The Memory Hierarchy of a Computer**

The efficiency of a program is determined not just by the number of instructions executed, but also by how long it takes to execute each of these instructions. The time taken to execute an instruction can vary significantly, since the time taken to access different parts of memory can vary from

nanoseconds to milliseconds. Data-intensive programs can therefore benefit significantly from optimizations that make good use of the memory subsystem.
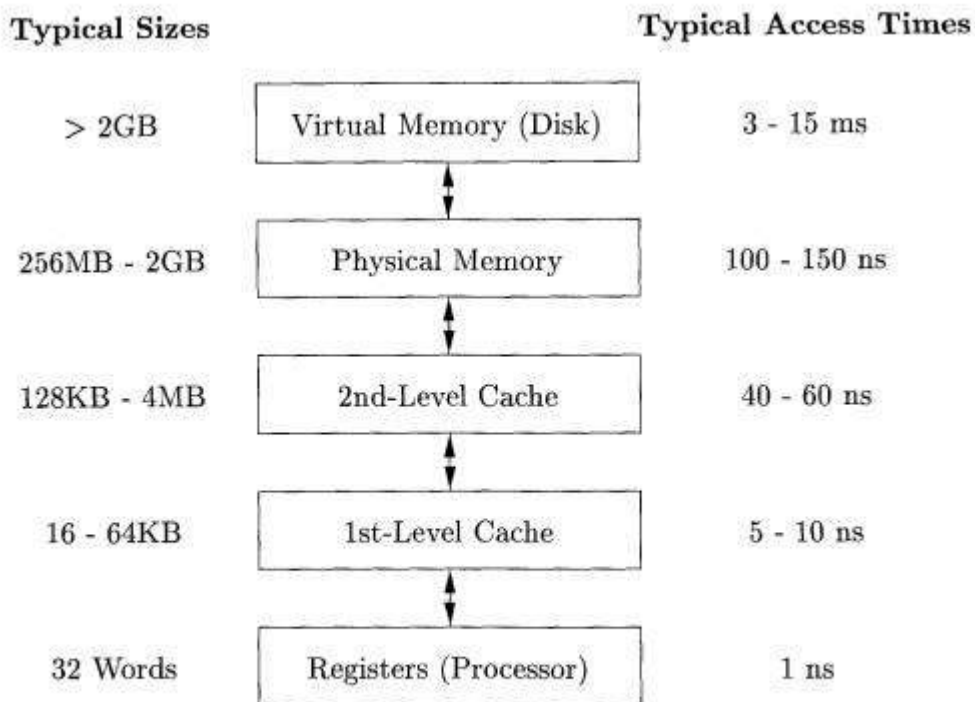
Typical Sizes   **Typical Access Times**

> 2GB	Virtual Memory (Disk)	3 - 15 ms
256MB - 2GB	Physical Memory	100 - 150 ns
128KB - 4MB	2nd-Level Cache	40 - 60 ns
16 - 64KB	1st-Level Cache	5 - 10 ns
32 Words	Registers (Processor)	1 ns

Figure 7.16: Typical Memory Hierarchy Configurations

3. Locality in Programs

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

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

The typical program spends most of its time executing innermost loops and tight recursive cycles 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 be lowered significantly.

4. Reducing Fragmentation

To begin with the whole heap is a single chunk of size 500K bytes

After a few allocations and deallocations, there are holes

In the above picture, it is not possible to allocate 100K or 150K even though total free memory is 150K

With each deallocation request, the freed chunks of memory are added back to the pool of free space. We coalesce contiguous holes into larger holes, as the holes can only get smaller otherwise. If we 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, even though there may be sufficient aggregate free space.

Best - Fit and Next - Fit Object Placement

We reduce fragmentation by controlling how the memory manager places new objects in the heap. It has been found empirically that a good strategy for minimizing fragmentation for real life programs is to allocate the requested memory in the smallest available hole that is large enough. This *best-fit* algorithm tends to spare the large holes to satisfy subsequent, larger requests. An alternative, called *first-fit,* where an object is placed in the first (lowest-address) hole in which it fits, takes less time to place objects, but has been found inferior to best-fit in overall performance.

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

M a n a g i n g and Coalescing Free Space

When an object is deallocated manually, the memory manager must make its chunk free, so it can be allocated again. In some circumstances, it may also be possible to combine *(coalesce)* that chunk with adjacent chunks of the heap, to form a larger chunk. There is an advantage to doing so, since we can always use a large chunk to do the work of small chunks of equal total size, but many small chunks cannot hold one large object, as the combined chunk could.

Automatic garbage collection can eliminate fragmentation altogether if it moves all the allocated objects to contiguous storage.

5. Manual Deallocation Requests

In manual memory management, where the programmer must explicitly arrange for the deallocation of data, as in C and C + + . Ideally, any storage that will no longer be accessed should be deleted.

Problems with Manual Deallocation

1.Memory leaks ‰

Failing to delete data that cannot be referenced ‰

Important in long running or nonstop programs ,,

2.Dangling pointer dereferencing ‰

Referencing deleted data ,,

Both are serious and hard to debug

Garbage Collection ,,

1. Reclamation of chunks of storage holding objects that can no longer be accessed by a program ,,

2. GC should be able to determine types of objects ‰

Then, size and pointer fields of objects can be determined by the GC ‰

Languages in which types of objects can be determined at compile time or run-time are type safe ,,

Java is type safe ,,

C and C++ are not type safe because they permit type casting, which creates new pointers

,,

Thus, any memory location can be (theoretically) accessed at any time and hence cannot be considered inaccessible

Code Generation

It takes as input the intermediate representation (IR) produced by the front end of the compiler, along with relevant symbol table information, and produces as output a semantically equivalent target program
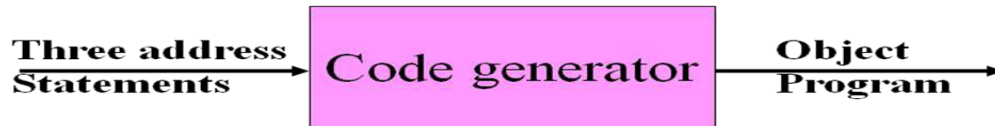
The most important criterion for a code generator is that it produce correct code.

The following issue arises during the code generation phase:

> 1 Input to the Code Generator
> 2 The Target Program
> 3 Instruction Selection
> 4 Register Allocation
> 5.Evaluation Order

Input to code generator

The input to code generator is the intermediate code generated by the front end, along with information in the symbol table that determines the run-time addresses of the data-objects denoted by the names in the intermediate representation. Intermediate codes may be represented mostly in quadruples, triples, indirect triples, Postfix notation, syntax trees, DAG's, etc. The code generation phase just proceeds on an assumption that the input are free from all of syntactic and state semantic errors, the necessary type checking has taken place and the type-conversion operators have been inserted wherever necessary

Target program
The target program is the output of the code generator. The output may be absolute machine language, relocatable machine language, assembly language.

- **Absolute** machine language as output has advantages that it can be placed in a fixed memory location and can be immediately executed.
- **Relocatable** machine language as an output allows subprograms and subroutines to be compiled separately. Relocatable object modules can be linked together and loaded by linking loader. But there is added expense of linking and loading.
- **Assembly** language as output makes the code generation easier. We can generate symbolic instructions and use macro-facilities of assembler in generating code. And we need an additional assembly step after code generation.

.

Instruction selection

Selecting the best instructions will improve the efficiency of the program. It includes the instructions that should be complete and uniform. Instruction speeds and machine idioms also plays a major role when efficiency is considered. But if we do not care about the efficiency of the target program then instruction selection is straight-forward.

For example, three-address statements would be translated into the latter code sequence as shown below:

P:=Q+R

S:=P+T

MOV Q, R0

ADD R, R0

MOV R0, P

MOV P, R0

ADD T, R0

MOV R0, S

Here the fourth statement is redundant as the value of the P is loaded again in that statement that just has been stored in the previous statement. It leads to an inefficient code sequence. A given intermediate representation can be translated into many code sequences, with significant cost differences between the different implementations. A prior knowledge of instruction cost is needed in order to design good sequences, but accurate cost information is difficult to predict.

Register allocation issues

Use of registers make the computations faster in comparison to that of memory, so efficient utilization of registers is important. The use of registers are subdivided into two subproblems:

1.During **Register allocation** – we select only those set of variables that will reside in the registers at each point in the program.
2.During a subsequent **Register assignment** phase, the specific register is picked to access the variable.

As the number of variables increases, the optimal assignment of registers to variables becomes difficult. Mathematically, this problem becomes NP-complete. Certain machine requires register pairs consist of an even and next odd-numbered register. For example

M a, b

These types of multiplicative instruction involve register pairs where the multiplicand is an even register and b, the multiplier is the odd register of the even/odd register pair.

Evaluation order –
The code generator decides the order in which the instruction will be executed. The order of computations affects the efficiency of the target code. Among many computational orders, some will require only fewer registers to hold the intermediate results. However, picking the best order in the general case is a difficult NP-complete problem.

Approaches to code generation issues:
Code generator must always generate the correct code. It is   essential because of the number of special cases that a code generator might face. Some of the design goals of code generator are:

- Correct
- Easily maintainable
- Testable
- Efficient

The target Language

1 A Simple Target Machine Model

2 Program and Instruction Costs

A Simple Target Machine Model

op source, destination

Where, op is used as an op-code and source and destination are used as a data field.

- o It has the following op-codes:
  - ADD (add source to destination)
  - SUB (subtract source from destination)
  - MOV (move source to destination)
- o The source and destination of an instruction can be specified by the combination of registers and memory location with address modes.

MODE	FORM	ADDRESS	EXAMPLE	ADDED COST
Absolute	M	M	Add R0, R1	1
Register	R	R	Add temp, R1	0
indexed	c(R)	C+ contents(R)	ADD 100 (R2), R1	1
indirect register	*R	contents(R)	ADD * 100	0
indirect indexed	*c(R)	contents(c+ contents(R))	(R2), R1	1
literal	#c	c	ADD #3, R1	1

- o Here, cost 1 means that it occupies only one word of memory.
- o Each instruction has a cost of 1 plus added costs for the source and destination.
- o Instruction cost = 1 + cost is used for source and destination mode.

2 Program and Instruction Costs

Cost of an instruction to be one plus the costs associated with the addressing modes of the operands . This cost corresponds to the length in words of the instruction. Addressing modes involving registers have zero additional cost, while those involving a memory location or constant in them have an additional cost of one, because such operands have to be stored in the words following the instruction.

Examples:

• The instruction LD RO, Rl copies the contents of register Rl into register RO. This instruction has a cost of one because no additional memory words are required.

• The instruction LD RO, M loads the contents of memory location M into register RO. T h e cost is two since the address of memory location M is in the word following the instruction.

• The instruction LD R 1 , *100(R2) loads into register Rl the value given by contents(contents(100 + contents(K2))). The cost is three because the constant 100 is stored in the word following the instruction.

Example:
1. Move register to memory R0 $\rightarrow$ M

    MOV R0, M

    cost = 1+1+1   (since address of memory location M is in word following the instruction)

2. Indirect indexed mode: MOV * 4(R0), M

    cost = 1+1+1   (since one word **for** memory location M, one word
    result of *4(R0) and one **for** instruction)

3. Literal Mode:

    MOV #1, R0
    cost = 1+1+1 = 3   (one word **for** constant 1 and one **for** instruction)

Address in the target code

The information which required during an execution of a procedure is kept in a block of storage called an activation record. The activation record includes storage for names local to the procedure. We can describe address in the target code using the following ways:

1. Static allocation

2. Stack allocation

In static allocation, the position of an activation record is fixed in memory at compile time.

.
In the stack allocation, for each execution of a procedure a new activation record is pushed onto the stack. When the activation ends then the record is popped.

For the run-time allocation and deallocation of activation records the following three-address statements are associated:

1. Call

2. Return

3. Halt

4. Action, a placeholder for other statements

Assume that the run-time memory is divided into areas for:

1. Code

2. Static data

3. Stack

Static allocation:

1. Implementation of call statement:

The following code is needed to implement static allocation:

MOV #here + 20, callee.static_area   /*it saves return address*/</p>

GOTO callee.code_area   /* It transfers control to the target code for the called procedure*/

Where,

callee.static_area shows the address of the activation record.

callee.code_area shows the address of the first instruction for called procedure.

#here + 20 literal are used to return address of the instruction following GOTO.

2. Implementation of return statement:
The following code is needed to implement return from procedure callee:

GOTO * callee.static_area

It is used to transfer the control to the address that is saved at the beginning of the activation record.

3. Implementation of action statement:

The ACTION instruction is used to implement action statement.

4. Implementation of halt statement:

The HALT statement is the final instruction that is used to return the control to the operating system.

Stack allocation

Using the relative address, static allocation can become stack allocation for storage in activation records.

In stack allocation, register is used to store the position of activation record so words in activation records can be accessed as offsets from the value in this register.

The following code is needed to implement stack allocation:

1. Initialization of stack:

```
MOV #stackstart , SP   /*initializes stack*/
 HALT                      /*terminate execution*/
```

2. Implementation of Call statement:

```
        ADD #caller.recordsize, SP/* increment stack pointer */

        MOV #here + 16, *SP              /*Save return address */

        GOTO callee.code_area
```

Where,

caller.recordsize is the size of the activation record

#here + 16 is the address of the instruction following the **GOTO**

3. Implementation of Return statement:

```
        GOTO *0 ( SP ) /*return to the caller */
         SUB #caller.recordsize, SP          /*decrement SP and restore to previous value */
```

Basic blocks and Flow graphs

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. Flow graph of a program can be used as a vehicle to collect information about the intermediate program. Some register-assignment algorithms use flow graphs to find the inner loops where a program is expected to spend most of its time.

Basic block contains a sequence of statement. The flow of control enters at the beginning of the statement and leave at the end without any halt (except may be the last instruction of the block).

The following sequence of three address statements forms a basic block:

1. t1:= x * x
2. t2:= x * y
3. t3:= 2 * t2
4. t4:= t1 + t3
5. t5:= y * y
6. t6:= t4 + t5

Basic block construction:

Algorithm: Partition into basic blocks

Input: It contains the sequence of three address statements

Output: it contains a list of basic blocks with each three address statement in exactly one block

Method: First identify the leader in the code. The rules for finding leaders are as follows:

- o   The first statement is a leader.
- o   Statement L is a leader if there is an conditional or unconditional goto statement like: if....goto L or goto L
- o   Instruction L is a leader if it immediately follows a goto or conditional goto statement like: if goto B or goto B

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

Consider the following source code for dot product of two vectors a and b of length 10:

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

The three address code for the above source program is given below:

B1

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

(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)   if i<=10 goto (3)

Basic block B1 contains the statement (1) to (2)
Basic block B2 contains the statement (3) to (12)

Flow Graph

Flow graph is a directed graph. It contains the flow of control information for the set of basic block.

A control flow graph is used to depict that how the program control is being parsed among the blocks. It is useful in the loop optimization.Flow graph for the vector dot product is given as follows:
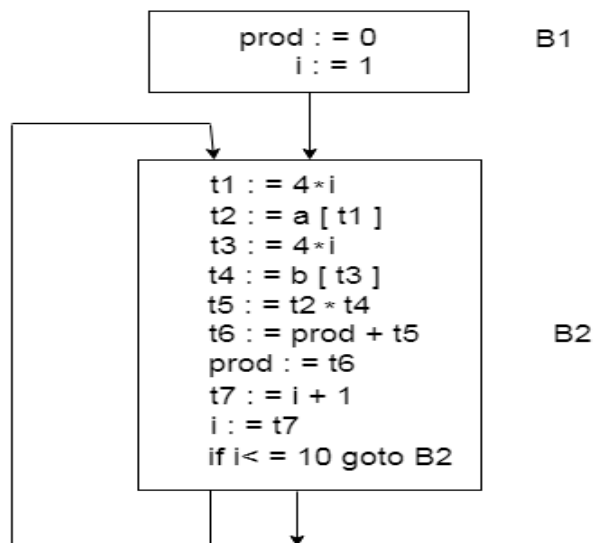
1.Block B1 is the initial node. Block B2 immediately follows B1, so from B2 to B1 there is an edge.

2.The target of jump from last statement of B1 is the first statement B2, so from B1 to B2 there is an edge.

A Simple Code generation.

Code generator is used to produce the target code for three-address statements. It uses registers to store the operands of the three address statement.

Consider the three address statement x:= y + z. It can have the following sequence of codes:

MOV x, R_0

ADD y, R_0

Register and Address Descriptors:

o A register descriptor contains the track of what is currently in each register. The register descriptors show that all the registers are initially empty.

o An address descriptor is used to store the location where current value of the name can be found at run time.

A code-generation algorithm:

The algorithm takes a sequence of three-address statements as input. For each three address statement of the form a:= b op c perform the various actions. These are as follows:

1. Invoke a function getreg to find out the location L where the result of computation b op c should be stored.

2. Consult the address description for y to determine y'. If the value of y currently in memory and register both then prefer the register y' . If the value of y is not already in L then generate the instruction **MOV y' , L** to place a copy of y in L.

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

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

Generating Code for Assignment Statements:

The assignment statement d:= (a-b) + (a-c) + (a-c) can be translated into the following sequence of three address code:

t:= a-b

u:= a-c

v:= t +u

d:= v+u

Code sequence for the example is as follows:

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