Module 5

Run-Time Environments & Intermediate Code Generation

RUN-TIME ENVIRONMENTS

Ques 1) What is run-time environment? What are the entities which are managed at runtime?

Ans: Run-Time Environment

Compiler must cooperate with OS and other system software to support implementation of different abstractions (names, scopes, bindings, data types, operators, procedures, parameters, flow-of-control) on the target machine.

Compiler does this by Run-Time Environment in which it assumes its target programs are being executed.

Run-Time Environment deals with:

1) Layout and allocation of storage
2) Access to variable and data
3) Linkage between procedures
4) Parameters passing
5) Interface to OS, I/O devices etc.

Run-time environment refers to the program snap-shot during execution. A program consists of three main segments:

1) Code for the program,
2) Static and global variables,
3) Local variables and arguments.

Each of the above three segments need proper allocation of memory to hold their values. Thus, three kinds of entities are to be managed at the run-time:

1) **Generated Code:** For various procedures and programs that form text or code segment. The size of this segment is known at compile time. Thus, the space can be allocated statically before the execution commences.
2) **Data Objects:** They can be of different types:
   i) **Global Variables/Constants:** The size of the total space required by global variables or constants is known at compile time.
   ii) **Local Variables:** For local variables also, the size is known at compile time.
   iii) **Variables Created Dynamically:** These variables correspond to the space created in response to the memory allocation requests from the program during execution. Since it depends on the execution sequence of the program, the size is unknown at compile time. The dynamic allocation is done in heap.
3) **Stack:** To keep track of procedure activations.

Ques 2) What are the source language issues?

Or

What is activation tree?

Ans: Source Language Issues

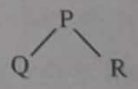
The main source language issues are as follows:

1) **Procedures:** A procedure definition is a declaration that associates an identifier with a statement. The identifier is the procedure name, and the statement is the procedure body. For example, the following is the definition of procedure named readarray:

```
procedure readarray; var i : integer;
begin
    for i : = 1 to 9 do read(a[i])
end;
```

When a procedure name appears within an executable statement, the procedure is said to be called at that point.

2) **Activation Tree:** The flow of control between activations can be depicted by a tree called an **activation tree**. The nodes in the tree represent activations. When activation P calls activation Q, the node for P has the node for Q as a child. If P calls Q before it calls R, then the node for Q appears to the left of the node for R:



A procedure is recursive if a new activation can begin before an earlier activation of the same procedure has ended. A recursive procedure need not call itself directly; p may call another procedure q, which may then call p through some sequence of procedure calls.

One can use tree, called an **activation tree**, to depict the way control enters and leaves activations. In an activation tree:

i) Each node represents an activation of a procedure,

ii) Root represents the activation of the main program,
iii) Node for a is the parent of the node for b if and only if the control flows from activation a to b, and
iv) Node for a, is to the left of the node for b if and only if the lifetime of a, occurs before the lifetime of b.

For example, consider a variant of the factorial function, which computes n – 1 by calling a function pred(n), as in the following program:

```
program activations;
    function pred(m: integer): integer;
    begin
            pred := m – 1
            end;
            function f(n: integer): integer;
            begin
            if n = 0 then f := 1 else f := n*f(pred(n))
    end;
    begin
    f(3)
    end.
```

The activation tree in **figure 5.1** illustrates the activations that occur during an execution of this program:

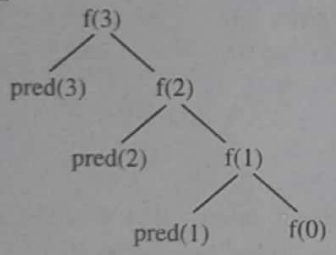


Figure 5.1: Activation Tree for the Program

3) **Control Stack:** A control stack is used to keep track of live procedure activations. The idea is to push the node for activation onto the control stack as the activation begins and to pop the node when the activation ends. The contents of the control stack are related to paths to the root of the activation tree. When node n is at the top of control stack, the stack contains the nodes along the path from n to the root.

4) **Scope of a Declaration:** A declaration is a syntactic construct that associates information with a n Declarations may be explicit, such as:

```
var i : integer ;
```

Or they may be implicit. Example, any variable name starting with I is assumed to denote an integer. The portion of the program to which a declaration applies is called the **scope of that declaration**.

5) **Binding of Names:** Even if each name is declared once in a program, the same name may denote different data objects at run time. "Data object" corresponds to a storage location that holds values. The term environment refers to a function that maps a name to a storage location. The term 'state' refers to a function that maps a storage location to the value held there. When an environment associates storage location s with a name x, we say that x is bound to s. This association is referred to as a binding of x.

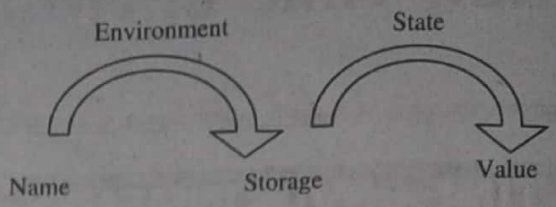


Figure 5.2: Two-Stage Mapping from Names to Values

Ques 3) Discuss about the storage organisation.

Ans: Storage Organisations

In run-time environment, the executing target program runs in its own logical address space in which each program value has a location. The management and organisation of this logical address is shared between the compiler, operating system, and target machine. The OS maps the logical address into physical addresses, which are usually spread throughout memory.

Compiler should consequently perform the following steps:

1) Allocate memory for a variable
2) Initialize allocated raw memory with a default value
3) Allow a programmer accessing this memory
4) Clean (if necessary) and free memory once the corresponding resource is no longer used
5) Finally, the freed memory should be utilized and marked as ready for further reuse.

The run-time storage might be divided into following types (**figure 5.3**):

1) Code Segment
2) Data Segment (Holds Global Data)
3) Stack where the local variables and other temporary information is stored.
4) Heap

This kind of organisation of run time storage is used for language such as FORTRAN, PASCAL and C.

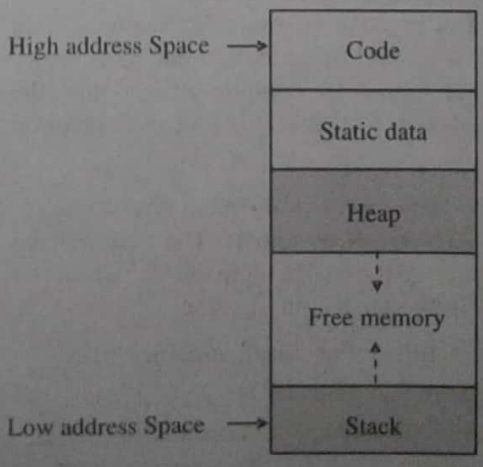


Figure 5.3: Subdivision of Run-Time Memory

The size of the generated target is fixed at compile time, so the compiler can place the executable target code in a statistically determined area "code" usually in the low end of memory.

Similarly, the size of some program data objects, such as global constants and data generated by the compiler, such as information to support garbage collection, may be known at compile time and these data objects can be placed in another statistically determined area called **static data**.

To maximize the utilisation of space at run time, the other 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. These areas grow towards each other as needed.

Ques 4) What are the different storage allocation strategies?

Or

What is static, stack and heap allocation?

Ans: Storage Allocation Strategies

There are two types of storage allocation strategies. These are:

1) **Static Storage Allocation:** If the size of every data item can be determined by the compiler and if recursion procedure calls are not permitted, then the space for all programs and data can be allocated at compile time, i.e., statically. Moreover, the association of names to location can also be done statically in this case.

   In **static allocation**, names are bound to storage as the program is compiled, so there is no need for a run-time support package. Since the bindings do not change at run time, every time a procedure is activated, its names are bounded to the same storage locations. This property allows the values of the local names to be retained across **activations** of a procedure. That is, when control returns to a procedure, the values of the locals are the same as they are when control left the last time.

   Static allocation has the virtues that it is easy to implement efficiently and requires no rub (means message) – time support for allocation.

   Static memory management is performed by the compiler before execution starts. Statically allocated memory objects reside in a fixed zone of memory (which is determined by the compiler) and they remain there for the entire duration of the program's execution. Typical elements for which it is possible statically to allocate memory are global variables. These indeed can be stored in a memory area that is fixed before execution begins because they are visible throughout the program. The object code instructions produced by the compiler can be considered another kind of static object, given that normally they do not change during the execution of the program, so in this case also memory will be allocated by the compiler.

   Constants are other elements that can be handled statically (in the case in which their values do not depend on other values which are unknown at compile time).

   Finally, various compiler-generated tables, necessary for the run-time support of the language (e.g., for handling names, for type checking, for garbage collection) are stored in reserved areas allocated by the compiler.

   The situation of a language with only static memory allocation is shown in **figure 5.4**:

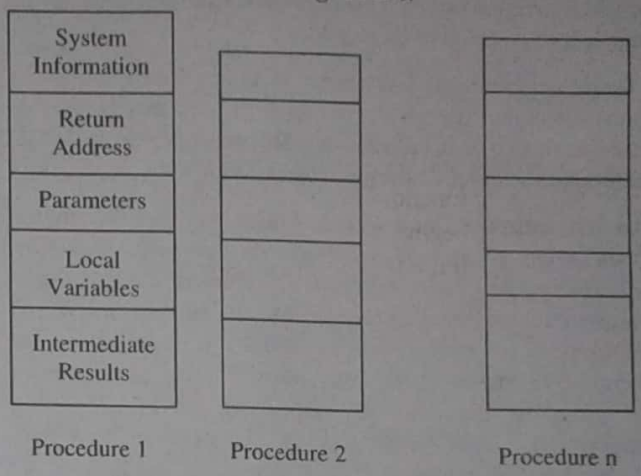


Figure 5.4: Static Memory Management

2) **Dynamic Storage Allocation:** If the programming language permits either recursive procedures or data structure whose size is adjustable, then some sort of dynamic storage management is necessary.

   Types of Dynamic Storage Allocation

   There are two kinds of dynamic storage allocation:

   i) **Stack Storage Allocation:** Storage is organised as a stack and activation records are pushed and popped as activation begin and end respectively. Locals are contained in activation records so they are bound to fresh storage in each activation. Recursion is supported in stack allocation

   ii) **Heap Storage Allocation:** Memory allocation and deallocation can be done at any time and at any place depending on the requirement of the user. Heap allocation is used to dynamically allocate memory to the variables and claim it back when the variables are no more required. Recursion is supported.

Ques 5) What are the limitations of static storage allocation?

Ans: Limitations of Static Storage Allocation

1) The size of a data object and constraints on its position in memory must be known at compile time.
2) Recursive procedures are restricted, because all activations of a procedure use the same bindings for local names.
3) Data structures cannot be created dynamically, since there is no mechanism for storage allocation at run time.

Ques 6) Discuss about the allocation and management of stack. What are the limitations of stack based memory allocation.

Ans: Allocation and Management of Stack

In a stack, allocations and deallocations are performed in a Last-In-First-Out (LIFO) manner in response to push and pop operations, respectively. We assume the each entry in the stack to be of some standard size, say, l bytes. Only the last entry of stack is accessible at any time. A contiguous area of memory is reserved for the stack. A pointer called **Stack Base (SB)** points to the first entry of the stack, while a pointer called **Top of Stack (TOS)** points to the last entry allocated in the stack.

During execution of a program, a stack is used to support function calls. The group of stack entries that pertain to one function call is called **stack frame**; it is also called an **activation record** in compiler terminology.

An **activation record (stack frame)** is pushed on the stack when a function is called. To start with, the stack frame contains either addresses or values of the function's parameters, and the return address, i.e., the address of the instruction to which control should be returned after completing the function's execution. During execution of the function, the runtime support of the programming language in which the program is coded creates local data of the function within the stack frame. At the end of the function's execution, the entire stack frame is popped-off the stack and the return address is used to pass control back to the calling program.

Two provisions are made to facilitate use of stack frames:

1) The first entry in a stack frame is a pointer to the previous stack frame on the stack. This entry facilitates popping-off of a stack frame.
2) An additional pointer called the Frame Base (FB) is used to point to the start of the topmost stack frame in the stack. It helps in accessing various stack entries in the stack frame.

Limitations of Stack based Memory Allocation

1) A limitation of stack based memory allocation is that a thread's stack size can be as small as a few dozen kilobytes. Allocating more memory on the stack than is available can result in a crash due to stack overflow.
2) Another disadvantage is that the memory stored on the stack is automatically deallocated when the function that created it returns, and thus the function must copy the data if they should be available to other parts of the program after it returns.

Ques 7) What is activation record in stack storage allocation?

Ans: Activation Record

The segment of the stack that contains the variables for a function is the "**activation record**" or "**stack frame**" of that function. In addition to storing local variables, the activation record for a function also stores some saved register values.

A special register, called the **Frame Pointer (FP)**, points to the beginning of the current frame. Another register, the "Stack Pointer" (SP), contains the address of the first unused location on the stack. SimpleJava stacks will "grow" from large addresses to small addresses. Thus, when we push items onto the stack, we will subtract from the stack pointer, and when we pop items-off the stack, we will add to the stack pointer. In the **stack frame** for a function, local variables will appear before saved registers. The return value for a function will not be passed on the stack, but instead will be stored in a special result register. Thus, when a function is called, the values of the input parameters are placed on the stack, and when a function returns a value, that value is stored in the result register.

For example, the simple Java function:

```
int foo() {
        int a;
        int b;
        /* body of foo */
}
```

would have the activation record:

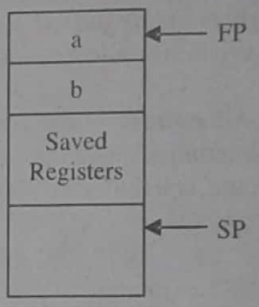


Figure 5.5

Since the frame pointer always points to the beginning of the current activation record, one can access local variables by using their offset from the frame pointer.

Function parameters are stored in the activation record of the calling function, not the called function. Input parameters to a function can also be accessed through the frame pointer, using an offset in the opposite direction from that of local variables. The complete format for activation records is in **figure 5.6**:

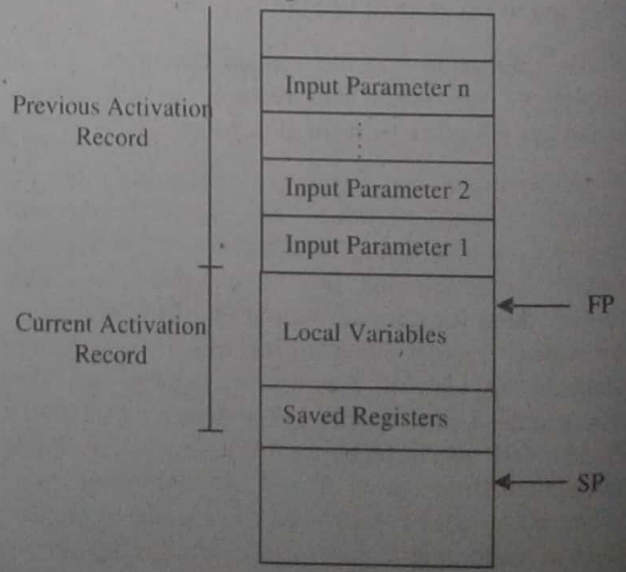


Figure 5.6: Format of Activation Records for SimpleJava

Ques 8) Discuss the activation records for procedures.

Ans: Activation Records for Procedures

The case of procedures and functions is analogous to that of in-line blocks but with some additional complications due to the fact that, when a procedure is activated, it is necessary to store a greater amount of information to manage correctly the control flow. The structure of a generic activation record for a procedure is shown in **figure 5.7**:

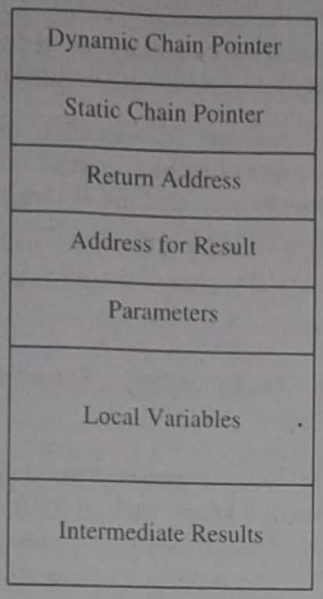


Figure 5.7: Structure of the Activation Record for a Procedure

The various fields of an activation record are as follows:

1) **Intermediate Results, Local Variables, Dynamic Chain Pointer:** The same as for in-line blocks.
2) **Static Chain Pointer:** This stores the information needed to implement the static scope rules. =
3) **Return Address:** Contains the address of the first instruction to execute after the call to the current procedure/function has terminated execution.
4) **Returned Result:** Present only in functions. Contains the address of the memory location where the subprogram stores the value to be returned by the function when it terminates. This memory location is inside the caller's activation record.
5) **Parameters:** The values of actual parameters used to call the procedure or function are stored here.

For example, blocks, whether in-line or associated with procedures, are entered and left using the LIFO scheme. When a block A is entered, and then a block B is entered, before leaving A, it is necessary to leave B. It is therefore natural to manage the memory space required to store the information local to each block using a stack.

For example, Let consider the following program:

```
A: {
        int a = 1;
        int b = 0;
        B: {
                int c = 3;
                int b = 3;
           }
        b = a + 1;
}
```

At run-time, when block A is entered, a push operation allocates a space large enough to hold the variables a and b, as shown in **figure 5.8.**

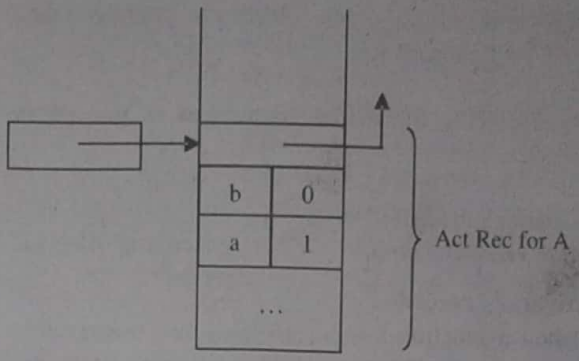


Figure 5.8: Allocation of an Activation Record for Block A

When block B is entered, one have to allocate a new space on the stack for the variables c and b (recall that the inner variable b is different from the outer one) and therefore the situation, after this second allocation, is that shown in **figure 5.9**.

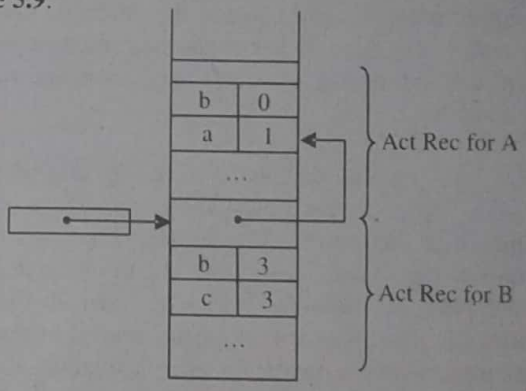


Figure 5.9: Allocation of Activation Records for Blocks A and B

When block B exits, on the other hand, it is necessary to perform a pop operation to deallocate the space that had been reserved for the block from the stack. The situation after such a deallocation and after the assignment is shown in **figure 5.10**.

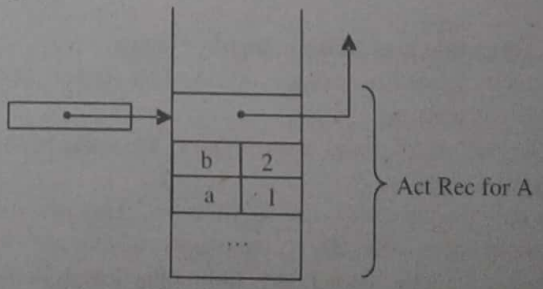


Figure 5.10: Organisation after the Execution of the Assignment

Analogously, when block A exits, it will be necessary to perform another pop to deallocate the space for A as well.

The memory space, allocated on the stack, dedicated to an in-line block or to an activation of a procedure is called the activation record, or frame. Note that an activation record is associated with a specific activation of a procedure (one is created when the procedure is called) and not with the declaration of a procedure. The values that must be stored in an activation record (local variables, temporary variables, etc.) are indeed different for the different calls on the same procedure.

The stack on which activation records are stored is called **run-time** (or system) **stack**.

Ques 9) Describe the calling sequences in procedure calls.

Ans: Calling Sequences
Procedure calls are implemented by generating what are known as **calling sequences** in the target code.

A **call sequence** allocates an activation record and enters information into its fields. A **return sequence** restores the state of the machine so the calling procedure can continue execution.

Calling sequences and activation records differ, even for implementations of the same language. The code in a calling sequence is often divided between the calling procedure and the procedure it calls. There is no exact division of run-time tasks between the caller and the callee –the source language, the target machine and the operating system impose requirements that may favor one solution over another.

A principle that aids the design of calling sequences and activation records is that fields whose sizes are fixed early are placed in the middle. The decision about whether or not to use control and access links is part of the design of the compiler, so these fields can be fixed at compiler-construction time. If exactly the same amount of machine-status information is saved for each activation, then the same code can do the saving and restoring for all activations. Moreover programs such as debuggers will have an easier time deciphering the stack contents when an error occurs.

Ques 10) Discuss the allocation and management of Heap.

Ans: Allocation and Management of Heap
The stack allocation cannot be used if either of the following is possible:

1) The values of local names must be retained when activation ends.
2) A called activation outlives the caller. This possibility cannot occur for those languages where activation trees correctly depict the flow of control between procedures.

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

For example, let consider the following C fragment:

```
int *p, *q; /* p, q NULL pointers to integers */
p = malloc(sizeof (int));
        /* allocates the memory pointed to by p */
q = malloc(sizeof (int));
        /* allocates the memory pointed to by q */
*p = 0;  /* dereferences and assigns */
*q = 1;  /* dereferences and assigns */
free(p); /* deallocates the memory pointed to by p */
free(q); /* deallocates the memory pointed to by q */
```

Given that the memory deallocation operations are performed in the same order as allocations (first p, then q), the memory cannot be allocated in LIFO order.

To manage explicit memory allocations, which can happen at any time, a particular area of memory, called a heap, is used.

Ques 11) What are the types of memory manager functions in heap?

Ans: Types of Memory Manager Functions Using Heap
The memory manager keeps track of all the free space in heap storage at all times. It performs two basic functions:

1) **Allocation:** When a program requests memory for a variable or object, 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.
2) **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 system, even if the program's heap usage drops.

Ques 12) What are the different types of heap management methods? Discuss about them.

Or

Describe the Fixed-Length and Variable-Length Blocks.

Or

Write short note on the following:

1) **Single Free List**
2) **Multiple Free Lists**

Ans: Type of Heap Management Methods
Heap management methods fall into two main categories according to whether the memory blocks are considered to be of:

1) **Fixed-Length Blocks:** In this case, the heap is divided into a certain number of elements, or blocks,

of fairly small fixed length, linked into a list structure called the free list, as shown in **figure 5.11**.

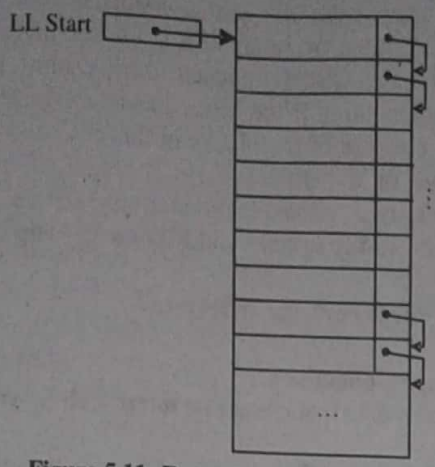


Figure 5.11: Free List in a Heap with Fixed-Size Blocks

At run-time, when an operation requires the allocation of a memory block from the heap (e.g., using the malloc command), the first element of the free list is removed from the list, the pointer to this element is returned to the operation that requested the memory and the pointer to the free list is updated so that it points to the next element.

When memory is, on the other hand, freed or deallocated (e.g., using free), the freed block is linked again to the head of the free list. The situation after some memory allocations is shown in **figure 5.12**.

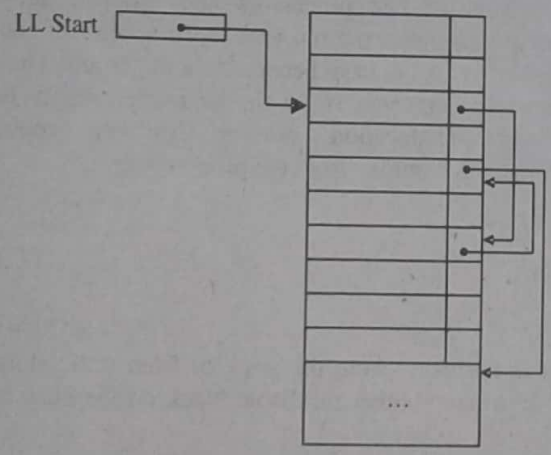


Figure 5.12: Free List for Heap with Fixed-Size Blocks after Allocation of Some Memory. Grey Blocks are Allocated (in Use)

Conceptually, therefore, management of a heap with fixed-size blocks is simple, provided that it is known how to identify and reclaim the memory that must be returned to the free list easily.

2) **Variable-Length Blocks:** In the case in which the language allows the run-time allocation of variable-length memory spaces, e.g., to store an array of variable dimension, fixed-length blocks are no longer adequate. In fact the memory to be allocated can have a size greater than the fixed block size, and the storage of an array requires a contiguous region of memory that cannot be allocated as a series of blocks. In such cases, a heap-based management scheme using variable-length blocks is used.

This type of management uses different techniques, mainly defined with the aim of increasing memory occupation and execution speed for heap management operations (that they are performed at run-time and therefore impact on the execution time of the program). As usual, these two characteristics are difficult to reconcile and good implementations tend towards a rational compromise.

In particular, as far as memory occupancy is concerned, it is a goal to avoid the phenomenon of memory fragmentation. So-called **internal fragmentation** occurs when a block of size strictly larger than the requested by the program is allocated. The portion of unused memory internal to the block clearly will be wasted until the block is returned to the free list.

But this is not the most serious problem. Indeed, so-called **external fragmentation** is worse. This occurs when the free list is composed of blocks of a relatively small size and for which, even if the sum of the total available free memory is enough, the free memory cannot be effectively used. **Figure 5.13** shows an example of this problem:

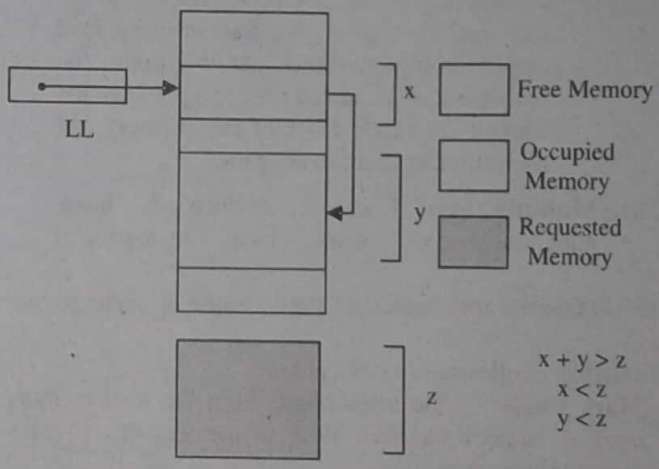


Figure 5.13: External Fragmentation

If one have blocks of size x and y (words or some other unit – it has no relevance here) on the free list and request the allocation of a block of greater size, request cannot be satisfied despite the fact that the total amount of free memory is greater than the amount of memory that has been requested. The memory allocation techniques tend therefore to "compact" free memory, merging contiguous free blocks in such a way as to avoid external fragmentation. To achieve this objective, merging operations can be called which increase the load imposed by the management methods and therefore reduce efficiency.

This can be divided into two types:

i) **Single Free List:** The first technique we examine deals with a single free list, initially composed of a single memory block containing the entire heap. It is indeed convenient to seek to maintain blocks of largest possible size. It makes no sense; therefore, initially to divide the heap into many small blocks as, on the other hand, we did in the case of fixed-size blocks. When the allocation of a block of n words of memory is requested, the first n words are allocated and the pointer to the start of the head is incremented by n. Successive requests are handled in a similar way, in which deallocated blocks are collected on a free list. When the end of the heap's memory space is reached, is necessary to reuse deallocated memory and this can be done in the two following ways:

   a) **Direct Use of the Free List:** In this case, a free list of blocks of variable size is used. When the allocation of a memory block n words in length is requested, the free list is searched for a block of size k words, where k is greater than or equal to n. The requested memory is allocated inside this block.

   b) **Free Memory Compaction:** In this technique, when the end of the space initially allocated to the heap is reached, all blocks that are still active are moved to the end; they are the blocks that cannot be returned to the free list, leaving all the free memory in a single contiguous block. At this point, the heap pointer is updated so that it points to the start of the single block of free memory and allocation starts all over again.

ii) **Multiple Free Lists:** To reduce the block allocation cost, some heap management techniques use different free lists for blocks of different sizes. When a block of size n is requested, the list that contains blocks of size greater than or equal to n is chosen and a block from this list is chosen (with some internal fragmentation if the block has a size greater than n). The size of the blocks in this case, too, can be static or dynamic and, in the case of dynamic sizes, two management methods are commonly used: **buddy system** and **Fibonacci heap**.

Ques 13) What is garbage collection?

Ans: Garbage Collections

Garbage means data that cannot be referenced (anymore).

Garbage collection is automatic deallocation of unreachable data in many programming language like C++, Java, etc., supported by many programming languages:

1) **Object-Oriented:** Java, Smalltalk.
2) **Functional:** Lisp (first GC), ML, Haskell.
3) **Logic:** Prolog
4) **Scripting:** Perl

While a programmer-controlled memory manager can be, in theory, more efficient than an automatic memory manager, in practice this is not the case. Human programmers have proved to be poor memory managers – causing both dangling references and memory leaks. With the advent of fast processors and efficient automatic memory managers, the run-time inefficiencies of complex memory managers have become less important. The trend in modern languages is for the language itself to handle memory deallocation through garbage collection, removing that burden from the programmer.

Ques 14) Discuss the mark and sweep phase of garbage collection.

Ans: Garbage collection has two phases:

1) **Mark Phase:** In the mark phase, each block of memory that is reachable from the stack or from static storage is marked. In addition, each block of memory that is reachable from some other reachable block on the heap is also marked. In pseudo-code:

```
for each pointer P on the stack
    mark(P)

mark(P){
    if((P is not null) and (mark bit of Mem[P] is not set))
      set mark bit of Mem[P]
      for each pointer Q in the block Mem[p]
          mark(Q)
}
```

If one initializes all pointers (both on the stack and on the heap) to zero, then all pointers whose value is not zero must point to valid locations on the heap. Thus to make garbage collection work correctly, one will initialize all memory locations to have the value zero. One will need to insert code in the beginning of each function to initialize all local variables to have the value zero, and one will need to modify the allocate so that all memory locations within the block are set to zero.

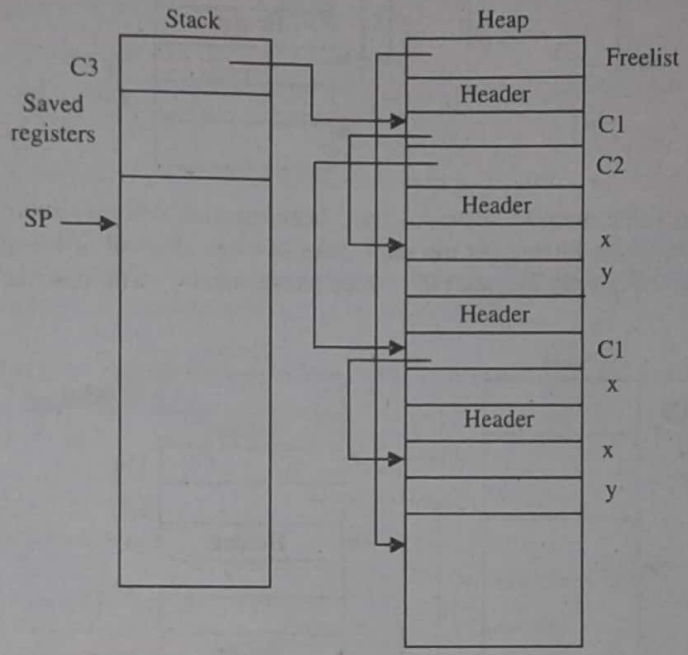
For example, now consider a concrete example. For the following simpleJava classes:

```
class Class1{
    int x;
    int y;
}
class Class2{
    Class1 C1
    int x;
}class Class3{
    Class1 C1;
    Class2 C2;
}
```

After the following code:

```
Class3 C3 = new Class3();
C3.C1 = new Class1();
C3.C2 = new Class2();
C3.C2.C1 = new Class1();
```

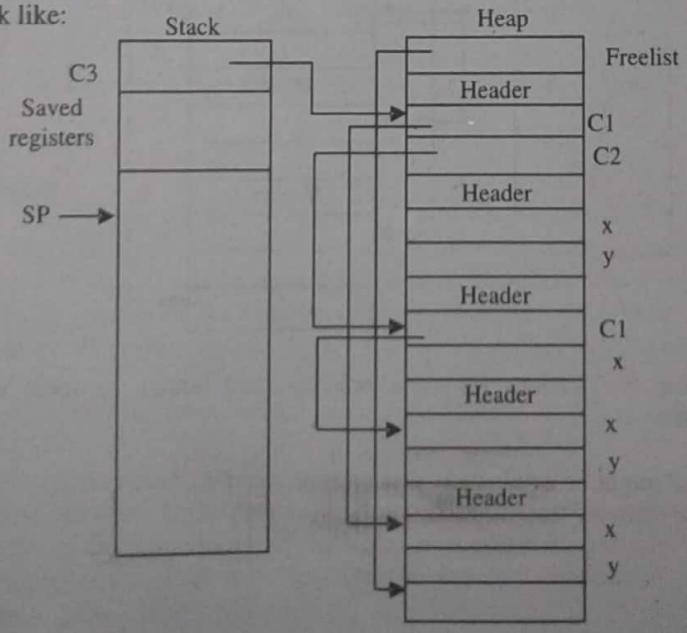
The heap and stack will look like this. This drawing is slightly stylized, and there are some bookkeeping details left out for clarity.



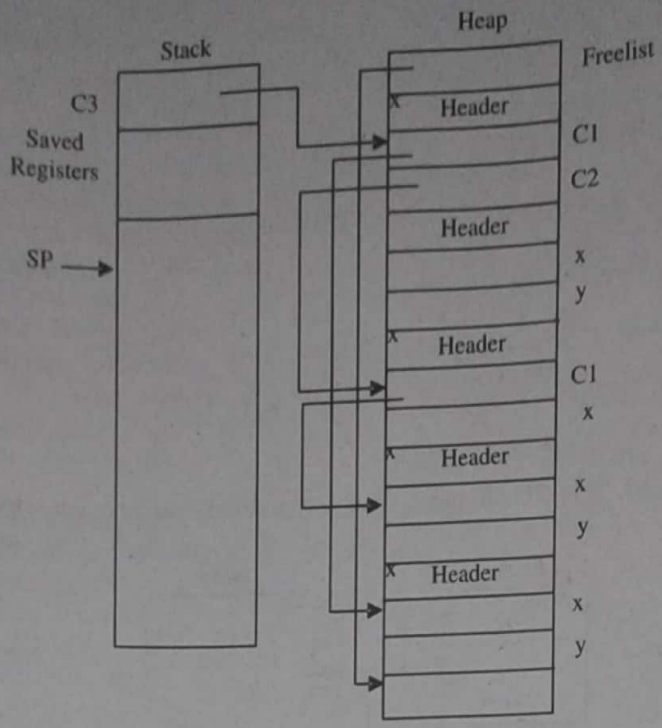
After the code;

```
C3.C1 = new Class1();
```

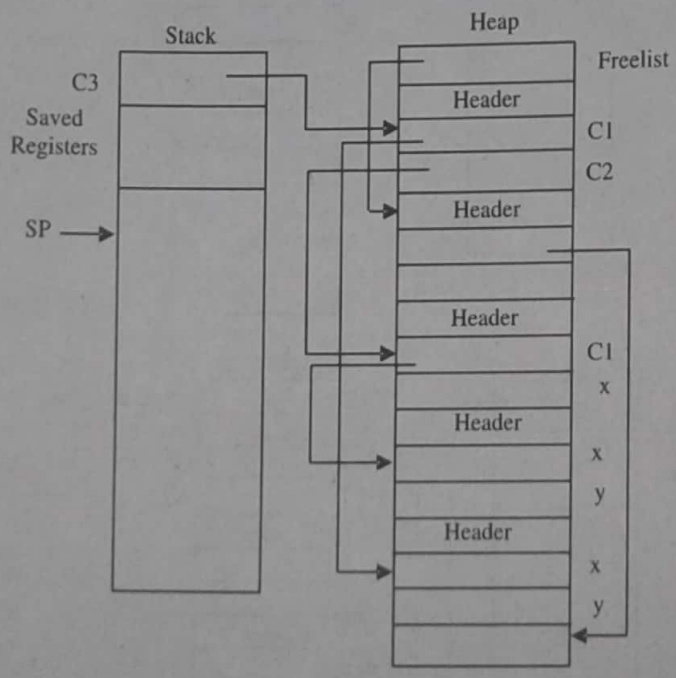
Heap memory will look like:



Note that there are new blocks of memory that cannot be reached from the stack. After running the mark phase, all locations that can be reached from the stack are marked.



2) **Sweep Phase:** One the reachable memory locations have been marked, one can sweep through memory, adding all unmarked memory locations to the freelist. At the same time, one can clear all of the marks, to be ready for the next mark phase. Continuing the above example, after the sweep phase, memory will look like the following:



How can one tell which memory locations represent pointers, and which are local variables? There are three approaches that one can follow:

i) Tag each pointer,
ii) Assume that any value that might be a pointer is a pointer, or
iii) Create tables that store the memory locations of all pointers.

INTERMEDIATE CODE GENERATION

Ques 15) What do you understand by intermediate code generation?

Ans: Intermediate Code Generation

The use of syntax–directed translation is not restricted for compiling. In many compilers the source code is translated into a language, which is intermediate in complexity between a (high–level) programming language and machine code. Such a language is therefore called **intermediate code or intermediate text.** It is possible to translate directly from source to machine or assembly language in syntax – directed way but, as we have mentioned, doing so makes generation of optimal, or even relatively good, code a difficult task.

The reason efficient machine or assembly language is hard to generate is that one is immediately forced to choose particular register to hold the result of each computation, making the efficient use of registers difficult. Therefore one usually chooses for intermediate text a notation in which, as in assembly language, each statement involves at most one arithmetic operation or one test, but where, unlike in assembly language, the register in which each operation occurs is left unspecified.

The usual intermediate text introduces symbols to stand for various temporary quantities such as the value of B*C in the source language expression A + B*C. Four kinds of intermediate code often used in compilers are **postfix notation, syntax trees, quadruples, and triples.**

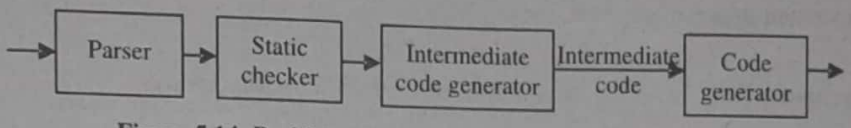


Figure 5.14: Position of Intermediate code generator

The semantic phase of a compiler first translates parse trees into an **intermediate representation (IR),** which is independent of the underlying computer architecture, and then generates machine code from the IRs. This makes the task of retargeting the compiler to another computer architecture easier to handle.

The intermediate code generator receives syntax directed syntactical constructs and represents them in any one of the following intermediate codes (**figure 5.15**):

1) Postfix,
2) Syntax tree
3) Three address code.

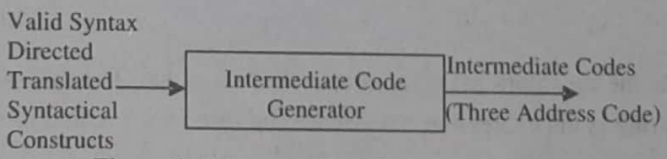


Figure 5.15: Intermediate Code Generation

Ques 16) Describe about intermediate languages in detail. What are the properties of good intermediate representations?

Ans: Intermediate Languages

An Intermediate representation (IR) is the data structure or code used internally by a compiler or virtual machine to represent source code. An IR is designed to be conducive for further processing, such as optimization and translation. A "good" IR must be accurate – capable of representing the source code without loss of information and independent of any particular source or target language. An IR may take one of several forms: an in-memory data structure or a special tuple- or stack-based code readable by the program. In the latter case it is also called an **intermediate language.**

An Intermediate Representation (IR) is a language for an abstract machine (or a language that can be easily evaluated by an abstract machine):

1) It should not include too much machine specific detail.
2) It provides a separation between front and back ends which helps compiler portability. Code generation and assignment of temporaries to registers are clearly separated from semantic analysis.
3) It allows optimization independent of the target machine. Intermediate representations are by design more abstract and uniform, so optimization routines are simpler.

An intermediate language is the language of an abstract machine designed to aid in the analysis of computer programs. The term comes from their use in compilers, where the source code of a program is translated into a form more suitable for code-improving transformations before being used to generate object or machine code for a target machine.

Properties of good intermediate representations:

1) Convenient to produce in the semantic analysis phase.
2) Convenient to translate into code for the desired target architecture.
3) Each construct must have a clear and simple meaning such that optimizing transformations that rewrite the IR can be easily specified and implemented.

Ques 17) What do you understand by postfix notation?

Ans: Postfix Notation

There are basically three types of notation for an expression (i.e., mathematical expression). An expression is defined as the number of operands or data items combined with several operators:

1) **Infix Notation:** The infix notation is what we come across in our general mathematics, where the operator is written in between the operands. **For example**, the expression to add two numbers A and B is written in infix notation as:

   A + B

2) **Prefix Notation:** The prefix notation is a notation in which the operator(s) is written before the operands. It is also called polish notation in the honor of the polish mathematician **Jan Lukasiewicz,** who developed this notation. The same expression, when written in prefix notation, looks like:

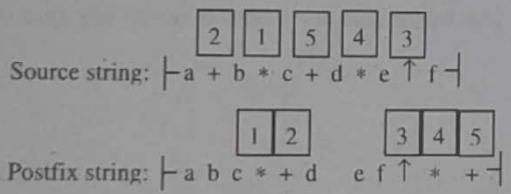
   + A B

   As the operator '+' is written before the operands A and B, this notation is called prefix.

3) **Postfix Notation:** In the postfix notation the operator(s) are written after the operands, so it is called the postfix notation (post means after). It is also known as suffix notation or reverse polish notation. The above expression, if written in postfix expression, looks like;

   A B +

   Example: Consider the following strings:



   The numbers appearing over the operators indicate their evaluation order. The second operand of the operator '+' marked [2] in the source string is the expression b*c. Hence, its operands b, c and its operator '*' appear before '+' in the postfix string.

Ques 18) What is the graphical representation of intermediate language?

Or

What is syntax tree?

Ans: Graphical Representation of Intermediate Language: Syntax Trees

A Syntax or parse tree pictorially shows how the start symbol of a grammar derives a string in the language or it may be defined as a graphical representation that filters out the choice regarding replacement order. This representation is called **syntax tree**.

A derivation provides a method for constructing a particular string of terminals from a starting non-terminal. But derivations do not uniquely represent the structure of the strings they construct. In general, there are many derivations for the same string.

For example, we constructed the string of tokens:

(number – number) * number

from our simple expression grammar using the derivation in **below**.

1) exp ⇒ exp op exp	[exp → exp op exp]
2) ⇒ (exp) op number	[exp → number]
3) ⇒ exp * number	[exp → *]
4) ⇒ (exp) * number	[exp → (exp)]
5) ⇒ (exp or exp) * number	[op → exp or exp]
6) ⇒ (number op number) * number	[exp → number]
7) ⇒ (exp – number) * number	[op → –]
8) ⇒ (number – number) * number	[exp → number]

Figure 5.16: A Derivation for the Arithmetic Expression (34 – 3) * 42

A second derivation for this string is given in **below**. The only difference between the two derivations is the order in which the replacements are supplied, and this is in fact a superficial difference. To make this clear, we need a representation for the structure of a string of terminals that abstracts the essential features of a derivation while factoring-out superficial differences in ordering. The representation that does this is a tree structure, and is called a parse tree.

1) exp ⇒ exp op exp	[exp → exp op exp]
2) ⇒ (exp) op exp	[exp → (exp)]
3) ⇒ (exp op exp) op exp	[exp → exp op exp]
4) ⇒ (number op exp) op exp	[exp → number]
5) ⇒ (number – exp) op exp	[op → –]
6) ⇒ (number – number) op exp	[exp → number]
7) ⇒ (number – number) * exp	[op → *]
8) ⇒ (number – number) * number	[exp → number]

Figure 5.17: Another Derivation for the Expression (34 – 3) * 42

For example, To give a simple example, the derivation

exp ⇒ exp op exp
⇒ number op exp
⇒ number + exp
⇒ number + number

corresponds to the syntax tree

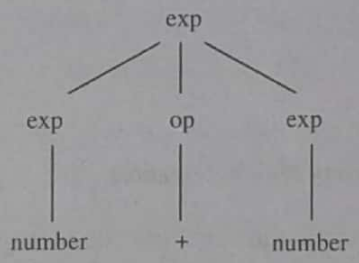


Figure 5.18

A more complex example of a syntax tree and leftmost and rightmost derivations to the expression (34 -3)* 42 and the derivations in **figure 5.16** and **5.17**. This syntax tree for this expression is given in **figure 5.19**.

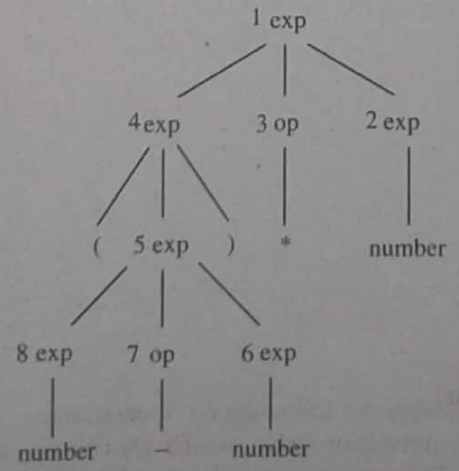


Figure 5.19: Parse Tree for the Arithmetic Expression (34 – 3) * 42

Ques 19) Consider the grammar

$$E \rightarrow E + E / E * E / (E) / - E / id$$

Construct the syntax tree for the string id + id * id.

Solution:

Left Most Derivation	Right Most Derivation
E → E + E	E → E * E
→ id + E	→ E + E * E
→ id + E * E	→ E + E * id
→ id + id * E	→ E + id * id
→ id + id * id	→ id + id * id

The syntax tree is shown below:

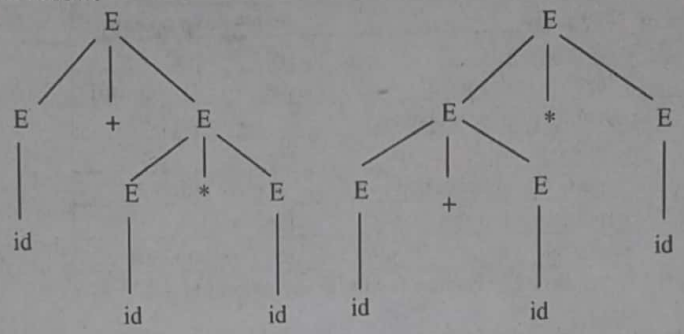


Figure 5.20: Two Parse Trees for id + id * id

Ques 20) What is three address code? What are the different types of three address code representation?

Or

Discuss about the following:

1) **Quadruples**
2) **Triples**
3) **Indirect Triples**

Ans: Three Address Code

Three address codes is a term used to describe a variety of representations.

This intermediate form is called **three-address** because each "line" of code contains one operator and up to three operands, represented as addresses. Since most assembly languages represent a single operation in an instruction, three-address code is closer to the target code than the parse tree representation. There are a number of variants of three-address code, some more appropriate than others for optimization.

In general, three address codes allow statements of the form:

$x \leftarrow y \text{ op } z$

with a single operator and, at most, three names.

Simpler form of expression

$x - 2 * y$

becomes

$t_1 \leftarrow 2 * y$

$t_2 \leftarrow x - t_1$

Types of Three Address Code Representation

The implementation of three address codes can be done by following representations:

1) **Quadruples:** A quadruple (or just "quad") have four fields, which we call op, arg_1, arg_2, and result. The op field contains an internal code for the operator. **For example**, the three-address instruction x = y + z is represented by placing + in op, y in arg_1, z in arg_2, and x in result.

The following are some exceptions to this rule:

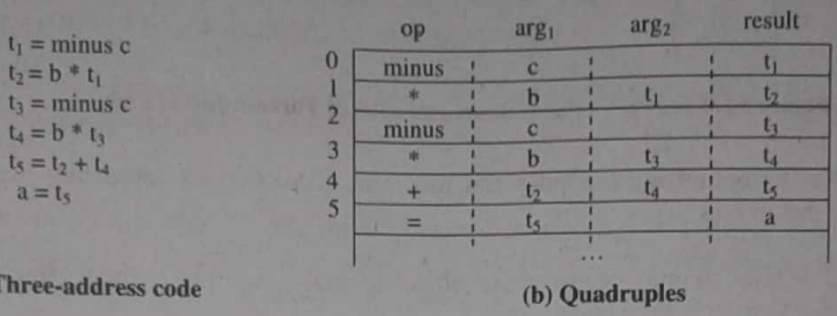
i) Instructions with unary operators like x = minus y or x = y do not use arg_2. Note that for a copy statement like x = y, op is =, while for most other operations, the assignment operator is implied.

ii) Operators like param use neither arg_2 nor result.

iii) Conditional and unconditional jumps put the target label in result.

Example: Three-address code for the assignment a = b * – c + b * – c; appears in **figure 5.21(a)**. The special operator minus is used to distinguish the unary minus operator, as in – c, from the binary minus operator, as in b – c. Note that the unary-minus "three-address" statement has only two addresses, as does the copy statement $a = t_5$.

The quadruples in **figure 5.21 (b)** implement the three-address code in **figure 5.21 (a)**:



(a) Three-address code **(b) Quadruples**

Figure 5.21: Three-Address Code and its Quadruple Representation

For readability, we use actual identifiers like a, b, and c in the fields arg_1, arg_2 and result in **figure 5.21 (b)**, instead of pointers to their symbol-table entries. Temporary names can either by entered into the symbol table like programmer-defined names, or they can be implemented as objects of a class Temp with its own methods.

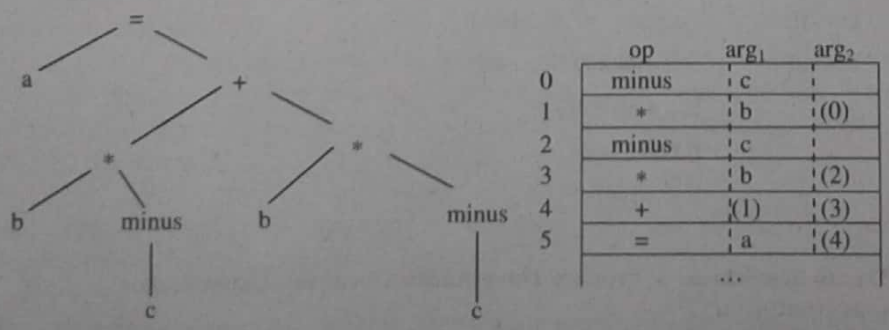
2) **Triples:** The contents of the operand1, operand2, and result fields are therefore normally the pointers to the symbol records for the names represented by these fields. Hence, it becomes necessary to enter temporary names into the symbol table as they are created. This can be avoided by using the position of the statement to refer to a temporary value. If this is done, then a record structure with three fields is enough to represent the three-address statements: the first holds the operator value, and the next two holding values for the operand1 and operand2, respectively. Such a representation is called a "triple representation". The contents of the operand1 and operand2 fields are either pointers to the symbol table records, or they are pointers to records (for temporary names) within the triple representation itself.

For example, a triple representation of the three-address code for the statement x = (a + b) * – c/d is shown in **table 5.1**.

Table 5.1: Triple Representation of x = (a + b) * – c/d

	Operator	Operand1	Operand2
(1)	+	a	b
(2)	–	c	
(3)	*	(1)	(2)
(4)	/	(3)	d
(5)	=	x	(4)

3) **Indirect Triples:** Indirect triples consist of a listing of pointers to triples, rather than a listing of triples themselves. **For example,** let us use an array instruction to list pointers to triples in the desired order. Then, the triples in **figure 5.22 (b)** might be represented as in **figure 5.23**:



(a) Syntax Tree **(b) Triples**

Figure 5.22: Representations of a + a * (b – c) + (b – c) * d

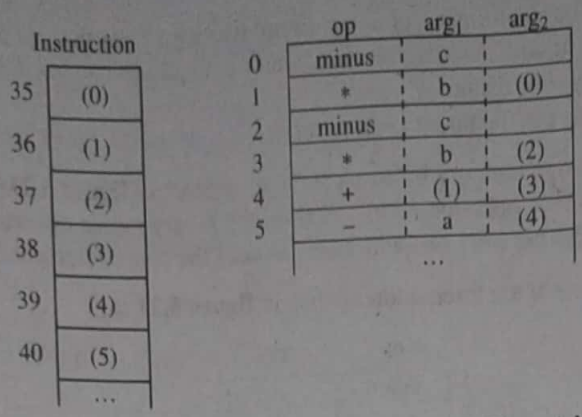


Figure 5.23: Indirect Triples Representation of Three-Address Code

With indirect triples, an optimizing compiler can move an instruction by reordering the instruction list, without affecting the triples themselves.

When implemented in Java, an array of instruction objects is analogous to an indirect triple representation, since Java treats the array elements as references to objects.

Ques 21) Discuss assignment statements with example.

Ans: Assignment Statements

Expression can be of type integer, real, array, and record. As part of the translations of assignments into three-address code, we show how names can be looked up in the symbol table and how elements of arrays and records can be accessed.

Name in the Symbol Table

The translation scheme in **figure 5.24** shows how such symbol-table entries can be found. The lexeme for the name represented by id is, given by attribute id.name. Operation lookup (id.name) checks if there is an entry for this occurrence of the name in the symbol table. If so, a pointer to the entry is returned; otherwise, lookup returns nil to indicate that no entry was found.

$S \rightarrow id := E$	{p := lookup (id.name):
	If p ≠ nil then
	emit (p ':=' E.place)
	else error}
$E \rightarrow E_1 + E_2$	{E.place := newtemp;
	emit (E.place ':=' E_1.place '+' E_2.place)}
$E \rightarrow E_1 * E_2$	{E.place := newtemp;
	emit (E.place ':=' E_1.place '*' E_2.place)}
$E \rightarrow - E_1$	{E.place := newtemp;
	emit (E.place ':=' 'uminus' E_1.place)}
$E \rightarrow (E_1)$	{E.place ':=' E_1.place}
$E \rightarrow id$	{p := lookup (id.name):
	If p ≠ nil then
	E.place := p
	else error}

Figure 5.24: Translation Scheme to Produce Three-Address Code for Assignments.

The semantic actions in **figure 5.24** use procedure emit to emit three-address statements to an output file, rather than building up code attributes for non-terminals, translation can be done by emitting to an output file if the code attributes of the non-terminals on the left sides of production are formed by concatenating the code attributes of the nonterminal on the right, in the same order that the nonterminal appear on the right side, perhaps with some additional strings in between.

By reinterpreting the lookup operation in **figure 5.24,** the translation scheme can be used even if the most closely nested scope rule applies to nonlocal names, as in Pascal, for concreteness, suppose that the context in which an assignment appears is given by the following grammar:

$P \rightarrow MD$

$M \rightarrow \in$

$D \rightarrow D: D \mid id: T \mid pro\ id: N\ D: S$

$N \rightarrow \in$

Nonterminal P becomes the new start symbol when these productions are added to those in **Figure 5.24.**
For each procedure generated by this grammar, the translation scheme sets up a separate symbol table. Each such symbol table has a header containing a pointer to the table for the enclosing procedure.

When the statement forming a procedure body is examined, a pointer to the symbol table for the procedure appears on top of the stack tblptr. This pointer is pussed onto the stack by actions associated with the marker nonterminal N on the right side of $D \rightarrow$ proc id: N D_1; S.

Let the productions for nonterminal S be those in **Figure 5.24** Names in an assignment generated by S must have been declared in either the procedure that S appears in, or in some enclosing procedure. When applied to name, the modified lookup operations first checks if name appears in the current symbol table, accessible through top (tblptr). If not, lookup uses the pointer in the header of a table to find the symbol table for the enclosing procedure and looks for the name there. If the name cannot be found in any of these scope, then lookup returns nil.

For example, suppose that the symbol tables are as in **figure 5.25** and that an assignment in the body of procedure partition is being examined. Operation lookup (i) will find an entry in the symbol table for partition. Since v is not in this symbol table, lookup (v) will use the pointer in the header in this symbol table to continue the search in the symbol table for the enclosing producer quicksort.

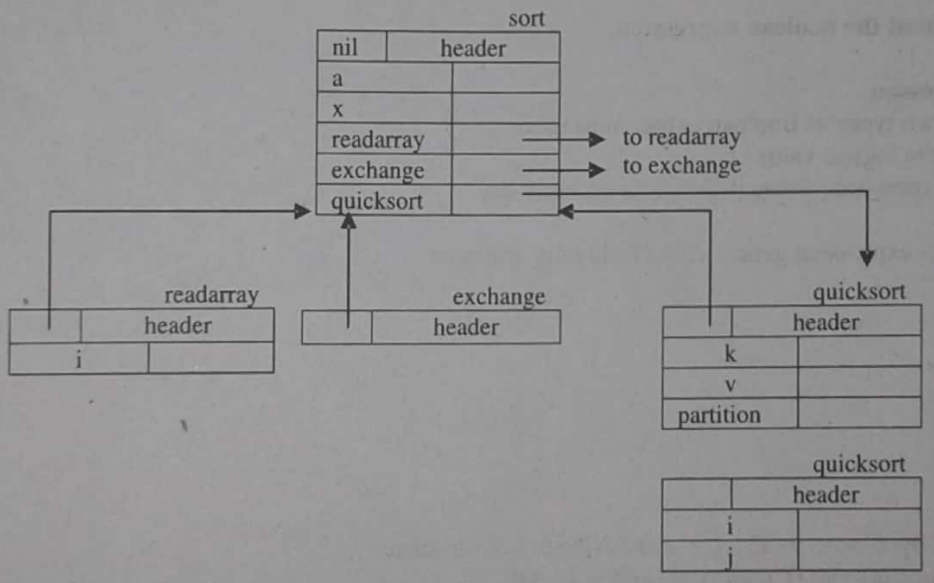


Figure 5.25: Symbol Tables for Nested Procedures

Reusing Temporary Names

We have been going along assuming that newtemp generates a new temporary name each time a temporary in need. It is useful, especially in optimising compilers, to actually created a distance name each time newtemp is called. However, the temporaries used to hold intermediate values in expression calculations tend to clutter up the symbol table, and space has to be allocated to hold their values.

Temporaries can be reused by changing newtemp. An alternative approach of packing distinct temporaries into the same location during code generation in explored in the next chapter.

The bulk of temporaries denoting data are generated during the synatx-directed translation of expressions, by rules such as those in **Figure 5.24.** The code generated by the rules for $E \rightarrow E_1 + E_2$ has the general form:

evaluate E_1 into t_1

evaluate E_2 into t_2

$t := t_1 + t_2$

From the rules for the synthesised attribute E.place it follows that t_1 and t_2 are not used elsewhere in the program. The lifetimes of these temporaries are nested like matching pairs of balanced parentheses. In fact, the lifetimes of all temporaries used in the evolution of E_2 are contained in the lifetime of t_1. It is therefore possible to modify newtemp so that it uses, as if it were a stack, a small array in a procedure's data area to hold temporaries.

Let us assume for simplicity that we are dealing only with integers. Keep a count c, initialised to zero. Whenever a temporary name is used as an operand, decrement c by 1. Whenever a new temporary name is generated, use $c and increase c by 1. Note that the "stack" of temporaries is not pushed or popped at run time, although it happens that stores and loads of temporary values are made by the complier to occur at the "top".

Example: Consider the assignment
x := a * b + c * d – e * f

Figure 5.26 shows the sequence of three-address statements that would be generated by semantic rules in **Figure 5.24** if newtemp were modified. The **figure** also contains an indication of the "current" value of c after the generation of each statement. Note that when we compute $0 – $1, c is decremented to zero, so $0 is again available to hold the result.

Statement	Count
	0
t0 := a * b	1
t1 := c * d	2
t0 := t0 + t1	1
t1 := e * f	2
t0 := t0 – t1	1
x := t0	0

Figure 5.26:

Ques 22) Discuss about the boolean expression.

Ans: Boolean Expression
Normally there are two types of Boolean expressions used:
1) For computing the logical values.
2) In conditional expressions using if-then-else or while-do.

Consider the Boolean expression generated by following grammar:
E - > E OR E
E - > E AND E
E - > NOT E
E - > (E)
E - > id relop id
E - > TRUE
E - > FALSE

The relop is denoted by <, >, <, >. The OR and AND are left associate.
The highest precedence is to NOT then AND and lastly OR.

E - > E1 OR E2	{
	E.place : = newtemt( )
	Emit (E. place ':= E1. Place "OR" E2.Place
	}

E - > E1 AND E2	{
	E. place := newtemt( )
	Emit (E.place ':= E1. Place "AND" E2.Place
	}

E - > NOT E1	{
	E. place := newtemt( )
	Emit (E.place ':= "NOT" E1.Place
	}

E - > (E1)	{ E.place := E1.Place}

E - > id_1 relop id_2	{
	E.place := newtemp( )
	Emit ('if id. Place relop.op id_2, place 'goto' next_state + 3;
	Emit (E.place':= '0');
	Emit ('goto' next state + 2);
	Emit (E.place ':=' '1')
	}

E - > TRUE	{
	E.place := newtemp( );
	Emit (E.place ':=' '1')
	}

E - > FALSE	{
	E. place := newtemp ( )
	Emit (E.place ':=' '0')
	}

The function Emit generates the three address code and newtemp ( ) is for generation of temporary variables.

For the semantic action for the rule E - > id1 relop id2 contains next_state which gives the index of next three address statement in the output sequence.

Let us take an **example** and generate the three address code using above translation scheme:

p > q AND r < s OR u > v

100: if p > q goto 103
101: t1: = 0
102: goto 104
103: t1: = 1
104: if r < s goto 107
105: t2: = 0
106: goto 108
107: t2 = 1
108: if u > v goto 111
109: t3: = 0
110: goto 112
111: t3 = 1
112: t4 = t1 AND t2
113: t5 = t4 OR t3