Compiler Construction
Principles and Practice

Kenneth C. Louden
San Jose State University

PWS Publishing Company

I(T)P

An International Thomson Publishing Company

Boston • Albany • Bonn • Cincinnati • Detroit • London • Madrid • Melbourne • Mexico City • New York

Pacific Grove • Paris • San Francisco • Singapore • Tokyo • Toronto • Washington

Chapter 7

Runtime Environments

In previous chapters we have studied the phases of a compiler that perform static analysis of the source language. This has included scanning, parsing, and static semantic analysis. This analysis depends only on the properties of the source language—it is completely independent of the target (assembly or machine) language and the properties of the target machine and its operating system.

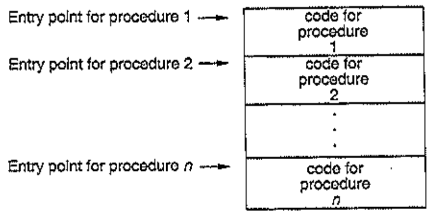
In this chapter and the next we turn to the task of studying how a compiler generates executable code. This can involve additional analysis, such as that performed by an optimizer, and some of this can be machine independent. But much of the task of code generation is dependent on the details of the target machine. Nevertheless, the general characteristics of code generation remain the same across a wide variety of architectures. This is particularly true for the **runtime environment**, which is the structure of the target computer's registers and memory that serves to manage memory and maintain the information needed to guide the execution process. In fact, almost all programming languages use one of three kinds of runtime environment, whose essential structure does not depend on the specific details of the target machine. These three kinds of environments are the **fully static environment** characteristic of FORTRAN77; the **stack-based environment** of languages like C, C++, Pascal, and Ada; and the **fully dynamic environment** of functional languages like LISP. Hybrids of these are also possible.

In this chapter, we will discuss each of these kinds of environments in turn, together with the language features that dictate which environments are feasible, and what their properties must be. This includes scoping and allocation issues, the nature of procedure calls, and the varieties of parameter passing mechanisms. Here, the focus will be on the general structure of the environment, while in the next chapter, the focus will be on the actual code that needs to be generated to maintain the environment. In this regard, it is important to keep in mind that a compiler can maintain an environment only indirectly, in that it must generate code to perform the necessary maintenance operations during program execution. By contrast, an interpreter has an easier task, since it can maintain the environment directly within its own data structures.

The first section of this chapter contains a discussion of the general characteristics of all runtime environments and their relationship to the architecture of the target machine. The next two sections discuss the static and stack-based environments, together with examples of their operation during execution. Since the stack-based environment is the most common, some detail is given about the different varieties and structure of a stack-based system. A subsequent section discusses dynamic memory issues, including fully dynamic environments and object-oriented environments. Following that is a discussion of the effect of various parameter passing techniques on the operation of an environment. The chapter closes with a brief description of the simple environment needed to implement the TINY language.

7.1 MEMORY ORGANIZATION DURING PROGRAM EXECUTION

The memory of a typical computer is divided into a register area and a slower directly addressible random access memory (RAM). The RAM area may be further divided into a code area and a data area. In most compiled languages, it is not possible to make changes to the code area during execution, and the code and data area can be viewed as conceptually separate. Further, since the code area is fixed prior to execution, all code addresses are computable at compile time, and the code area can be visualized as follows:



Entry point for procedure 1
code for procedure 1
Entry point for procedure 2
code for procedure 2
Entry point for procedure n
code for procedure n

Code memory

In particular, the entry point of each procedure and function is known at compile time.[1] The same cannot be said for the allocation of data, only a small part of which can be assigned fixed locations in memory before execution begins. Much of the rest of the chapter will be taken up with how to deal with nonfixed, or dynamic, data allocation.

1. More likely, the code is loaded by a loader into an area in memory that is assigned at the beginning of execution and, thus, is not absolutely predictable. However, all actual addresses are then automatically computed by offset from a fixed base load address, so the principle of fixed addresses remains the same. Sometimes, the compiler writer must take care to generate so-called **relocatable code**, in which jumps, calls, and references are all performed relative to some base, usually a register. Examples of this will be given in the next chapter.

There *is* one class of data that can be fixed in memory prior to execution and that comprises the global and/or static data of a program. (In FORTRAN77, unlike most languages, all data are in this class.) Such data are usually allocated separately in a fixed area in a similar fashion to the code. In Pascal, global variables are in this class, as are the external and static variables of C.

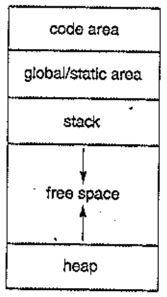
One question that arises in the organization of the global/static area involves constants that are known at compile time. These include the `const` declarations of C and Pascal, as well as literal values used in the code itself, such as the string `"Hello %d\n"` and the integer value 12345 in the C statement

```
printf("Hello %d\n",12345);
```

Small compile-time constants such as 0 and 1 are usually inserted directly into the code by the compiler and are not allocated any data space. Also, no space needs to be allocated in the global data area for global functions or procedures, since their entry points are known to the compiler and can be inserted directly into the code as well. However, large integer values, floating-point values, and particularly string literals are usually allocated memory in the global/static area, stored once on start-up and then are fetched from those locations by the executing code. (Indeed, in C string literals are viewed as pointers, so they must be stored in this way.)

The memory area used for the allocation of dynamic data can be organized in many different ways. A typical organization divides this memory into a **stack** area and a **heap** area, with the stack area used for data whose allocation occurs in LIFO (last-in, first-out) fashion and the heap area used for dynamic allocation that does not conform to a LIFO protocol (pointer allocation in C, for example).[2] Often the architecture of the target machine will include a processor stack, and using this stack makes it possible to use processor support for procedure calls and returns (the principal mechanism that uses stack-based memory allocation). Sometimes, a compiler will have to arrange for the explicit allocation of the processor stack in an appropriate place in memory.

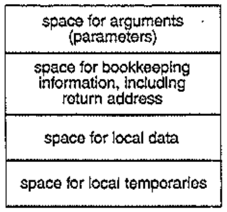
A general organization of runtime storage that has all of the described memory categories might look as follows:



2. It should be noted that the heap is usually a simple linear memory area. It is called a heap for historical reasons and is unrelated to the heap data structure used in algorithms such as heapsort.

The arrows in this picture indicate the direction of growth of the stack and heap. Traditionally, the stack is pictured as growing downward in memory, so that its top is actually at the bottom of its pictured area. Also, the heap is pictured as being similar to the stack, but it is not a LIFO structure and its growth and shrinkage is more complicated than the arrow indicates (see Section 7.4). In some organizations the stack and heap are allocated separate sections of memory, rather than occupying the same area.

An important unit of memory allocation is the **procedure activation record**, which contains memory allocated for the local data of a procedure or function when it is called, or activated. An activation record, at a minimum, must contain the following sections:


space for arguments (parameters)
space for bookkeeping information, including return address
space for local data
space for local temporaries

We emphasize here (and repeatedly in following sections) that this picture only illustrates the general organization of an activation record. Specific details, including the order of the data it contains, will depend on the architecture of target machine, the properties of the language being compiled, and even the taste of the compiler writer.

Some parts of an activation record have the same size for all procedures—the space for bookkeeping information, for example. Other parts, like the space for arguments and local data, may remain fixed for each individual procedure, but will vary from procedure to procedure. Also, some parts of the activation record may be allocated automatically by the processor on procedure calls (storing the return address, for example). Other parts (like the local temporary space) may need to be allocated explicitly by instructions generated by the compiler. Depending on the language, activation records may be allocated in the static area (FORTRAN77), the stack area (C, Pascal), or the heap area (LISP). When activation records are kept in the stack, they are sometimes referred to as **stack frames**.

Processor registers are also part of the structure of the runtime environment. Registers may be used to store temporaries, local variables, or even global variables. When a processor has many registers, as in the newer RISC processors, the entire static area and whole activation records may be kept entirely in registers. Processors also have special-purpose registers to keep track of execution, such as the **program counter (pc)** and **stack pointer (sp)** in most architectures. There may also be registers specifically designed to keep track of procedure activations. Typical such registers are the **frame pointer (fp)**, which points to the current activation record, and the **argument pointer** **(ap)**, which points to the area of the activation record reserved for arguments (parameter values).[3]

A particularly important part of the design of a runtime environment is the determination of the sequence of operations that must occur when a procedure or function is called. Such operations may include the allocation of memory for the activation record, the computation and storing of the arguments, and the storing and setting of the necessary registers to effect the call. These operations are usually referred to as the **calling sequence**. The additional operations needed when a procedure or function returns, such as the placing of the return value where it can be accessed by the caller, the readjustment of registers, and the possible releasing of activation record memory, are commonly also considered to be part of the calling sequence. If necessary, we will refer to the part of the calling sequence that is performed during a call as the **call sequence** and the part that is performed on return the **return sequence**.

Important aspects of the design of the calling sequence are (1) how to divide the calling sequence operations between the caller and callee (that is, how much of the code of the calling sequence to place at the point of call and how much to place at the beginning of the code of each procedure) and (2) to what extent to rely on processor support for calls rather that generating explicit code for each step of the calling sequence. Point (1) is a particularly thorny issue, since it is usually easier to generate calling sequence code at the point of call rather than inside the callee, but doing so causes the size of the generated code to grow, since the same code must be duplicated at each call site. These issues will be handled in more detail later on.

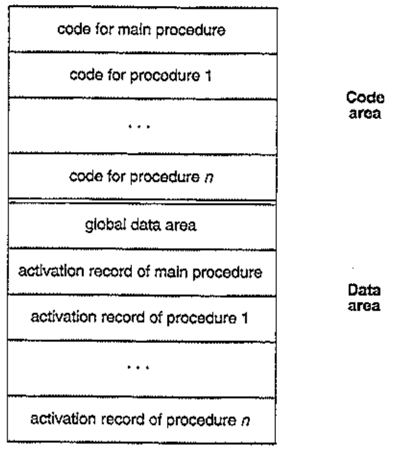
At a minimum, the caller is responsible for computing the arguments and placing them in locations where they may be found by the callee (perhaps directly in the activation record of the callee). In addition, the state of the machine at the point of call, including the return address and, possibly, registers that are in use, must be saved, either by the caller or the callee, or partially by both. Finally, any additional bookkeeping information must also be set up, again in some possibly cooperative manner between caller and callee.

7.2 FULLY STATIC RUNTIME ENVIRONMENTS

The simplest kind of a runtime environment is that in which all data are static, remaining fixed in memory for the duration of program execution. Such an environment can be used to implement a language in which there are no pointers or dynamic allocation, and in which procedures may not be called recursively. The standard example of such a language is FORTRAN77.

In a fully static environment not only the global variables, but *all* variables are allocated statically. Thus, each procedure has only a single activation record, which is allocated statically prior to execution. All variables, whether local or global, can be accessed directly via fixed addresses, and the entire program memory can be visualized as follows:

3. These names are taken from the VAX architecture, but similar names occur in other architectures.



In such an environment there is relatively little overhead in terms of bookkeeping information to retain in each activation record, and no extra information about the environment (other than possibly the return address) needs to be kept in an activation record. The calling sequence for such an environment is also particularly simple. When a procedure is called, each argument is computed and stored into its appropriate parameter location in the activation of the procedure being called. Then the return address in the code of the caller is saved, and a jump is made to the beginning of the code of the called procedure. On return, a simple jump is made to the return address.[4]

Example 7.1 As a concrete example of this kind of environment, consider the FORTRAN77 program of Figure 7.1. This program has a main procedure and a single additional procedure **QUADMEAN**.[5] There is a single global variable given by the **COMMON MAXSIZE** declaration in both the main procedure and **QUADMEAN**.[6]

4. In most architectures, a subroutine jump automatically saves the return address; this address is also automatically reloaded when a return instruction is executed.

5. We ignore the library function **SQRT**, which is called by **QUADMEAN** and which is linked in prior to execution.

6. In fact, FORTRAN77 allows COMMON variables to have different names in different procedures, while still referring to the same memory location. From now on in this example, we will silently ignore such complexities.

Figure 7.1
A FORTRAN77 sample program

```
      PROGRAM TEST
      COMMON MAXSIZE
      INTEGER MAXSIZE
      REAL TABLE(10),TEMP
      MAXSIZE = 10
      READ *, TABLE(1),TABLE(2),TABLE(3)
      CALL QUADMEAN(TABLE,3,TEMP)
      PRINT *, TEMP
      END

      SUBROUTINE QUADMEAN(A,SIZE,QMEAN)
      COMMON MAXSIZE
      INTEGER MAXSIZE,SIZE
      REAL A(SIZE),QMEAN, TEMP
      INTEGER K
      TEMP = 0.0
      IF ((SIZE.GT.MAXSIZE).OR.(SIZE.LT.1)) GOTO 99
      DO 10 K = 1,SIZE
        TEMP = TEMP + A(K)*A(K)
10    CONTINUE
99    QMEAN = SQRT(TEMP/SIZE)
      RETURN
      END
```

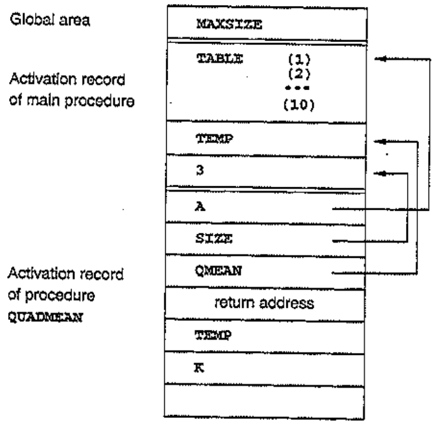
Ignoring the possible size difference between integer and floating-point values in memory, we show a runtime environment for this program in Figure 7.2 (page 352).[7] In this picture we have drawn in arrows to indicate the values that the parameters **A**, **SIZE**, and **QMEAN** of procedure **QUADMEAN** have during the call from the main procedure. In FORTRAN77, parameter values are implicitly memory references, so the locations of the arguments of the call (**TABLE**, **3**, and **TEMP**) are copied into the parameter locations of **QUADMEAN**. This has several consequences. First, an extra dereference is required to access parameter values. Second, array parameters do not need to be reallocated and copied (thus, array parameter **A** in **QUADMEAN** is allocated only one space, which points to the base location of **TABLE** during the call). Third, constant arguments, such as the value 3 in the call, must be stored to a memory location and this location used during the call. (Parameter passing mechanisms are more fully discussed in Section 7.5.)

There is one more feature of Figure 7.2 that requires explanation, and that is the unnamed location allocated at the end of the activation record of **QUADMEAN**. This location is a "scratch" location used to store temporary values during the computation of arithmetic expressions. There are two computations in **QUADMEAN** where this may

7. Again we emphasize that the details of this picture are meant to be illustrative only. Actual implementations may differ substantially from those given here.

be needed. The first is the computation of `TEMP + A(K)*A(K)` in the loop, and the second is the computation of `TEMP/SIZE` as the parameter in the call to `SQRT`. We have already discussed the need to allocate space for parameter values (although in a call to a library function the convention may in fact be different). The reason a temporary memory location may also be needed for the loop computation is that each arithmetic operation must be applied in a single step, so that `A(K)*A(K)` is computed and then added to the value of `TEMP` in the next step. If there are not enough registers available to hold this temporary value, or if a call is made requiring this value to be saved, then the value will be stored in the activation record prior to the completion of the computation. A compiler can always predict whether this will be necessary during execution, and arrange for the allocation of the appropriate number (and size) of temporary locations.

Figure 7.2
A runtime environment for the program of Figure 7.1


Global area
MAXSIZE
Activation record of main procedure
TABLE (1) (2) ... (10)
TEMP
3
A
SIZE
QMEAN
Activation record of procedure QUADMEAN
return address
TEMP
K

§

7.3 STACK-BASED RUNTIME ENVIRONMENTS

In a language in which recursive calls are allowed, and in which local variables are newly allocated at each call, activation records cannot be allocated statically. Instead, activation records must be allocated in a stack-based fashion, in which each new activation record is allocated at the top of the stack as a new procedure call is made (a **push** of the activation record) and deallocated again when the call exits (a **pop** of the activation record). The **stack of activation records** (also referred to as the **runtime stack** or **call stack**) then grows and shrinks with the chain of calls that have occurred in the executing program. Each procedure may have several different activation records on the call stack at one time, each representing a distinct call. Such an environment requires a more complex strategy for bookkeeping and variable access than a fully static environment. In particular, additional bookkeeping information must be kept in the activation records, and the calling sequence must also include the steps necessary to set up and

maintain this extra information. The correctness of a stack-based environment, and the amount of bookkeeping information required, depends heavily on the properties of the language being compiled. In this section we will consider the organization of stack-based environments in order of increasing complexity, classified by the language properties involved.

7.3.1 Stack-Based Environments Without Local Procedures

In a language where all procedures are global (such as the C language), a stack-based environment requires two things: the maintenance of a pointer to the current activation record to allow access to local variables and a record of the position or size of the immediately preceding activation record (the caller's activation record) to allow that activation record to be recovered (and the current activation to be discarded) when the current call ends. The pointer to the current activation is usually called the **frame pointer**, or **fp**, and is usually kept in a register (often also referred to as the fp). The information about the previous activation is commonly kept in the current activation as a pointer to the previous activation record and is referred to as the **control link** or **dynamic link** (*dynamic*, since it points to the caller's activation record during execution). Sometimes this pointer is called the **old fp**, since it represents the previous value of the fp. Typically, this pointer is kept somewhere in the middle of the stack, between the parameter area and the local variable area, and points to the location of the control link of the previous activation record. Additionally, there may be a **stack pointer**, or **sp**, which always points to the last location allocated on the call stack (sometimes this is called the **top of stack** pointer, or **tos**).

We consider some examples.

Example 7.2

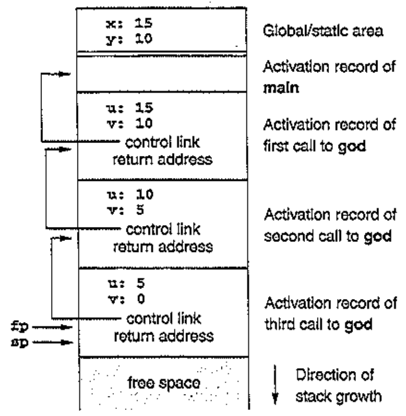
Consider the simple recursive implementation of Euclid's algorithm to compute the greatest common divisor of two nonnegative integers, whose code (in C) is given in Figure 7.3.

Figure 7.3
C code for Example 7.2

```
#include <stdio.h>

int x,y;

int gcd( int u, int v)
{ if (v == 0) return u;
  else return gcd(v,u % v);
}

main()
{ scanf("%d%d",&x,&y);
  printf("%d\n",gcd(x,y));
  return 0;
}
```

Suppose the user inputs the values 15 and 10 to this program, so that `main` initially makes the call `gcd(15,10)`. This call results in a second, recursive call `gcd(10,5)` (since 15 % 10 = 5), and this results in a third call `gcd(5,0)` (since 10 % 5 = 0), which then returns the value 5. During the third call the runtime environment may be visualized as in Figure 7.4. Note how each call to `gcd` adds a new activation record of exactly the same size to the top of the stack, and in each new activation record, the control link points to the control link of the previous activation record. Note also that the fp points to the control link of the current activation record, so on the next call the current fp becomes the control link of the next activation record.

Figure 7.4
Stack-based environment for Example 7.2


x: 15
y: 10
Global/static area
Activation record of main
u: 15
v: 10
control link
return address
Activation record of first call to gcd
u: 10
v: 5
control link
return address
Activation record of second call to gcd
u: 5
v: 0
control link
return address
Activation record of third call to gcd
fp
sp
free space
Direction of stack growth

After the final call to `gcd`, each of the activations is removed in turn from the stack, so that when the `printf` statement is executed in `main`, only the activation record for `main` and the global/static area remain in the environment. (We have shown the activation record of `main` as empty. In reality, it would contain information that would be used to transfer control back to the operating system.)

Finally, we remark that no space in the caller is needed for the argument values in the calls to `gcd` (unlike the constant 3 in the FORTRAN77 environment of Figure 7.2), since the C language uses value parameters. This point will be discussed in more detail in Section 7.5. §

Example 7.3 Consider the C code of Figure 7.5. This code contains variables that will be used to illustrate further points later in this section, but its basic operation is as follows. The

Figure 7.5
C program of Example 7.3

```
int x = 2;

void g(int); /* prototype */

void f(int n)
{ static int x = 1;
  g(n);
  x--;
}

void g(int m)
{ int y = m-1;
  if (y > 0)
  { f(y);
    x--;
    g(y);
  }
}


main()
{ g(x);
  return 0;
}
```

first call from `main` is to `g(2)` (since `x` has the value 2 at that point). In this call, `m` becomes 2, and `y` becomes 1. Then, `g` makes the call `f(1)`, and `f` in turn makes the call `g(1)`. In this call to `g`, `m` becomes 1, and `y` becomes 0, so no further calls are made. The runtime environment at this point (during the second call to `g`) is shown in Figure 7.6(a) (page 356).

Now the calls to `g` and `f` exit (with `f` decrementing its static local variable `x` before returning), their activation records are popped from the stack, and control is returned to the point directly following the call to `f` in the first call to `g`. Now `g` decrements the external variable `x` and makes a further call `g(1)`, which sets `m` to 2 and `y` to 1, resulting in the runtime environment shown in Figure 7.6(b). After that no further calls are made, the remaining activation records are popped from the stack, and the program exits.

Note how, in Figure 7.6(b), the activation record of the third call to `g` occupies (and overwrites) the area of memory previously occupied by the activation record of `f`. Note, also, that the static variable `x` in `f` cannot be allocated in an activation record of `f`, since it must persist across all calls to `f`. Thus, it must be allocated in the global/static area along with the external variable `x`, even though it is not a global variable. There can in fact be no confusion with the external `x`, since the symbol table will always distinguish them and determine the correct variable to access at each point in the program.

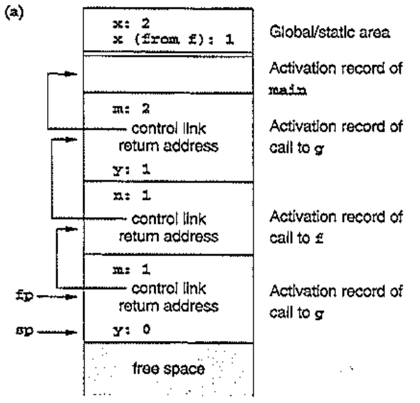
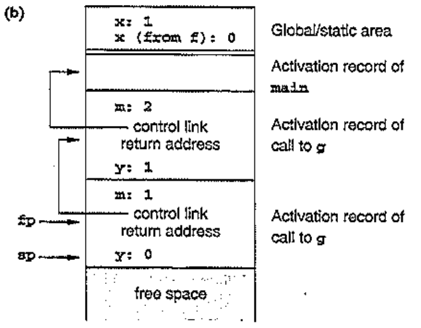


Figure 7.6
(a) Runtime environment of the program of Figure 7.5 during the second call to g



(b) Runtime environment of the program of Figure 7.5 during the third call to g

§

A useful tool for the analysis of complex calling structures in a program is the **activation tree**: each activation record (or call) becomes a node in this tree, and the descendants of each node represent all the calls made during the call corresponding to that node. For example, the activation tree of the program of Figure 7.3 is linear and is depicted (for the inputs 15 and 10) in Figure 7.7(a), while the activation tree of the program of Figure 7.5 is depicted in Figure 7.7(b). Note that the environments shown in Figures 7.4 and 7.6 represent the environments during the calls represented by each of the leaves of the activation trees. In general, the stack of activation records at the beginning of a particular call has a structure equivalent to the path from the corresponding node in the activation tree to the root.

Access to Names In a stack-based environment, parameters and local variables can no longer be accessed by fixed addresses as in a fully static environment. Instead, they

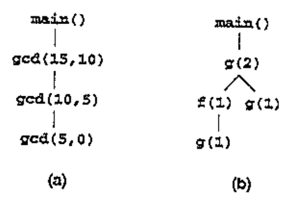
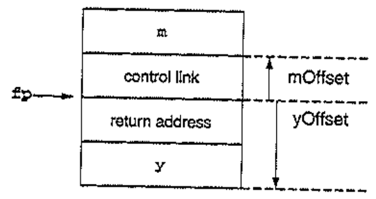


Figure 7.7
Activation trees for the programs of Figures 7.3 and 7.5

must be found by offset from the current frame pointer. In most languages, the offset for each local declaration is still statically computable by the compiler, since the declarations of a procedure are fixed at compile time and the memory size to be allocated for each declaration is fixed by its data type.

Consider the procedure `g` in the C program of Figure 7.5 (see also the runtime environments pictured in Figure 7.6). Each activation record of `g` has exactly the same form, and the parameter `m` and the local variable `y` are always in exactly the same relative location in the activation record. Let us call these distances **`mOffset`** and **`yOffset`**. Then, during any call to `g`, we have the following picture of the local environment



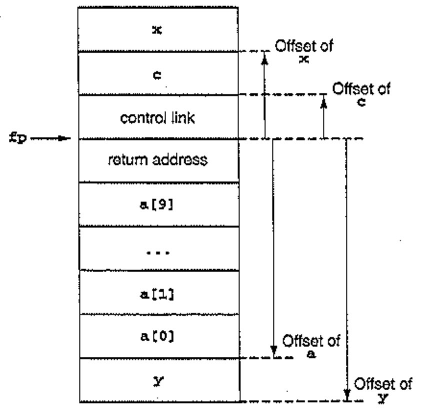
Both `m` and `y` can be accessed by their fixed offsets from the fp. For instance, assume for concreteness that the runtime stack grows from higher to lower memory addresses, that integer variables require 2 bytes of storage, and that addresses require 4 bytes. With the organization of an activation record as shown, we have **`mOffset`** = +4 and **`yOffset`** = −6, and references to `m` and `y` can be written in machine code (assuming standard assembler conventions) as **`4(fp)`** and **`-6(fp)`**, respectively.

Local arrays and structures are no more difficult to allocate and compute addresses for than are simple variables, as the following example demonstrates.

Example 7.4 Consider the C procedure

```
void f(int x, char c)
{ int a[10];
  double y;
  ...
}
```

The activation record for a call to `f` would appear as



and, assuming two bytes for integers, four bytes for addresses, one byte for characters, and eight bytes for double-precision floating point, we would have the following offset values (again assuming a negative direction of growth for the stack), which are all computable at compile time:

Name	Offset
`x`	+5
`c`	+4
`a`	−24
`y`	−32

Now an access of, say, `a[i]` would require the computation of the address

```
(-24+2*i)(fp)
```

(here the factor of 2 in the product `2*i` is the **scale factor** resulting from the assumption that integer values occupy two bytes). Such a memory access, depending on the location of `i` and the architecture, might only need a single instruction. §

Nonlocal and static names in this environment cannot be accessed in the same way that local names are. Indeed, in the case we are considering here—languages with no local procedures—all nonlocals are global and hence static. Thus, in Figure 7.6, the

external (global) C variable `x` has a fixed static location, and so can be accessed directly (or by offset from some base pointer other than the fp). The static local variable `x` from `f` is accessed in exactly the same fashion. Note that this mechanism implements static (or lexical) scope, as described in the previous chapter. If dynamic scope is desired, then a more complex accessing mechanism is required (described later in this section).

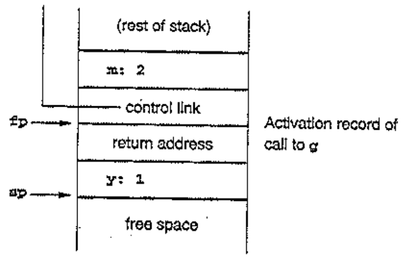
The Calling Sequence The calling sequence now comprises approximately the following steps.[8] When a procedure is called,

1. Compute the arguments and store them in their correct positions in the new activation record of the procedure (pushing them in order onto the runtime stack will achieve this).
2. Store (push) the fp as the control link in the new activation record.
3. Change the fp so that it points to the beginning of the new activation record (if there is an sp, copying the sp into the fp at this point will achieve this).
4. Store the return address in the new activation record (if necessary).
5. Perform a jump to the code of the procedure to be called.

When a procedure exits,

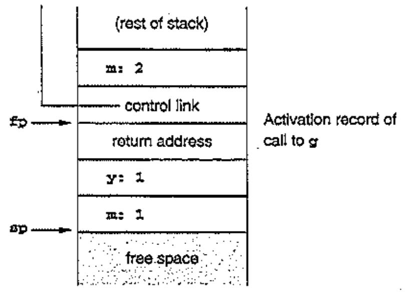
1. Copy the fp to the sp.
2. Load the control link into the fp.
3. Perform a jump to the return address.
4. Change the sp to pop the arguments.

Example 7.5 Consider the situation just before the last call to `g` in Figure 7.6(b):

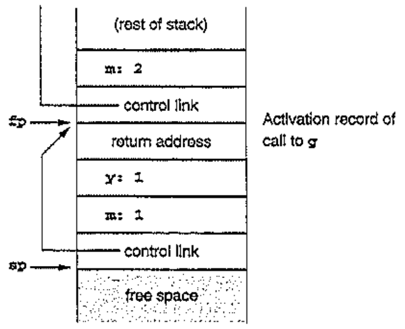


As the new call to `g` is made, first the value of parameter `m` is pushed onto the runtime stack:

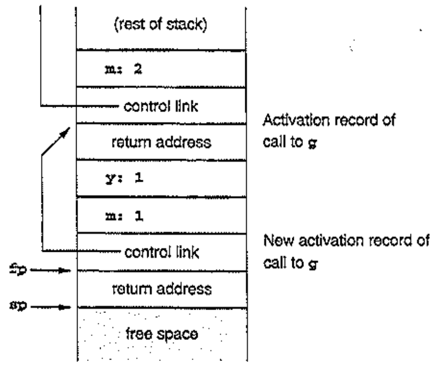
8. This description ignores any saving of registers that must take place. It also ignores the need to place a return value into an available location.



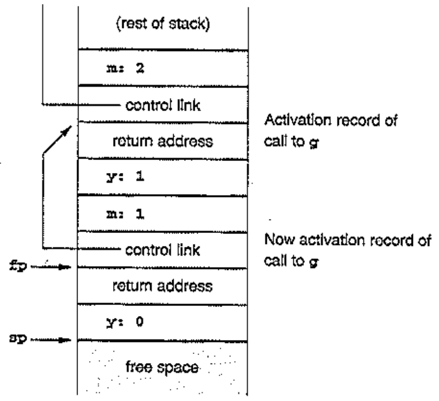
Then the fp is pushed onto the stack:



Now the sp is copied into the fp, the return address is pushed onto the stack, and the jump to the new call of g is made:



Finally, g allocates and initializes the new y on the stack to complete the construction of the new activation record:



§

Dealing with Variable-Length Data So far we have described a situation in which all data, whether local or global, can be found in a fixed place or at a fixed offset from the fp that can be computed by the compiler. Sometimes a compiler must deal with the possibility that data may vary, both in the number of data objects and in the size of each object. Two examples that occur in languages that support stack-based environments are (1) the number of arguments in a call may vary from call to call, and (2) the size of an array parameter or a local array variable may vary from call to call.

A typical example of situation 1 is the `printf` function in C, where the number of arguments is determined from the format string that is passed as the first argument. Thus,

```
printf("%d%s%c",n,prompt,ch);
```

has four arguments (including the format string `"%d%s%c"`), while

```
printf("Hello, world\n");
```

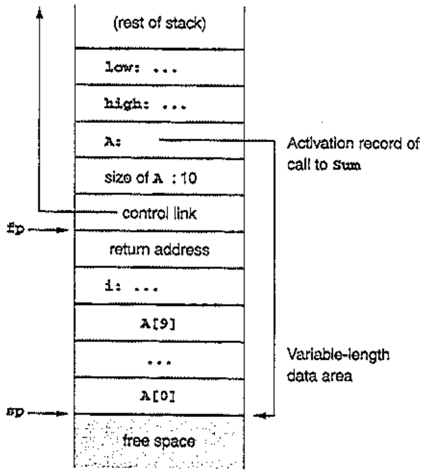
has only one argument. C compilers typically deal with this by pushing the arguments to a call **in reverse order** onto the runtime stack. Then, the first parameter (which tells the code for `printf` how many more parameters there are) is always located at a fixed offset from the fp in the implementation described above (indeed +4, using the assumptions of the previous example). Another option is to use a processor mechanism such as the ap (argument pointer) in VAX architectures. This and other possibilities are treated further in the exercises.

An example of situation 2 is the **unconstrained array** of Ada:

```
type Int_Vector is
      array(INTEGER range <>) of INTEGER;

procedure Sum (low,high: INTEGER;
                  A: Int_Vector) return INTEGER
is
   temp: Int_Array (low..high);
begin
  ...
end Sum;
```

(Note the local variable `temp` which also has unpredictable size.) A typical method for dealing with this situation is to use an extra level of indirection for the variable-length data, storing a pointer to the actual data in a location that can be predicted at compile time, while performing the actual allocation at the top of the runtime stack in a way that can be managed by the sp during execution.

Example 7.6 Given the Ada `Sum` procedure as defined above, and assuming the same organization for the environment as before,[9] we could implement an activation record for `Sum` as follows (this picture shows, for concreteness, a call to `Sum` with an array of size 10):



Now, for instance, access to `A[i]` can be achieved by computing

```
@6 (fp) +2*i
```

9. This is actually not sufficient for Ada, which allows nested procedures; see the discussion later in this section.

where the @ means indirection, and where we are again assuming two bytes for integers and four bytes for addresses.

§

Note that in the implementation described in the previous example, the caller must know the size of any activation record of `Sum`. The size of the parameter part and the bookkeeping part is known to the compiler at the point of call (since the arguments sizes can be counted, and the bookkeeping part is the same for all procedures), but the size of the local variable part is not, in general, known at the point of call. Thus, this implementation requires that the compiler precompute a local-variable size attribute for each procedure and store it in the symbol table for this later use. Variable-length local variables can be dealt with in a similar way.

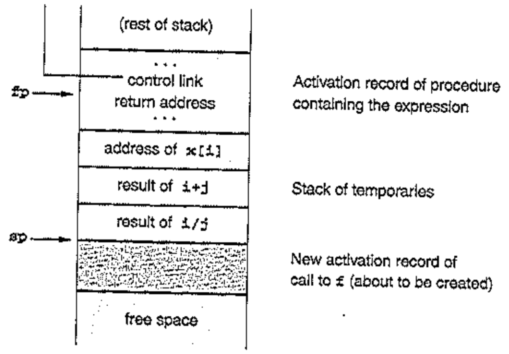
It is worth remarking that C arrays do not fall into the class of such variable-length data. Indeed, C arrays are pointers, so array parameters are passed by reference in C and not allocated locally (and they carry no size information).

Local Temporaries and Nested Declarations There are two more complications to the basic stack-based runtime environment that deserve mention: local temporaries and nested declarations.

Local temporaries are partial results of computations that must be saved across procedure calls. Consider, for example, the C expression

```
x[i] = (i + j)*(i/k + f(j))
```

In a left-to-right evaluation of this expression, three partial results need to be saved across the call to `f`: the address of `x[i]` (for the pending assignment), the sum `i+j` (pending the multiplication), and the quotient `i/k` (pending the sum with the result of the call `f(j)`). These partial results could be computed into registers and saved and restored according to some register management mechanism, or they could be stored as temporaries on the runtime stack prior to the call to `f`. In this latter case, the runtime stack might appear as follows at the point just before the call to `f`:



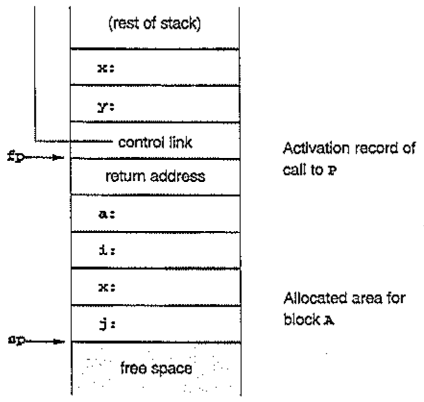
In this situation, the previously described calling sequence using the sp works without change. Alternatively, the compiler can also easily compute the position of the stack top from the fp (in the absence of variable-length data), since the number of required temporaries is a compile-time quantity.

Nested declarations present a similar problem. Consider the C code

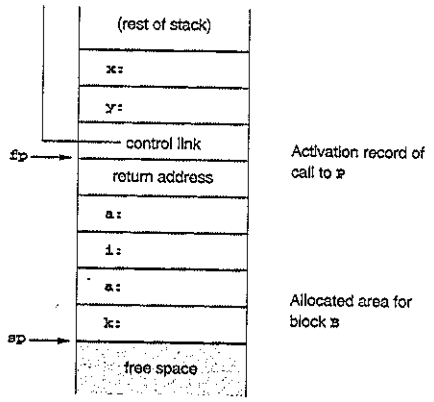
```
void p( int x, double y)
{ char a;
  int i;
  ...
A:{ double x;
    int j;
    ...
  }
  ...
B:{ char * a;
    int k;
    ...
  }
  ...
}
```

In this code there are two blocks (also called *compound statements*), labeled A and B, nested inside the body of procedure p, each with two local declarations whose scope extends only over the block in which they are located (that is, up until the next closing bracket). The local declarations of each of these blocks do not need to be allocated until the block is entered, and the declarations of block A and block B do not need to be allocated simultaneously. A compiler *could* treat a block just like a procedure and create a new activation record each time a block is entered and discard it on exit. However, this would be inefficient, since such blocks are much simpler than procedures: such a block has no parameters and no return address and is always executed immediately, rather than called from elsewhere. A simpler method is to treat declarations in nested blocks in a similar way to temporary expressions, allocating them on the stack as the block is entered and deallocating them on exit.

For instance, just after entering block A in the sample C code just given, the runtime stack would appear as follows:



and just after entry to block B it would look as follows:



Such an implementation must be careful to allocate nested declarations in such a way that the offsets from the fp of the surrounding procedure block are computable at compile time. In particular, such data must be allocated before any variable-length data. For example, in the code just given, the variable j local to block A would have offset -17 from the fp of p (assuming again 2 bytes for integers, 4 bytes for addresses, 8 bytes for floating-point reals, and 1 byte for characters), while k in block B would have offset -13.

7.3.2 Stack-Based Environments with Local Procedures

If local procedure declarations are permitted in the language being compiled, then the runtime environment we have described so far is insufficient, since no provision has been made for nonlocal, nonglobal references.

Consider, for example, the Pascal code of Figure 7.8, page 366 (similar programs could be written in Ada). During the call to q the runtime environment would appear as in Figure 7.9. Using the standard static scoping rule, any mention of n inside q must refer to the local integer variable n of p. As we can see from Figure 7.9, this n cannot be found using any of the bookkeeping information that is kept in the runtime environment up to now.

It *would* be possible to find n using the control links, if we are willing to accept dynamic scoping. Looking at Figure 7.9, we see that the n in the activation record of r could be found by following the control link, and if r had no declaration of n, then the n of p could be found by following a second control link (this process is called **chaining**, a method we will see again shortly). Unfortunately, not only does this implement dynamic scope, but the offsets at which n can be found may vary with different calls (note that the n in r has a different offset from the n in p). Thus, in such an implementation, local symbol tables for each procedure must be kept during execution to allow an identifier to be looked up in each activation record, to see if it exists, and to determine its offset. This is a major additional complication to the runtime environment.

Figure 7.8
Pascal program showing nonlocal, nonglobal reference

```
program nonLocalRef;

procedure p;
var n: integer;

    procedure q;
    begin
      (* a reference to n is now
        non-local non-global *)
    end; (* q *)

    procedure r(n: integer);
    begin
      q;
    end; (* r *)

begin (* p *)
  n := 1;
  r(2);
end; (* p *)

begin (* main *)
  p;
end.
```

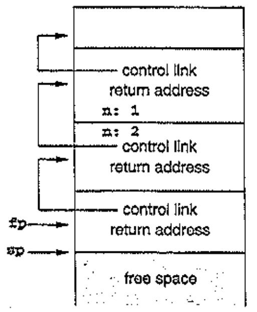
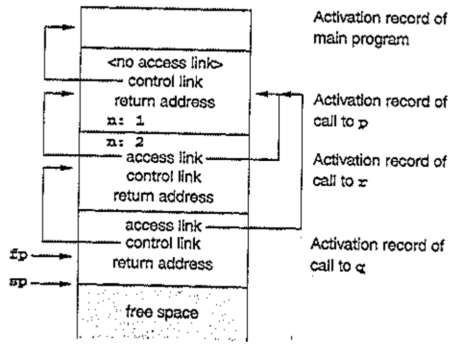

Activation record of main program
control link
return address
n: 1
Activation record of call to p
n: 2
control link
return address
Activation record of call to r
control link
return address
Activation record of call to q
fp
sp
free space

Figure 7.9
Runtime stack for the program of Figure 7.8

The solution to this problem, which also implements static scoping, is to add an extra piece of bookkeeping information called the **access link** to each activation record. The access link is like the control link, except that it points to the activation record that represents the *defining environment* of the procedure instead of the calling environment. For this reason, the access link is sometimes also called the **static link**, even though it is not a compile-time quantity.[10]

Figure 7.10 shows the runtime stack of Figure 7.9 modified to include access links. In this new environment, the access links of the activation records of both r and q point to the activation record of p, since r and q are both declared within p. Now a nonlocal reference to n inside q will cause the access link to be followed, where n will be found at a fixed offset, since this will always be an activation record of p. Typically, this can be achieved in code by loading the access link into a register and then accessing n by offset from this register (which now functions as the fp). For instance, using the size conventions described earlier, if register r is used for the access link, then n inside p can be accessed as −6(r) after r has been loaded with the value 4(fp) (the access link has offset +4 from the fp in Figure 7.10).

Figure 7.10
Runtime stack for the program of Figure 7.8 with access links added



Note that the activation record of procedure p itself contains no access link, as indicated by the bracketed comment in the location where it would go. This is because p is a global procedure, so any nonlocal reference within p must be a global reference and is accessed via the global reference mechanism. Thus, there is no need for an access link. (In fact, a null or otherwise arbitrary access link may be inserted simply for consistency with the other procedures.)

The case we have been describing is actually the simplest situation, where the nonlocal reference is to a declaration in the next outermost scope. It is also possible that nonlocal references refer to declarations in more distant scopes. Consider, for example, the code in Figure 7.11.

10. The defining procedure is of course known, but not the exact location of its activation record.

Figure 7.11
Pascal code demonstrating access chaining

```
program chain;

procedure p;
var x: integer;

   procedure q;
      procedure r;
      begin
        x := 2;
        ...
        if ... then p;
      end; (* r *)
   begin
     r;
   end; (* q *)

begin
  q;
end; (* p *)

begin (* main *)
  p;
end.
```

In this code, procedure `r` is declared in procedure `q`, which in turn is declared in procedure `p`. Thus, the assignment to `x` inside `r`, which refers to the `x` of `p`, must traverse two scope levels to find `x`. Figure 7.12 shows the runtime stack after the (first) call to `r` (there may be more than one call to `r`, since `r` may call `p` recursively). In this environment, `x` must be reached by following *two* access links, a process that is called **access chaining.** Access chaining is implemented by repeatedly fetching the access link, using the previously fetched link as if it were the fp. As a concrete example, `x` in Figure 7.12 can be accessed (using the previous size conventions) as follows:

Load 4(fp) into register r.

Load 4(r) into register r.

Now access `x` as −6(r).

For the method of access chaining to work, the compiler must be able to determine how many nesting levels to chain through before accessing the name locally. This requires the compiler to precompute a **nesting level** attribute for each declaration. Usually, the outermost scope (the main program level in Pascal or the external scope of C) is given nesting level 0, and each time a function or procedure is entered (during compilation), the nesting level is increased by 1, and decreased by the same amount on exit. For example, in the code of Figure 7.11, procedure `p` is given nesting level 0 since it is global; variable `x` is given nesting level 1, since the nesting level is increased when `p` is entered; procedure `q` is also given nesting level 1, since it is local to `p`; and procedure `r` is given nesting level 2, since the nesting level is again increased when `q` is entered. Finally, inside `r` the nesting level is again increased to 3.

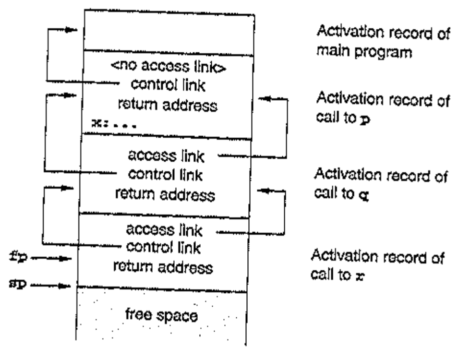


Figure 7.12
Runtime stack after the first call to `r` in the code of Figure 7.11

Now the amount of chaining necessary to access a nonlocal name can be determined by comparing the nesting level at the point of access with the nesting level of the declaration of the name; the number of access links to follow is the difference between these two nesting levels. For example, in the previous situation, the assignment to `x` occurs at nesting level 3, and `x` has nesting level 1, so two access links must be followed. In general, if the difference in nesting levels is m, then the code that must be generated for access chaining must have m loads to a register r, the first using the fp, and the remainder using r.

It may seem that access chaining is an inefficient method for variable access, since a lengthy sequence of instructions must be executed for each nonlocal reference with a large nesting difference. In practice, however, nesting levels are rarely more than two or three deep, and most nonlocal references are to global variables (nesting level 0), which can continue to be accessed by the direct methods previously discussed. There is a method of implementing access links in a lookup table indexed by nesting level that does not carry the execution overhead of chaining. The data structure used for this method is called the **display**. Its structure and use are treated in the exercises.

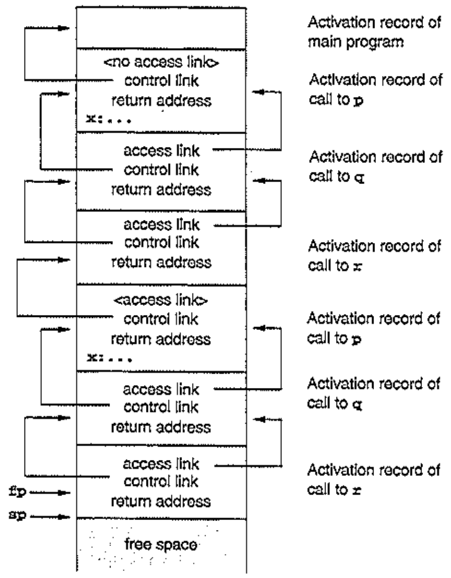
The Calling Sequence The changes to the calling sequence needed to implement access links are relatively straightforward. In the implementation shown, during a call, the access link must be pushed onto the runtime stack just before the fp, and after an exit, the sp must be adjusted by an extra amount to remove the access link as well as the arguments.

The only problem is that of finding the access link of a procedure during a call. This can be achieved by using the (compile-time) nesting level information attached to the declaration of the procedure being called. Indeed, all we need to do is to generate an access chain, just as if we were going to access a variable at the same nesting level as that of the procedure being called. The address so computed will be the appropriate access link. Of course, if the procedure is local (the difference in nesting levels is 0), then the access link and the control link are the same (and are equal to the fp at the point of the call).

Consider, for example the call to q from within r in Figure 7.8. Inside r, we are at nesting level 2, while the declaration of q carries a nesting level of 1 (since q is local to p and inside p the nesting level is 1). Thus, one access step is required to compute the access link of q, and indeed in Figure 7.10, the access link of q points to the activation record of p (and is the same as the access link of r).

Note that even in the presence of multiple activations of the defining environment, this process will compute the correct access link, since the computation is performed at runtime (using the compile-time nesting levels), not at compile time. For example, given the code of Figure 7.11, the runtime stack after the *second* call to r (assuming a recursive call to p) would look as in Figure 7.13. In this picture, r has two different activation records with two different access links, pointing at the different activation records of q, which represent different defining environments for r.

Figure 7.13
Runtime stack after the second call to r in the code of Figure 7.11



7.3.3 Stack-Based Environments with Procedure Parameters

In some languages, not only are local procedures allowed, but procedures may also be passed as parameters. In such a language, when a procedure that has been passed as a parameter is called, it is impossible for a compiler to generate code to compute the access link at the time of call, as described in the previous section. Instead, the access link for a procedure must be precomputed and passed along with a pointer to the code for the procedure when the procedure is passed as a parameter. Thus, a procedure parameter value can no longer be viewed as a simple code pointer, but must also include an access pointer that defines the environment in which nonlocal references are resolved. This pair of pointers—a code pointer and an access link or an **instruction pointer** and an **environment pointer**—together represent the value of a procedure or function parameter and are commonly called a **closure** (since the access link "closes" the "holes" caused by nonlocal references).[11] We will write closures as <ip, ep>, where ip refers to the instruction pointer (code pointer or entry point) of the procedure, and ep refers to the environment pointer (access link) of the procedure.

Example 7.7 Consider the Standard Pascal program of Figure 7.14, which has a procedure p, with a parameter a that is also a procedure. After the call to p in q, in which the local procedure r of q is passed to p, the call to a inside p actually calls r, and this call must still find the nonlocal variable x in the activation of q. When p is called, a is constructed as a closure <ip, ep>, where ip is a pointer to the code of r and ep is a copy of the fp at the point of call (that is, it points to the environment of the call to q in which r is defined). The value of the ep of a is indicated by the dashed line in Figure 7.15 (page 372), which represents the environment just after the call to p in q. Then, when a is called inside p, the ep of a is used as the static link in its activation record, as indicated in Figure 7.16.

§

Figure 7.14
Standard Pascal code with a procedure as parameter

```
program closureEx(output);

procedure p(procedure a);
begin
  a;
end;

procedure q;
var x:integer;

    procedure r;
    begin
      writeln(x);
    end;

begin
  x := 2;
  p(r);
end; (* q *)

begin (* main *)
  q;
end.
```

§

11. This term has its origin in lambda calculus and is not to be confused with the (Kleene) closure operation in regular expressions or the ε-closure of a set of NFA states.

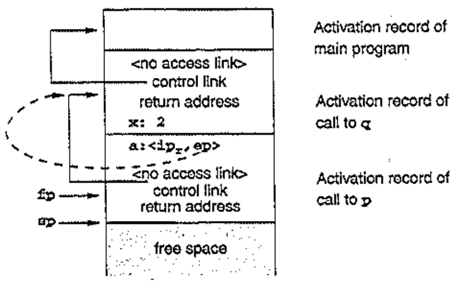


Figure 7.15
Runtime stack just after the call to p in the code of Figure 7.14

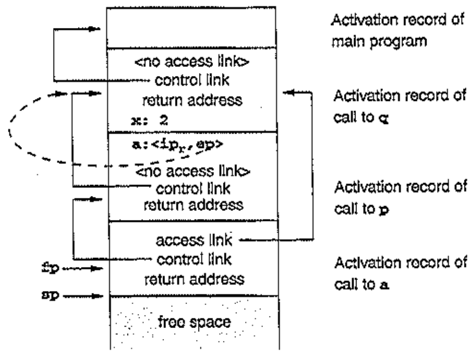


Figure 7.16
Runtime stack just after the call to a in the code of Figure 7.14

The calling sequence in an environment such as we have just described must now distinguish clearly between ordinary procedures and procedure parameters. An ordinary procedure is called, as before, by fetching the access link using the nesting level of the procedure and jumping directly to the code of the procedure (which is known at compile time). A procedure parameter, on the other hand, has its access link already available, stored in the local activation record, which must be fetched and inserted into the new activation record. The location of the code for the procedure, however, is not known directly to the compiler; instead, an indirect call must be performed to the ip stored in the current activation record.

A compiler writer may, for reasons of simplicity or uniformity, wish to avoid this distinction between ordinary procedures and procedure parameters and keep all procedures as closures in the environment. Indeed, the more general a language is in its treatment of procedures, the more reasonable such an approach becomes. For example, if procedure variables are allowed, or if procedure values can be dynamically computed, then the <ip, ep> representation of procedures becomes a requirement for all such situations. Figure 7.17 shows what the environment of Figure 7.16 would look like if all procedure values are stored in the environment as closures.

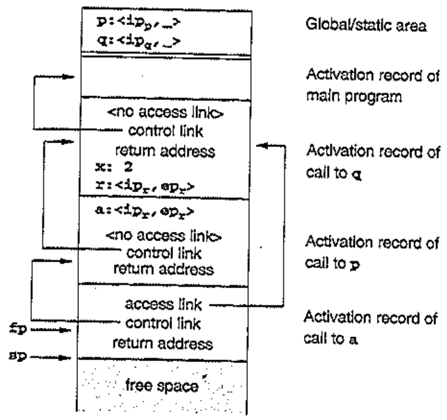


Figure 7.17
Runtime stack just after the call to a in the code of Figure 7.14 with all procedures kept as closures in the environment

Finally, we note that C, Modula-2, and Ada all avoid the complications described in this subsection: C, because it has no local procedures (even though it has procedure parameters and variables); Modula-2, because of a special rule restricting procedure parameter and procedure variable values to global procedures; and Ada, because it has no procedure parameters or variables.

7.4 DYNAMIC MEMORY

7.4.1 Fully Dynamic Runtime Environments

The stack based runtime environments discussed in the previous section are the most common forms of environment among the standard imperative languages such as C, Pascal, and Ada. Such environments do have limitations, however. In particular, in a language where a reference to a local variable in a procedure can be returned to the caller, either implicitly or explicitly, a stack-based environment will result in a **dangling reference** when the procedure is exited, since the activation record of the procedure will be deallocated from the stack. The simplest example of this is when the address of a local variable is returned, as for instance in the C code:

```
int * dangle(void)
{ int x;
  return &x;}
```

An assignment `addr = dangle()` now causes `addr` to point to an unsafe location in the activation stack whose value can be arbitrarily changed by subsequent calls to any procedure. C gets around this problem by simply declaring such a program to be erroneous (although no compiler will give an error message). In other words, the semantics of C are built around the underlying stack-based environment.

A somewhat more complex instance of a dangling reference occurs if a local function can be returned by a call. For instance, if C were to allow local function definitions, the code of Figure 7.18 would result in an indirect dangling reference to the parameter `x` of `g`, which can be accessed by calling `f` after `g` has exited. C, of course, avoids this problem by prohibiting local procedures. Other languages, like Modula-2, which have local procedures as well as procedure variables, parameters, and returned values, must state a special rule that makes such programs erroneous. (In Modula-2 the rule is that only global procedures can be arguments or returned values—a major retreat even from Pascal-style procedure parameters.)

Figure 7.18
Pseudo-C code showing a dangling reference caused by the return of a local function

```
typedef int (* proc)(void);

proc g(int x)
{ int f(void) /* illegal local function */
  { return x;}
  return f; }

main()
{ proc c;
  c = g(2);
  printf("%d\n",c()); /* should print 2 */
  return 0;
}
```

There is a large class of languages, however, where such rules are unacceptable, that is, the functional programming languages, such as LISP and ML. An essential principle in the design of a functional language is that functions be as general as possible, and this means that functions must be able to be locally defined, passed as parameters, and returned as results. Thus, for this large class of languages, a stack-based runtime environment is inadequate, and a more general form of environment is required. We call such an environment **fully dynamic**, because it can deallocate activation records only when all references to them have disappeared, and this requires that activation records be dynamically freed at arbitrary times during execution. A fully dynamic runtime environment is significantly more complicated than a stack-based environment, since it involves the tracking of references during execution, and the ability to find and deallocate inaccessible areas of memory at arbitrary times during execution (this process is called **garbage collection**).

Despite the added complexity of this kind of environment, the basic structure of an activation record remains the same: space must be allocated for parameters and local variables, and there is still a need for the control and access links. Of course, now when control is returned to the caller (and the control link is used to restore the previous environment), the exited activation record remains in memory, to be deallocated at some later time. Thus, the entire additional complexity of this environment can be encapsulated in a memory manager that replaces the runtime stack operations with more general allocation and deallocation routines. We discuss some of the issues in the design of such a memory manager later in this section.

7.4.2 Dynamic Memory in Object-Oriented Languages

Object-oriented languages require special mechanisms in the runtime environment to implement their added features: objects, methods, inheritance, and dynamic binding. In this subsection we give a brief overview of the variety of implementation techniques for these features. We assume the reader is familiar with basic object-oriented terminology and concepts.[12]

Object-oriented languages vary greatly in their requirements for the runtime environment. Smalltalk and C++ are good representatives of the extremes: Smalltalk requires a fully dynamic environment similar to that of LISP, while much of the design effort in C++ has gone into retaining the stack-based environment of C, without the need for automatic dynamic memory management. In both these languages, an object in memory can be viewed as a cross between a traditional record structure and an activation record, with the instance variables (data members) as the fields of the record. This structure differs from a traditional record in how methods and inherited features are accessed.

One straightforward mechanism for implementing objects would be for initialization code to copy all the currently inherited features (and methods) directly into the record structure (with methods as code pointers). This is extremely wasteful of space, however. An alternative is to keep a complete description of the class structure in memory at each point during execution, with inheritance maintained by superclass pointers, and all method pointers kept as fields in the class structure (this is sometimes called an **inheritance graph**). Each object then keeps, along with fields for its instance variables, a pointer to its defining class, through which all methods (both local and inherited) are found. In this way, method pointers are recorded only once (in the class structure) and not copied in memory for each object. This mechanism also implements inheritance and dynamic binding, since methods are found by a search of the class hierarchy. The disadvantage is that, while instance variables may have predictable offsets (just as local variables in a standard environment), the methods do not, and they must be maintained by name in a symbol table structure with lookup capabilities. Nevertheless, this is a reasonable structure for a highly dynamic language like Smalltalk, where changes to the class structure can occur during execution.

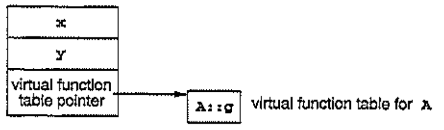
An alternative to keeping the entire class structure within the environment is to compute the list of code pointers for available methods of each class, and store this in (static) memory as a **virtual function table** (in C++ terminology). This has the advantage that it can be arranged so that each method has a predictable offset, and a traversal of the class hierarchy with a series of table lookups is no longer necessary. Now each object contains a pointer to the appropriate virtual function table, rather than to the class structure. (Of course, the location of this pointer must also have predictable offset.) This simplification only works if the class structure itself is fixed prior to execution. It is the method of choice in C++.

12. The following discussion also assumes only single inheritance is available. Multiple inheritance is treated in some of the works cited in the Notes and References section.

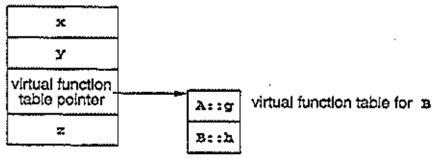
Example 7.8 Consider the following C++ class declarations:

```
class A
{ public:
  double x,y;
  void f();
  virtual void g();
};

class B: public A
{ public:
  double z;
  void f();
  virtual void h();
};
```

an object of class `A` would appear in memory (with its virtual function table) as follows:



while an object of class `B` would appear as follows:



Note how the virtual function pointer, once added to the object structure, remains in a fixed location, so that its offset is known prior to execution. Note also that the function `f` does not obey dynamic binding in C++ (since it is not declared "virtual"), and so does not appear in the virtual function table (or anywhere else in the environment); a call to `f` is resolved at compile time. §

7.4.3 Heap Management

In Section 7.4.1, we discussed the need for a runtime environment that is more dynamic than the stack-based environment used in most compiled languages, if general functions are to be fully supported. In most languages, however, even a stack-based environment needs some dynamic capabilities in order to handle pointer allocation and deallocation. The data structure that handles such allocation is called a *heap*, and the heap is usually

allocated as a linear block of memory in such a way that it can grow, if necessary, while interfering as little as possible with the stack. (On page 347 we showed the heap sitting in a block of memory at the opposite end of the stack area.)

So far in this chapter, we have concentrated on the organization of activation records and the runtime stack. In this section, we want to describe how the heap can be managed, and how the heap operations might be extended to provide the kind of dynamic allocation required in languages with general function capabilities.

A heap provides two operations, *allocate* and *free*. The *allocate* operation takes a size parameter (either explicitly or implicitly), usually in bytes, and returns a pointer to a block of memory of the correct size, or a null pointer if none exists. The *free* operation takes a pointer to an allocated block of memory and marks it as being free again. (The *free* operation must also be able to discover the size of the block to be freed, either implicitly or by an explicit parameter.) These two operations exist under different names in many languages: they are called **`new`** and **`dispose`** in Pascal and **`new`** and **`delete`** in C++. The C language has several versions of these operations, but the basic ones are called **`malloc`** and **`free`** and are part of the standard library (**`stdlib.h`**), where they have essentially the following declarations:

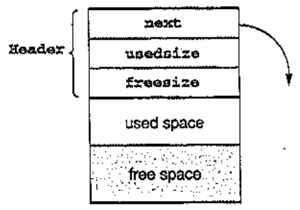
```
void * malloc (unsigned nbytes);
void free (void * ap);
```

We will use these declarations as the basis for our description of heap management.

A standard method for maintaining the heap and implementing these functions is to use a circular linked list of free blocks, from which memory is taken by **`malloc`** and returned to by **`free`**. This has the advantage of simplicity, but it also has drawbacks. One is that the **`free`** operation cannot tell whether its pointer argument is really pointing at a legitimate block that was previously allocated by **`malloc`**. If the user should pass an invalid pointer, then the heap can become easily and quickly corrupted. A second, but much less serious, drawback is that care must be taken to **coalesce** blocks that are returned to the free list with blocks that are adjacent to it, so that a free block of maximal size will result. Without coalescing, the heap can quickly become **fragmented**, that is, divided into a large number of small-sized blocks, so that the allocation of a large block may fail, even though there is enough total space available to allocate it. (Fragmentation is of course possible even with coalescing.)

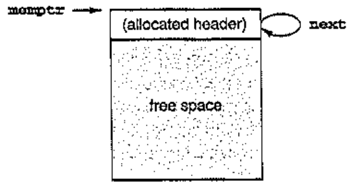
We offer here a slightly different implementation of **`malloc`** and **`free`** that uses a circular linked list data structure that keeps track of both allocated and free blocks (and thus is less susceptible to corruption) and that also has the advantage of providing self-coalescing blocks. The code is given in Figure 7.19 (page 379).

This code uses a statically allocated array of size **`MEMSIZE`** as the heap, but an operating system call could also be used to allocate the heap. We define a data type **`Header`** that will hold each memory block's bookkeeping information, and we define the heap array to have elements of type **`Header`** so that the bookkeeping information can be easily kept in the memory blocks themselves. The type **`Header`** contains three pieces of information: a pointer to the next block in the list, the size of the currently allocated space (which comes next in memory), and the size of any following free space (if there is any). Thus, each block in the list is of the form



The definition of type **Header** in Figure 7.19 also uses a **union** declaration and an **Align** data type (which we have set to **double** in the code). This is to align the memory elements on a reasonable byte boundary and, depending on the system, may or may not be necessary. This complication can be safely ignored in the remainder of this description.

The one additional piece of data needed by the heap operations is a pointer to one of the blocks in the circular linked list. This pointer is called **memptr**, and it always points to a block that has some free space (usually the last space to be allocated or freed). It is initialized to **NULL**, but on the first call to **malloc**, initialization code is executed that sets **memptr** to the beginning of the heap array and initializes the header in the array, as follows:



This initial header that is allocated on the first call to **malloc** will never be freed. There is now one block in the list, and the remainder of the code of **malloc** searches the list and returns a new block from the first block that has enough free space (this is a **first fit** algorithm). Thus, after, say, three calls to **malloc**, the list will look as follows:

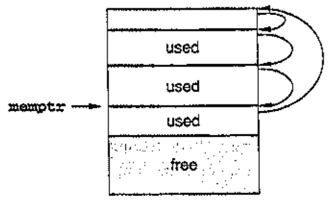


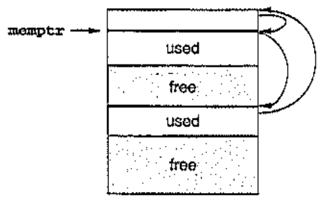
Figure 7.19
C code to maintain a heap of contiguous memory using a list of pointers to both used and free blocks

```
#define NULL 0
#define MEMSIZE 8096 /* change for different sizes */

typedef double Align;
typedef union header
  { struct { union header *next;
             unsigned usedsize;
             unsigned freesize;
           } s;
    Align a;
  } Header;

static Header mem[MEMSIZE];
static Header *memptr = NULL;

void *malloc(unsigned nbytes)
{ Header *p, *newp;
  unsigned nunits;
  nunits = (nbytes+sizeof(Header)-1)/sizeof(Header) + 1;
  if (memptr == NULL)
  { memptr->s.next = memptr = mem;
    memptr->s.usedsize = 1;
    memptr->s.freesize = MEMSIZE-1;
  }
  for(p=memptr;
      (p->s.next!=memptr) && (p->s.freesize<nunits);
      p=p->s.next);
  if (p->s.freesize < nunits) return NULL;
  /* no block big enough */
  newp = p+p->s.usedsize;
  newp->s.usedsize = nunits;
  newp->s.freesize = p->s.freesize - nunits;
  newp->s.next = p->s.next;
  p->s.freesize = 0;
  p->s.next = newp;
  memptr = newp;
  return (void *) (newp+1);
}

void free(void *ap)
{ Header *bp, *p, *prev;
  bp = (Header *) ap - 1;
  for (prev=memptr,p=memptr->s.next;
       (p!=bp) && (p!=memptr); prev=p,p=p->s.next);
  if (p!=bp) return;
  /* corrupted list, do nothing */
  prev->s.freesize += p->s.usedsize + p->s.freesize;
  prev->s.next = p->s.next;
  memptr = prev;
}
```

Note that as the blocks are allocated in succession, a new block is created each time, and the free space left over from the previous block is carried with it (so that the free space of the block from which the allocation took place always has **freesize** set to 0). Also, **memptr** follows the construction of the new blocks, and so always points to a block with some free space. Note also that **malloc** always increments the pointer to the newly created block, so that the header is protected from being overwritten by the client program (as long as only positive offsets into the returned memory are used).

Now consider the code for the **free** procedure. It first decrements the pointer passed by the user to find the header of the block. It then searches the list for a pointer that is identical to this one, thus protecting the list from becoming corrupted, and also allowing the pointer to the previous block to be computed. When found, the block is removed from the list, and both its used and free space are added to the free space of the previous block, thus automatically coalescing the free space. Note that **memptr** is also set to point to the block containing the memory just freed.

As an example, suppose the middle block of the three used blocks in the previous picture is freed. Then the heap and its associated block list would look as follows:



7.4.4 Automatic Management of the Heap

The use of **malloc** and **free** to perform dynamic allocation and deallocation of pointers is a **manual** method for the management of the heap, since the programmer must write explicit calls to allocate and free memory. By contrast, the runtime stack is managed **automatically** by the calling sequence. In a language that needs a fully dynamic runtime environment, the heap must similarly be managed automatically. Unfortunately, while calls to **malloc** can be easily scheduled at each procedure call, the calls to **free** cannot be similarly scheduled on exit, since activation records must persist until all references to them have disappeared. Thus, automatic memory management involves the reclamation of previously allocated but no longer used storage, possibly long after it was allocated, and without an explicit call to **free**. This process is called **garbage collection**.

Recognizing when a block of storage is no longer referenced, either directly or indirectly through pointers, is a much more difficult task than is the maintenance of a list of blocks of heap storage. The standard technique is to perform **mark and sweep** garbage collection.[13] In this method, no memory is freed until a call to **malloc** fails, at which point a garbage collector is activated that looks for all storage that can be referenced and frees all unreferenced storage. It does this in two passes. The first pass follows all pointers recursively, starting with all currently accessible pointer values, and marks each block of storage reached. This process requires an extra bit of storage for the marking. A second pass then sweeps linearly through memory, returning unmarked blocks to free memory. While this process usually will find enough contiguous free memory to satisfy a series of new requests, it is possible that memory is still so fragmented that a large memory request will still fail, even after garbage collection has been performed. Hence, a garbage collection usually also performs **memory compaction** by moving all the allocated space to one end of the heap, leaving only one large block of contiguous free space at the other end. This process must also update all references to those areas in memory that were moved within the executing program.

13. A simpler alternative called **reference counting** is also occasionally used. See the Notes and References section.

Mark and sweep garbage collection has several drawbacks: it requires extra storage (for the marks), and the double pass through memory causes a significant delay in processing, sometimes as much as a few seconds, each time the garbage collector is invoked—which can be every few minutes. This is clearly unacceptable for many applications involving interactive or immediate response.

A bookkeeping improvement can be made to this process by splitting available memory into two halves and allocating storage only from one half at a time. Then during the marking pass, all reached blocks are immediately copied to the second half of storage not in use. This means that no extra mark bit is required in storage, and only one pass is required. It also performs compaction automatically. Once all reachable blocks in the used area have been copied, the used and unused halves of memory are interchanged, and processing continues. This method is called **stop-and-copy** or **two-space** garbage collection. Unfortunately, it does little to improve processing delays during storage reclamation.

Recently, a method has been invented that reduces this delay significantly. Called **generational garbage collection**, it adds a permanent storage area to the reclamation scheme of the previous paragraph. Allocated objects that survive long enough are simply copied into permanent space and are never deallocated during subsequent storage reclamations. This means that the garbage collector needs to search only a very small section of memory for newer storage allocations, and the time for such a search is reduced to a fraction of a second. Of course, it is possible for permanent memory still to become exhausted with unreachable storage, but this is a much less severe problem than before, since temporary storage tends to disappear quickly, while storage that stays allocated for some time tends to persist anyway. This process has been shown to work very well, especially with a virtual memory system.

We refer the reader to sources listed in the Notes and References section for details on this and other methods of garbage collection.

7.5 PARAMETER PASSING MECHANISMS

We have seen how, in a procedure call, parameters correspond to locations in the activation record, which are filled in with the arguments, or parameters values, by the caller, prior to jumping to the code of the called procedure. Thus, to the code of the called procedure, a parameter represents a purely formal value to which no code is attached, but which serves only to establish a location in the activation record, where the code can find its eventual value, which will only exist once a call has taken place.

The process of building these values is sometimes referred to as the **binding** of the parameters to the arguments. How the argument values are interpreted by the procedure code depends on the particular **parameter passing mechanism**(s) adopted by the source language. As we have already indicated, FORTRAN77 adopts a mechanism that binds parameters to locations rather than values, while C views all arguments as values. Other languages, like C++, Pascal, and Ada, offer a choice of parameter passing mechanisms.

In this section we will discuss the two most common parameter passing mechanisms—**pass by value** and **pass by reference** (sometimes also referred to as call by value and call by reference)—as well as two additional important methods, **pass by value-result** and **pass by name** (also called **delayed evaluation**). Some variations on these will be discussed in the exercises.

One issue not addressed by the parameter passing mechanism itself is the order in which arguments are evaluated. In most situations, this order is unimportant for the execution of a program, and any evaluation order will produce the same results. In that case, for efficiency or other reasons, a compiler may choose to vary the order of argument evaluation. Many languages, however, permit arguments to calls that cause side effects (changes to memory). For example, the C function call

```
f(++x,x);
```

causes a change in the value of `x`, so that different evaluation orders have different results. In such languages, a standard evaluation order such as left to right may be specified, or it may be left to the compiler writer, in which case the result of a call may vary from implementation to implementation. C compilers typically evaluate their arguments from right to left, rather than left to right. This allows for a variable number of arguments (such as in the `printf` function), as discussed in Section 7.3.1, page 361.

7.5.1 Pass by Value

In this mechanism, the arguments are expressions that are evaluated at the time of the call, and their values become the values of the parameters during the execution of the procedure. This is the only parameter passing mechanism available in C and is the default in Pascal and Ada (Ada also allows such parameters to be explicitly specified as `in` parameters).

In its simplest form, this means that value parameters behave as constant values during the execution of a procedure, and one can interpret pass by value as replacing all the parameters in the body of a procedure by the values of the arguments. This form of pass by value is used by Ada, where such parameters cannot be assigned to or otherwise used as local variables. A more relaxed view is taken by C and Pascal, where value parameters are viewed essentially as initialized local variables, which can be used as ordinary variables, but changes to them never cause any nonlocal changes to take place.

In a language like C that offers only pass by value, it is impossible to directly write a procedure that achieves its effect by making changes to its parameters. For example, the following `inc2` function written in C does not achieve its desired effect:

```
void inc2( int x)
/* incorrect! */
{ ++x;++x; }
```

While in theory it is possible, with suitable generality of functions, to perform all computations by returning appropriate values instead of changing parameter values, languages like C usually offer a method of using pass by value in such a way as to achieve nonlocal changes. In C, this takes the form of passing the address instead of the value (and thus changing the data type of the parameter):

```
void inc2( int* x)
/* now ok */
{ ++(*x);++(*x); }
```

Of course, now to increment a variable `y` this function must be called as `inc2(&y)`, since the address of `y` and not its value is required.

This method works especially well in C for arrays, since they are implicitly pointers, and so pass by value allows the individual array elements to be changed:

```
void init(int x[],int size)
/* this works fine when called
   as init(a), where a is an array */
{ int i;
  for(i=0;i<size;++i) x[i]=0;
}
```

Pass by value requires no special effort on the part of the compiler. It is easily implemented by taking the most straightforward view of argument computation and activation record construction.

7.5.2 Pass by Reference

In pass by reference, the arguments must (at least in principle) be variables with allocated locations. Instead of passing the value of a variable, pass by reference passes the location of the variable, so that the parameter becomes an **alias** for the argument, and any changes made to the parameter occur to the argument as well. In FORTRAN77, pass by reference is the only parameter passing mechanism. In Pascal, pass by reference is achieved with the use of the **var** keyword and in C++ by the use of the special symbol `&` in the parameter declaration:

```
void inc2( int & x)
/* C++ reference parameter */
{ ++x;++x; }
```

This function can now be called without a special use of the address operator: `inc2(y)` works fine.

Pass by reference requires that the compiler compute the address of the argument (and it must have such an address), which is then stored in the local activation record.

The compiler must also turn local accesses to a reference parameter into indirect accesses, since the local "value" is actually the address elsewhere in the environment.

In languages like FORTRAN77, where only pass by reference is available, an accommodation is usually offered for arguments that are values without addresses. Rather than making a call like

```
p(2+3)
```

illegal in FORTRAN77, a compiler must instead "invent" an address for the expression **2+3**, compute the value into this address, and then pass the address to the call. Typically, this is done by creating a temporary location in the activation record of the caller (in FORTRAN77, this will be static). An example of this is in Example 7.1 (page 350), where the value 3 is passed as an argument by creating a memory location for it in the activation record of the main procedure.

One aspect of pass by reference is that it does not require a copy to be made of the passed value, unlike pass by value. This can sometimes be significant when the value to be copied is a large structure (or an array in a language other than C or C++). In that case, it may be important to be able to pass an argument by reference, but prohibit changes to be made to the argument's value, thus achieving pass by value without the overhead of copying the value. Such an option is provided by C++, in which one may write a call such as

```
void f( const MuchData & x )
```

where **MuchData** is a data type with a large structure. This is still pass by reference, but the compiler must also perform a static check that **x** never appears on the left of an assignment or may otherwise be changed.[14]

7.5.3 Pass by Value-Result

This mechanism achieves a similar result to pass by reference, except that no actual alias is established: the value of the argument is copied and used in the procedure, and then the final value of the parameter is copied back out to the location of the argument when the procedure exits. Thus, this method is sometimes known as copy-in, copy-out—or copy-restore. This is the mechanism of the Ada **in out** parameter. (Ada also has simply an **out** parameter, which has no initial value passed in; this could be called pass by result.)

Pass by value-result is only distinguishable from pass by reference in the presence of aliasing. For instance, in the following code (in C syntax),

```
void p(int x, int y)
{ ++x;
  ++y;
}
```

14. This cannot always be done in a completely secure way.

```
main()
{ int a = 1;
  p(a,a);
  return 0;
}
```

a has value 3 after p is called if pass by reference is used, while **a** has the value 2 if pass by value-result is used.

Issues left unspecified by this mechanism, and possibly differing in different languages or implementations, are the order in which results are copied back to the arguments and whether the locations of the arguments are calculated only on entry and stored or whether they are recalculated on exit.

Ada has a further quirk: its definition states that **in out** parameters may actually be implemented as pass by reference, and any computation that would be different under the two mechanisms (thus involving an alias) is an error.

From the point of view of the compiler writer, pass by value-result requires several modifications to the basic structure of the runtime stack and the calling sequence. First, the activation record cannot be freed by the callee, since the (local) values to be copied out must be still available to the caller. Second, the caller must either push the addresses of the arguments as temporaries onto the stack before beginning the construction of the new activation record, or it must recompute these addresses on return from the called procedure.

7.5.4 Pass by Name

This is the most complex of the parameter passing mechanisms. It is also called **delayed evaluation**, since the idea of pass by name is that the argument is not evaluated until its actual use (as a parameter) in the called program. Thus, the name of the argument, or its textual representation at the point of call, replaces the name of the parameter it corresponds to. As an example, in the code

```
void p(int x)
{ ++x; }
```

if a call such as **p(a[i])** is made, the effect is of evaluating **++(a[i])**. Thus, if **i** were to change before the use of **x** inside **p**, the result would be different from either pass by reference or pass by value-result. For instance, in the code (in C syntax),

```
int i;
int a[10];

void p(int x)
{ ++i;
  ++x;
}
```

```
main()
{ i = 1;
  a[1] = 1;
  a[2] = 2;
  p(a[i]);
  return 0;
}
```

the result of the call to `p` is that `a[2]` is set to 3 and `a[1]` is left unchanged.

The interpretation of pass by name is as follows. The text of an argument at the point of call is viewed as a function in its own right, which is evaluated every time the corresponding parameter name is reached in the code of the called procedure. However, the argument will always be evaluated in the environment of the caller, while the procedure will be executed in its defining environment.

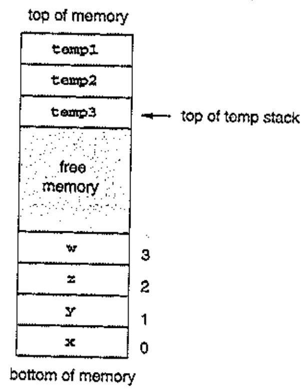
Pass by name was offered as a parameter passing mechanism (along with pass by value) in the language Algol60, but became unpopular for several reasons. First, it can give surprising and counterintuitive results in the presence of side effects (as the previous example shows). Second, it is difficult to implement, since each argument must be turned into what is essentially a procedure (sometimes called a **suspension** or **thunk**) that must be called whenever the argument is evaluated. Third, it is inefficient, since not only does it turn a simple argument evaluation into a procedure call, but it may also cause multiple evaluations to occur. A variation on this mechanism called **lazy evaluation** has recently become popular in purely functional languages, where reevaluation is prevented by **memoizing** the suspension with the computed value the first time it called. Lazy evaluation can actually result in a *more* efficient implementation, since an argument that is never used is also never evaluated. Languages that offer lazy evaluation as a parameter passing mechanism are **Miranda** and **Haskell**. We refer the reader to the Notes and References section for additional information.

7.6 A RUNTIME ENVIRONMENT FOR THE TINY LANGUAGE

In this final section of the chapter, we describe the structure of a runtime environment for the TINY language, our running example of a small, simple language for compilation. We do this in a machine-independent way here and refer the reader to the next chapter for an example of an implementation on a specific machine.

The environment needed by TINY is significantly simpler than any of the environments discussed in this chapter. Indeed, TINY has no procedures, and all of its variables are global, so that there is no need for a stack of activation records, and the only dynamic storage necessary is that for temporaries during expression evaluation (even this could be made static as in FORTRAN77—see the exercises).

One simple scheme for a TINY environment is to place the variables in absolute addresses at the bottom end of program memory, and allocate the temporary stack at the top end. Thus, given a program that used, say, four variables `x`, `y`, `z`, and `w`, these variables would get the absolute addresses 0 through 3 at the bottom of memory, and at a point during execution where three temporaries are being stored, the runtime environment would look as follows:



Depending on the architecture, we may need to set some bookkeeping registers to point to the bottom and/or top of memory, and then use the "absolute" addresses of the variables as offsets from the bottom pointer, and either use the top of memory pointer as the "top of temp stack" pointer or compute offsets for the temporaries from a fixed top pointer. It would, of course, also be possible to use the processor stack as the temporary stack, if it is available.

To implement this runtime environment, the symbol table in the TINY compiler must, as described in the last chapter, maintain the addresses of the variables in memory. It does this by providing a location parameter in the `st_insert` function and the inclusion of a `st_lookup` function that retrieves the location of a variable (Appendix B, lines 1166 and 1171):

```
void st_insert( char * name, int lineno, int loc );
int st_lookup ( char * name );
```

The semantic analyzer must, in its turn, assign addresses to variables as they are encountered the first time. It does this by maintaining a static memory location counter that is initialized to the first address (Appendix B, line 1413):

```
static int location = 0;
```

Then, whenever a variable is encountered (in a read statement, assignment statement, or identifier expression), the semantic analyzer executes the code (Appendix B, line 1454):

```
if (st_lookup(t->attr.name) == -1)
  st_insert(t->attr.name,t->lineno,location++);
else
  st_insert(t->attr.name,t->lineno,0);
```

When `st_lookup` returns −1, the variable is not yet in the table. In that case, a new location is recorded, and the location counter is incremented. Otherwise, the variable is already in the table, in which case the symbol table ignores the location parameter (and we write 0 as a dummy location).

This handles the allocation of the named variables in a TINY program; the allocation of the temporary variables at the top of memory, and the operations needed to maintain this allocation, will be the responsibility of the code generator, discussed in the next chapter.

EXERCISES

7.1 Draw a possible organization for the runtime environment of the following FORTRAN77 program, similar to that of Figure 7.2 (page 352). Be sure to include the memory pointers as they would exist during the call to **AVE**.

```
      REAL A(SIZE),AVE
      INTEGER N,I
   10 READ *, N
      IF (N.LE.0.OR.N.GT.SIZE) GOTO 99
      READ *,(A(I),I=1,N)
      PRINT *, 'AVE = ',AVE(A,N)
      GOTO 10
   99 CONTINUE
      END
      REAL FUNCTION AVE(B,N)
      INTEGER I,N
      REAL B(N),SUM
      SUM = 0.0
      DO 20 I=1,N
   20 SUM=SUM+B(I)
      AVE = SUM/N
      END
```

7.2 Draw a possible organization for the runtime environment of the following C program, similar to that of Figure 7.4 (page 354).

a. After entry into block **A** in function **f**.
b. After entry into block **B** in function **g**.

```
int a[10];
char * s = "hello";

int f(int i, int b[])
{ int j=i;
  A:{ int i=j;
      char c = b[i];
      ...
    }
  return 0;
}
```

```
void g(char * s)
{ char c = s[0];
  B:{ int a[5];
      ...
    }
}

main()
{ int x=1;
  x = f(x,a);
  g(s);
  return 0;
}
```

7.3 Draw a possible organization for the runtime environment of the C program of Figure 4.1 (page 148) after the second call to **factor**, given the input string **(2)**.

7.4 Draw the stack of activation records for the following Pascal program, showing the control and access links, after the second call to procedure **c**. Describe how the variable **x** is accessed from within **c**.

```
program env;

procedure a;
var x: integer;

  procedure b;
    procedure c;
    begin
      x := 2;
      b;
    end;
  begin (* b *)
    c;
    end;

begin (* a *)
  b;
end;

begin (* main *)
  a;
end.
```

7.5 Draw the stack of activation records for the following Pascal program

a. After the call to **a** in the first call of **p**.

b. After the call to a in the second call of p.
c. What does the program print and why?

```
program closureEx(output);
var x: integer;

procedure one;
begin
  writeln(x);
end;

procedure p(procedure a);
begin
  a;
end;

procedure q;
var x:integer;
      procedure two;
      begin
        writeln(x);
      end;
begin
  x := 2;
  p(one);
  p(two);
end; (* q *)

begin (* main *)
  x := 1;
  q;
end.
```

7.6 Consider the following Pascal program. Assuming a user input consisting of the three numbers 1, 2, 0, draw the stack of activation records when the number 1 is printed the first time. Include all control and access links, as well as all parameters and global variables, and assume all procedures are stored in the environment as closures.

```
program procenv(input,output);

procedure dolist (procedure print);
var x: integer;
  procedure newprint;
  begin
    print;
    writeln(x);
  end;
```

```
begin (* dolist *)
  readln(x);
  if x = 0 then begin
    print;
    print;
  end
  else dolist(newprint);
end; (* dolist *)

procedure null;
begin
end;

begin (* main *)
  dolist(null);
end.
```

7.7 To perform completely static allocation, a FORTRAN77 compiler needs to form an estimate of the maximum number of temporaries required for any expression computation in a program. Devise a method for estimating the number of temporaries required to compute an expression by performing a traversal of the expression tree. Assume that expressions are evaluated from left to right and that every left subexpression must be saved in a temporary.

7.8 In languages that permit variable numbers of arguments in procedure calls, one way to find the first argument is to compute the arguments in reverse order, as described in Section 7.3.1, page 361.

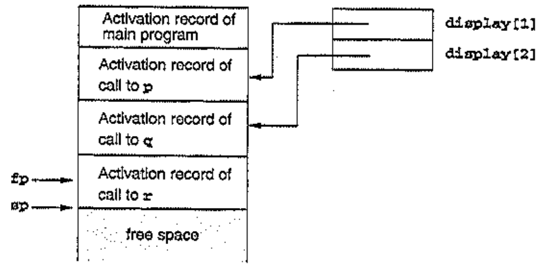
a. One alternative to computing the arguments in reverse would be to reorganize the activation record to make the first argument available even in the presence of variable arguments. Describe such an activation record organization and the calling sequence it would need.

b. Another alternative to computing the arguments in reverse is to use a third pointer (besides the sp and fp), which is usually called the ap (argument pointer). Describe an activation record structure that uses an ap to find the first argument and the calling sequence it would need.

7.9 The text describes how to deal with variable-length parameters (such as open arrays) that are passed by value (see Example 7.6, page 362) and states that a similar solution works for variable length local variables. However, a problem exists when *both* variable-length parameters and local variables are present. Describe the problem and a solution, using the following Ada procedure as an example:

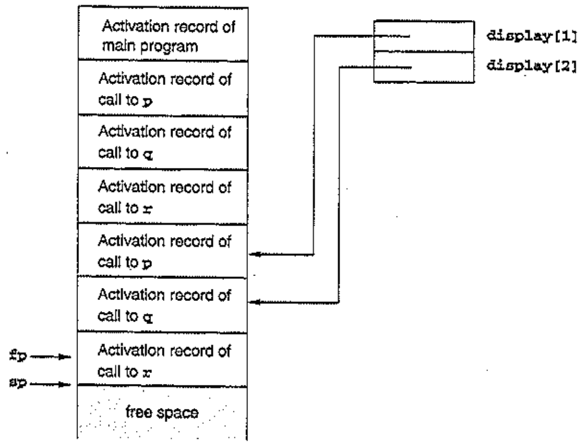
```
type IntAr is Array(Integer range <>) of Integer;
...
procedure f(x: IntAr; n:Integer) is
y: Array(1..n) of Integer;
i: Integer;
begin
  ...
end f;
```

7.10 An alternative to access chaining in a language with local procedures is to keep access links in an array outside the stack, indexed by nesting level. This array is called the **display**. For example, the runtime stack of Figure 7.12 (page 369) would look as follows with a display



Activation record of main program
Activation record of call to p
Activation record of call to q
Activation record of call to r
free space
fp
sp
display[1]
display[2]

while the runtime stack of Figure 7.13 (page 370) would look as follows:



Activation record of main program
Activation record of call to p
Activation record of call to q
Activation record of call to r
Activation record of call to p
Activation record of call to q
Activation record of call to r
free space
fp
sp
display[1]
display[2]

a. Describe how a display can improve the efficiency of nonlocal references from deeply nested procedures.
b. Redo Exercise 7.4 using a display.
c. Describe the calling sequence necessary to implement a display.
d. A problem exists in using a display in a language with procedure parameters. Describe the problem using Exercise 7.5.

7.11 Consider the following procedure in C syntax:

```
void f( char c, char s[10], double r )
{ int * x;
  int y[5];
  ...
}
```

a. Using the standard C parameter passing conventions, and assuming the data sizes integer = 2 bytes, char = 1 byte, double = 8 bytes, address = 4 bytes, determine the offsets from the fp of the following, using the activation record structure described in this chapter: (1) `c`, (2) `s[7]`, (3) `y[2]`.
b. Repeat (a) assuming all parameters are passed by value (including arrays).
c. Repeat (a) assuming all parameters are passed by reference.

7.12 Execute the following C program and explain its output in terms of the runtime environment:

```
#include <stdio.h>

void g(void)
{ {int x;
   printf("%d\n",x);
   x = 3;}
   {int y;
   printf("%d\n",y);}
}

int* f(void)
{ int x;
  printf("%d\n",x);
  return &x;
}

void main()
{ int *p;
  p = f();
  *p = 1;
  f();
  g();
}
```

7.13 Draw the memory layout of objects of the following C++ classes, together with the virtual function tables as described in Section 7.4.2:

```
class A
{ public:
  int a;
  virtual void f();
  virtual void g();
};
```

```
class B : public A
{ public:
  int b;
  virtual void f();
  void h();
};
class C : public B
{ public:
  int c;
  virtual void g();
}
```

7.14 A virtual function table in an object-oriented language saves traversing the inheritance graph searching for a method, but at a cost. Explain what the cost is.

7.15 Give the output of the following program (written in C syntax) using the four parameter passing methods discussed in Section 7.5:

```
#include <stdio.h>
int i=0;

void p(int x, int y)
{ x += 1;
  i += 1;
  y += 1;
}

main()
{ int a[2]={1,1};
  p(a[i],a[i]);
  printf("%d %d\n",a[0],a[1]);
  return 0;
}
```

7.16 Give the output of the following program (in C syntax) using the four parameter passing methods of Section 7.5:

```
#include <stdio.h>
int i=0;

void swap(int x, int y)
{ x = x + y;
  y = x - y;
  x = x - y;
}
```

```
main()
{ int a[3] = {1,2,0};
  swap(i,a[i]);
  printf("%d %d %d %d\n",i,a[0],a[1],a[2]);
  return 0;
}
```

7.17 Suppose that the FORTRAN77 subroutine `P` is declared as follows

```
SUBROUTINE P(A)
INTEGER A
PRINT *, A
A = A + 1
RETURN
END
```

and is called from the main program as follows:

```
CALL P(1)
```

In some FORTRAN77 systems, this will cause a runtime error. In others, no runtime error occurs, but if the subroutine is called again with 1 as its argument, it may print the value 2. Explain how both behaviors might occur in terms of the runtime environment.

7.18 A variation on pass by name is **pass by text**, in which the arguments are evaluated in delayed fashion, just as in pass by name, but each argument is evaluated in the environment of the called procedure rather than in the calling environment.

a. Show that pass by text can have different results than pass by name.

b. Describe a runtime environment organization and calling sequence that could be used to implement pass by text.

PROGRAMMING EXERCISES

7.19 As described in Section 7.5, pass by name, or delayed evaluation, can be viewed as packaging an argument in a function body (or suspension), which is called every time the parameter appears in the code. Rewrite the C code of Exercise 7.16 to implement the parameters of the `swap` function in this fashion, and verify that the result is indeed equivalent to pass by name.

7.20 **a.** As described in Section 7.5.4, an efficiency improvement in pass by name can be achieved by memoizing the value of an argument the first time it is evaluated. Rewrite your code of the previous exercise to implement such memoization, and compare the results to those of that exercise.

b. Memoization can cause different results from pass by name. Explain how this can happen.

7.21 Compaction (Section 7.4.4) can be made into a separate step from garbage collection and can be performed by `malloc` if a memory request fails because of the lack of a sufficiently large block.

a. Rewrite the `malloc` procedure of Section 7.4.3 to include a compaction step.

b. Compaction requires that the location of previously allocated space change, and this means that a program must find out about the change. Describe how to use a table of pointers to memory blocks to solve this problem, and rewrite your code of part (a) to include it.

NOTES AND REFERENCES

The fully static environment of FORTRAN77 (and earlier FORTRAN versions) represents a natural and straightforward approach to environment design and is similar to assembler environments. Stack-based environments became popular with the inclusion of recursion in languages such as Algol60 (Naur [1963]). Randell and Russell [1964] describe an early Algol60 stack-based environment in detail. The activation record organization and calling sequence for some C compilers is described in Johnson and Ritchie [1981]. The use of a display instead of access chains (Exercise 7.10) is described in detail in Fischer and LeBlanc [1991], including the problems with using it in a language with procedure parameters.

Dynamic memory management is discussed in many books on data structures, such as Aho, Hopcroft, and Ullman [1983]. A useful recent overview is given in Drozdek and Simon [1995]. Code for implementations of **`malloc`** and **`free`** that is similar to, but slightly less sophisticated than the code given in Section 7.4.3, appears in Kernighan and Ritchie [1988]. The design of a heap structure for use in compilation is discussed in Fraser and Hanson [1995].

An overview of garbage collection can be found in Wilson [1992] or Cohen [1981]. A generational garbage collector and runtime environment for the functional language ML is described in Appel [1992]. The Gofer functional language compiler (Jones [1984]) contains both a mark and sweep and a two-space garbage collector.

Budd [1987] describes a fully dynamic environment for a small Smalltalk system, including the use of the inheritance graph, and a garbage collector with reference counts. The use of a virtual function table in C++ is described in Ellis and Stroustrup [1990], together with extensions to handle multiple inheritance.

More examples of parameter passing techniques may be found in Louden [1993], where a description of lazy evaluation can also be found. Implementation techniques for lazy evaluation can be found in Peyton Jones [1987].