6.2 Expression evaluation

Historically, one of the first distinguishing characteristics of high-level programming languages was that they allowed the programmer to write algebraic expressions, such as the following Triangle expressions:

```
2 * (h + w)
(0 < i) /\ (i <= n)
a * b + (1 - (c * 2))
```

Such expressions are concise, and the notation is familiar from mathematics.

The implementation problem is the need to keep intermediate results somewhere, during evaluation of the more complicated expressions. For example, during evaluation of the expression '`a * b + (1 - (c * 2))`', the subexpressions '`a * b`', '`c * 2`', and '`1 - (c * 2)`' will give rise to intermediate results.

The problem can be seen in a more general setting if we consider the semantics of such expressions (1.21d). To evaluate an expression of the form 'E_1 O E_2', we must evaluate both the subexpressions E_1 and E_2, then apply the binary operator O to the two intermediate results. If we evaluate E_1 first, then its result must be kept somewhere safe during the evaluation of E_2.

Many machines provide ***registers*** that can be used to store intermediate results. Such a machine typically provides registers named `R0`, `R1`, `R2`, and so on, and instructions like those listed in Table 6.1. (Depending on the details of the instruction set, x could be the address of a storage cell, a literal, another register, etc.)

Example 6.10 Expression evaluation in a register machine

To evaluate the expression '`(a * b) + (1 - (c * 2))`' on our register machine, we could use the following sequence of instructions:

```
LOAD R1 a     – now R1 contains the value of a
MULT R1 b     – now R1 contains the value of a*b
LOAD R2 #1    – now R2 contains the literal value 1
LOAD R3 c     – now R3 contains the value of c
MULT R3 #2    – now R3 contains the value of c*2
SUB  R2 R3    – now R2 contains the value of 1-(c*2)
ADD  R1 R2    – now R1 contains the value of a*b+(1-(c*2))
```

Of course, if *address*⟦a⟧ = 100 (say), the first instruction would really be '`LOAD R1 100`', and the other instructions likewise. In order to make our examples of object code readable, we will adopt the convention that *a* stands for *address*⟦a⟧, *b* for *address*⟦b⟧, and so on.

□

Table 6.1 Typical instructions in a register machine

Instruction	Meaning
`STORE` *Ri a*	Store the value in register *i* at address *a*.
`LOAD` *Ri x*	Fetch the value of *x* and place it in register *i*.
`ADD` *Ri x*	Fetch the value of *x* and add it to the value in register *i*.
`SUB` *Ri x*	Fetch the value of *x* and subtract it from the value in register *i*.
`MULT` *Ri x*	Fetch the value of *x* and multiply it into the value in register *i*.

The object code for expression evaluation in registers is efficient but rather complicated. A compiler generating such code must assign a specific register to each intermediate result. It is important to do this well, but quite tricky. In particular, a problem arises when there are not enough registers for all the intermediate results. (See Exercise 6.11.)

A very different kind of machine is one that provides a ***stack*** for holding intermediate results. This allows us to evaluate expressions in a very natural way. Such a machine typically provides instructions like those listed in Table 6.2.

Example 6.11 Expression evaluation in a stack machine

To evaluate the expression '`(a * b) + (1 - (c * 2))`' on our stack machine, we could use the sequence of instructions shown below left. Note the one-to-one correspondence with the same expression's *postfix* representation, shown below right.

```
LOAD  a      a
LOAD  b      b
MULT         *
LOADL 1      1
LOAD  c      c
LOADL 2      2
MULT         *
SUB          -
ADD          +
```

Figure 6.7 shows the effect of each instruction on the stack, assuming that the stack is initially empty.[3]

□

[3] In Figure 6.7 and throughout this book, the stack is shown growing downwards, with the stack top nearest the bottom of the diagram. If this convention seems perverse, recall the convention for drawing trees in computer science textbooks! Shading indicates the unused space beyond the stack top.

Table 6.2 Typical instructions in a stack machine

Instruction	Meaning
STORE *a*	Pop the top value off the stack and store it at address *a*.
LOAD *a*	Fetch a value from address *a* and push it on to the stack.
LOADL *n*	Push the literal value *n* on to the stack.
ADD	Replace the top two values on the stack by their sum.
SUB	Replace the top two values on the stack by their difference.
MULT	Replace the top two values on the stack by their product.

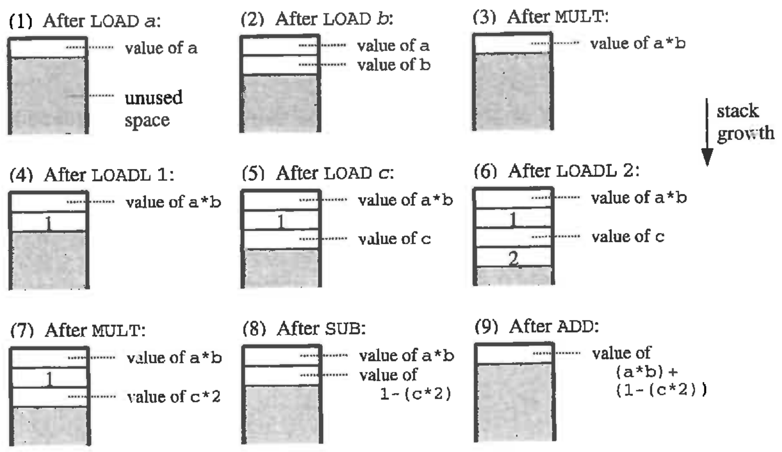


Figure 6.7 Evaluation of `(a*b)+(1-(c*2))` on a stack.

The stack machine requires more instructions than a register machine to evaluate an expression, but the individual instructions are simpler. There is one instruction for each operator, and one for each operand. In fact, as we noted in Example 6.11, the instruction sequence is in one-to-one correspondence with the expression's postfix representation. Because the problem of register assignment is removed, code generation for a stack machine is much simpler than code generation for a register machine.

The *net effect* of evaluating a (sub)expression on the stack is to leave its result at the stack top, on top of whatever was there already. For example, consider the evaluation of the subexpression '`c * 2`' – steps (5) through (7) in Figure 6.7. The net effect is to push the value of '`c * 2`' on to the stack top, and meanwhile the two values already on the stack remain undisturbed.

These desirable and simple properties of evaluation on the stack hold true regardless of how complicated the expression is. An expression involving function calls can be evaluated in just the same way. Likewise, an expression involving operands of different types (and therefore different sizes) can be evaluated in just the same way.

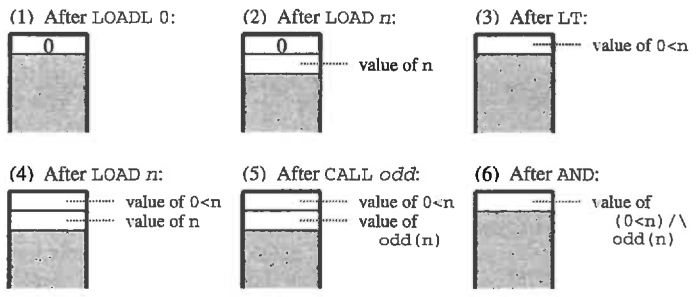


Figure 6.8 Evaluation of '`(0 < n) /\ odd(n)`' on a stack.

Example 6.12 Evaluation of function calls in a stack machine

To evaluate the expression '`(0 < n) /\ odd(n)`' on our stack machine, we could use the following sequence of instructions:

```
LOADL  0
LOAD   n
LT
LOAD   n
CALL   odd
AND
```

Figure 6.8 shows the effect of each instruction on the stack, assuming that the stack is initially empty. The instructions '`LT`' and '`AND`' are analogous to '`ADD`', '`SUB`', etc., in that each replaces two values at the stack top by a single value, but some of the values involved are truth values rather than integers.

Note the analogy between '`CALL` *odd*' and instructions like '`ADD`', '`LT`', etc. – each takes its argument(s) from the stack top, and replaces them by its result.

□

6.3 Static storage allocation

We now study the allocation of storage to variables. In this section we consider only global variables. In Section 6.4 we shall consider local variables, and in Section 6.6 heap variables.

Each variable in the source program requires enough storage to contain any value that might be assigned to it. The compiler cannot know, in general, which particular values will be assigned to the variable. But if the source language is statically typed, the compiler will know the variable's type, *T*. Thus, as a consequence of constant-size representation, the compiler will know how much storage needs to be allocated to the variable, namely *size T*.

The simplest case is storage allocation for *global variables*. These are variables that exist (and therefore occupy storage) throughout the program's run-time. The compiler can simply locate these variables at some fixed positions in storage. In this way it can decide each global variable's exact address. (More precisely, the compiler decides each global variable's address relative to the base of the storage region in which global variables are located.) This is called ***static storage allocation***.

Example 6.13 Static storage allocation

Consider the following Triangle program outline:

```
let
   type Date = record
                  y: Integer,
                  m: Integer,
                  d: Integer
               end;
   var a: array 3 of Integer;
   var b: Boolean;
   var c: Char;
   var t: Date
in
   ...
```

Assuming that each primitive value occupies one word, the global variables a, b, c, and t would be laid out as shown in Figure 6.9. Thus:

address⟦a⟧ = 0
address⟦b⟧ = 3
address⟦c⟧ = 4
address⟦t⟧ = 5

□

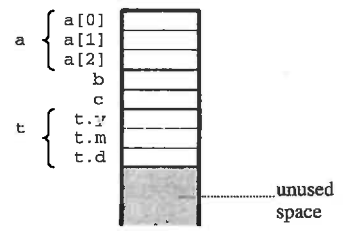


Figure 6.9 Layout of global variables for the program of Example 6.13.

6.4 Stack storage allocation

Let us now take into account *local variables*. A local variable *v* is one that is declared inside a procedure (or function). The variable *v* exists (i.e., occupies storage) only during an activation of that procedure. This time interval is called a *lifetime* of *v*. If the same procedure is activated several times, then *v* will have several lifetimes. (Each activation creates a distinct variable.)

Example 6.14 Stack storage allocation

Consider the following outline of a Triangle program, containing parameterless procedures Y and Z:

```
let
   var a: array 3 of Integer;
   var b: Boolean;
   var c: Char;

   proc Y () ~
      let
         var d: Integer;
         var e: record c: Char, n: Integer end
      in
         ...;

   proc Z () ~
      let
         var f: Integer
      in
         begin ...; Y(); ... end
in
   begin ...; Y(); ...; Z(); ... end
```

The variables a, b, and c are global. The variables d and e are local to procedure Y. The variable f is local to procedure Z.

The main program calls Y directly. Later it calls Z, which itself calls Y.

The lifetimes of the global and local variables are summarized in Figure 6.10. The lifetime of each local variable corresponds to an activation of the procedure in which it is declared. Since there are two activations of Y, its local variables have two lifetimes.

□

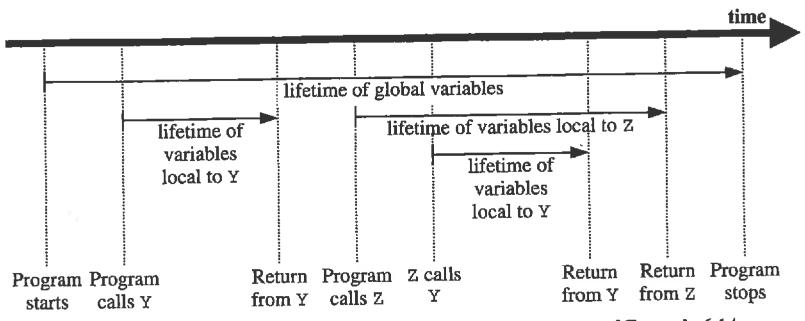

time
lifetime of global variables
lifetime of variables local to Y
lifetime of variables local to Z
lifetime of variables local to Y
Program starts
Program calls Y
Return from Y
Program calls Z
Z calls Y
Return from Y
Return from Z
Program stops

Figure 6.10 Lifetimes of global and local variables in the program of Example 6.14.

There are two important observations that we can make about programs with global and local variables:

- The global variables are the only ones that exist throughout the program's run-time.
- The lifetimes of local variables are properly nested. That is to say, the later a local variable is created, the sooner it must be deleted. The reason why variables' lifetimes are nested is simply that the procedure activations themselves are nested.

The first observation suggests that we should use static allocation for global variables only. The second observation suggests that for local variables we should use a *stack*. On entry to a procedure, we expand the stack to make space at the stack top for that procedure's local variables. On return, we release that space by contracting the stack. This is ***stack storage allocation***.

6.4.1 Accessing local and global variables

For the moment, we assume that a procedure may access global variables and its own local variables only. (This is the case in languages such as Fortran and C.)

The stack allocation method, in detail, works as follows. The global variables are always at the base of the stack (and therefore in fixed locations). At each point during run-time, the stack also contains a number of ***frames*** – one frame for each currently active procedure. Each procedure's frame contains space for its own local variables. Whenever a procedure is called, a new frame is pushed on to the stack. Whenever a procedure returns, its frame is popped off the stack.

Example 6.15 Stack frames

Consider again the Triangle program of Example 6.14. Successive snapshots of the stack are shown in Figure 6.11. (SB, ST, and LB are registers. The roles of these registers and of the dynamic links will be explained shortly.)

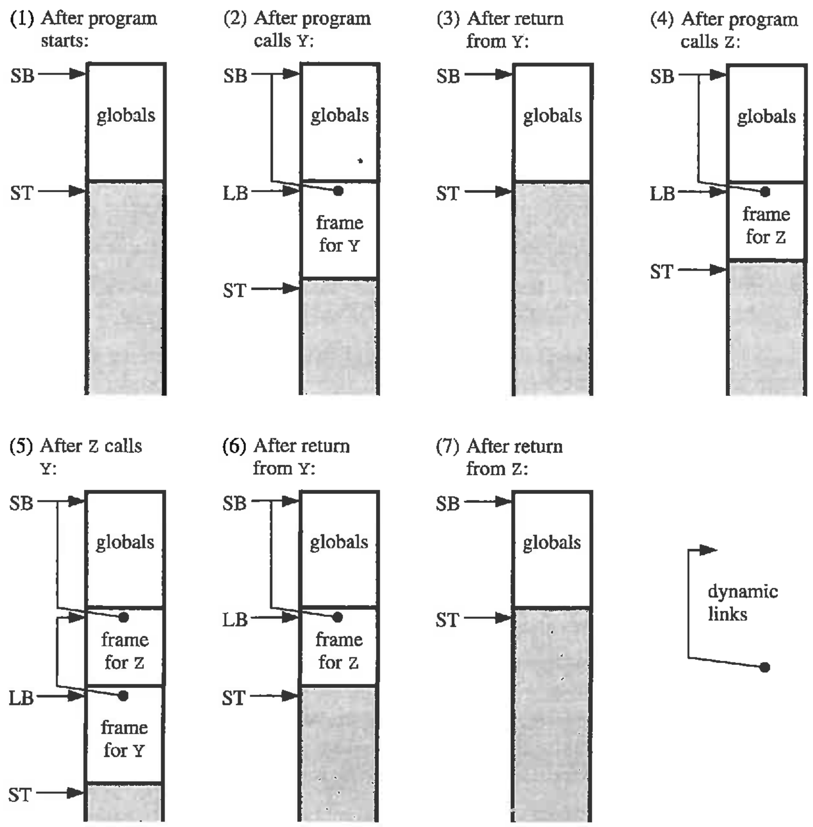


Figure 6.11 Stack snapshots in the program of Example 6.14 (showing dynamic links).

Initially, when the main program is running, only the global variables are occupying storage – snapshot (1). When the program calls procedure `Y`, a frame with space for `Y`'s local variables is pushed on to the stack – snapshot (2). When `Y` returns, this frame is popped, leaving only the global variables – snapshot (3). Later, when the program calls procedure `Z`, a frame for `Z` is pushed on to the stack – snapshot (4). When `Z` in turn calls `Y`, a frame for `Y` is pushed on top of that one – snapshot (5). And so on.

Compare Figure 6.11 in detail with Figure 6.10. This shows that the period during which the frame for `Z` is on the stack coincides with the lifetime of `Z`'s local variables, i.e., the activation of `Z`. Similarly, each period during which the frame for `Y` is on the stack coincides with a lifetime of `Y`'s local variables, i.e., an activation of `Y`.

□

The stack of course varies in size. Furthermore, the position of a particular frame within the stack cannot always be predicted in advance. For example, during the two activations of procedure `Y` in Example 6.15, the frames that provide space for `Y`'s local variables are in two different positions. So that variables can be addressed within the frames, registers must be dedicated to point to the frames. These dedicated registers, named SB, ST, and LB, are shown in Figure 6.11.

Register *SB* (Stack Base) is fixed, pointing to the base of the stack. This is where the global variables are located. So the global variables can be addressed relative to SB:

`LOAD` *d*`[SB]`	– fetch the value of the global variable at address *d*.
`STORE` *d*`[SB]`	– store a value in the global variable at address *d*.

Register *LB* (Local Base) points to the base of the topmost frame in the stack. This frame always contains the local variables of the currently running procedure. So these local variables can be addressed relative to LB.

`LOAD` *d*`[LB]`	– fetch the value of the local variable at address *d* relative to the frame base.
`STORE` *d*`[LB]`	– store a value in the local variable at address *d* relative to the frame base.

Register *ST* (Stack Top) points to the very top of the stack, i.e., the top of the topmost frame. If the currently running procedure evaluates an expression on the stack, the topmost frame expands and contracts, and ST keeps track of the frame boundary.

What about the frames that lie below the topmost one? Each such frame contains the local variables of a procedure that is active but not currently running. That frame is temporarily fixed in size. In the absence of a register pointing to the frame, the variables it contains cannot (currently) be accessed. Therefore only the global variables and the currently running procedure's local variables can be accessed.

As well as space for local variables, a frame contains certain housekeeping information, known collectively as ***link data***:

- The ***return address*** is the code address to which control will be returned at the end of the procedure activation. It is the address of the instruction following the call instruction that activated the procedure in the first place.
- The ***dynamic link*** is a pointer to the base of the underlying frame in the stack. It is the old content of LB, which will be restored at the end of the procedure activation.

The dynamic links are shown in Figure 6.11. Notice that they link together all the frames on the stack, in reverse order of creation.

A frame typically has the layout shown in Figure 6.12. The part shown as 'local data' contains space for local variables. It may be expanded to make space for anonymous data, such as the intermediate results of expression evaluation – but only when the frame is topmost in the stack. Since there are two words of link data, the local variables start at address displacement 2 within each frame.

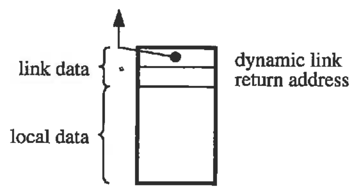


Figure 6.12 Layout of a frame (with dynamic but not static link).

Example 6.16 Accessing global and local variables

Consider again the Triangle program of Example 6.14. The layout of the globals and of the two procedures' frames would be as shown in Figure 6.13.

Here are some examples of instructions to access global and local variables:

`LOAD 0[SB]`	– for any part of the program to fetch the value of global variable `a[0]`
`LOAD 4[SB]`	– for any part of the program to fetch the value of global variable `c`
`LOAD 2[LB]`	– for procedure `Y` to fetch the value of its local variable `d`
`LOAD 4[LB]`	– for procedure `Y` to fetch the value of its local variable `e.n`
`LOAD 2[LB]`	– for procedure `Z` to fetch the value of its local variable `f`

It might appear that the local variables `d` and `f` have the same address, 2[LB]. But remember that `d` can be accessed only by procedure `Y`, and while that procedure is running LB is pointing to the base of a frame containing `Y`'s local variables. Similarly, `f` can be accessed only by procedure `Z`, and while that procedure is running LB is pointing to the base of a frame containing `Z`'s local variables.

□

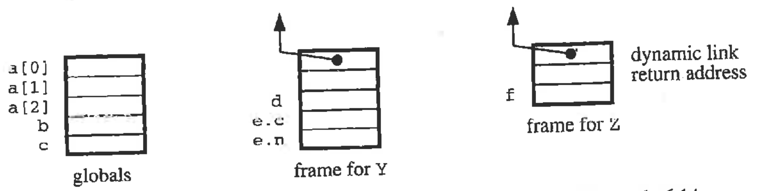


Figure 6.13 Layout of globals and frames for the program of Example 6.14.

The compiler cannot determine the *absolute* address of a local variable; but it can determine its address displacement *relative to the base of the frame* containing it. In order that the local variable can be accessed at run-time, we need only arrange that a particular register (such as LB) points to the base of the frame.

Stack allocation is economical of storage. If static allocation were used on the program of Example 6.14, every variable would occupy storage space throughout the program's run-time. With stack allocation, however, only some of the local variables occupy storage at any particular time. This is illustrated by Figure 6.11. (At snapshot (5) , all the local variables are occupying storage at the same time; but this rarely happens in real programs with many procedures.)

Even more importantly, stack storage allocation works well in the presence of recursive procedures, whereas static allocation would not work at all. The effect of recursion will be discussed in Section 6.5.4.

6.4.2 Accessing nonlocal variables

So far we have assumed that a procedure can access only global variables and its own local variables. Now we remove this restriction. Procedures are allowed to be nested. Moreover, a procedure *P* may directly access any *nonlocal* variable, i.e., a variable that is not local to *P* but is local to an enclosing procedure. (This is the case in languages such as Pascal and Ada.)

As we have already observed, the compiler cannot determine the absolute address of any variable (other than a global), but only its address displacement within a frame. To access the variable at run-time, we must arrange for a particular register to point to the base of that frame. We use SB to point to the global variables, and LB to point to the frame containing variables local to the running procedure. Now we also need registers pointing to any frames that contain accessible nonlocal variables. We introduce registers L1, L2, etc., for this purpose.

Example 6.17 Accessing nonlocal variables

Figure 6.14 shows an outline of a Triangle program with nested procedures. The levels of nesting are indicated by shades of gray. As a consequence of Triangle's scope rules:

- Procedure P can access global variables and its own local variables.
- Procedure Q can access global variables, its own local variables, and variables local to the enclosing procedure P.
- Procedure R can access global variables, its own local variables, and variables local to the enclosing procedures P and Q.
- Procedure S can access global variables, its own local variables, and variables local to the enclosing procedure P.

Figure 6.15 shows a possible sequence of stack snapshots as this program runs.

```
let
   var g1: Integer;
   var g2: array 3 of Boolean;

   proc P () ~
      let
         var p1: Boolean;
         var p2: Integer;

         proc Q () ~
            let
               var q: array 3 of Char;

               proc R () ~
                  let
                     var r: Boolean
                  in
                     begin ... end !R!
            in
               begin ... end; !Q!

         proc S () ~
            let
               var s: array 4 of Char
            in
               begin ... end !S!
      in
         begin ... end !P!
in
   begin ... end
```

Key:

▩	routine level 3
▨	routine level 2
░	routine level 1
☐	routine level 0

Figure 6.14 A Triangle program with global and local variables.

Consider snapshot (2), taken when procedure P has called procedure Q. At this time, register LB points to the frame that contains Q's local variables, and register L1 points to the underlying frame that contains P's local variables. This is necessary because Q can access P's local variables. Q might contain instructions like the following:

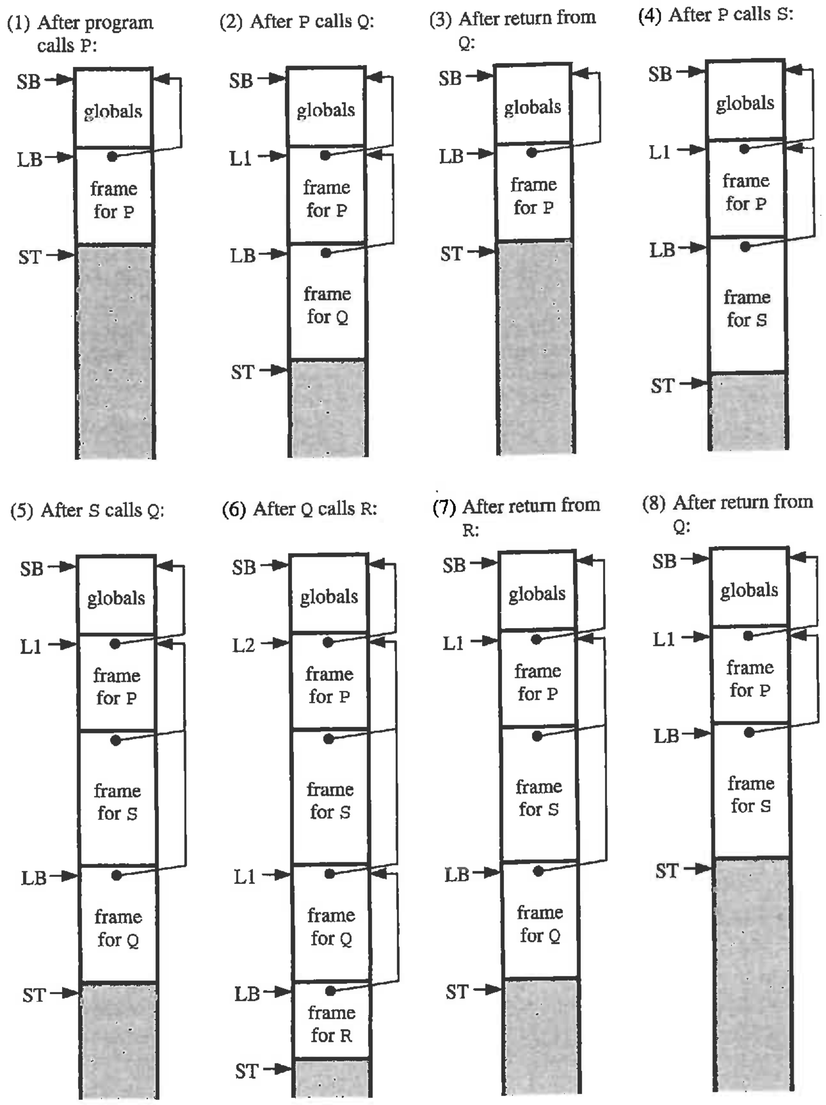


Figure 6.15 Stack snapshots in Example 6.17 (showing static links).

LOAD *d*[SB]	– for procedure Q to fetch the value of a global variable
LOAD *d*[LB]	– for procedure Q to fetch the value of a variable local to itself
LOAD *d*[L1]	– for procedure Q to fetch the value of a variable local to P

where in each case *d* is the appropriate address displacement.

Now consider snapshot (5), also taken when procedure P has called procedure Q, but this time indirectly through S. At this time also, LB points to the frame that contains Q's local variables, and L1 points to the underlying frame that contains P's local variables. So the above instructions will still work correctly. No register points to the frame that contains S's local variables. This is correct, because Q may not directly access these variables.

The following snapshot (6) illustrates a situation where R, the most deeply-nested procedure, has been activated by Q. Now register LB points to R's frame, register L1 points to the frame belonging to Q (the procedure immediately enclosing R), and register L2 points to the frame belonging to P (the procedure immediately enclosing Q). This allows R to access not only its own local variables, but also variables local to Q and P:

LOAD *d*[SB]	– for procedure R to fetch the value of a global variable
LOAD *d*[LB]	– for procedure R to fetch a variable local to itself
LOAD *d*[L1]	– for procedure R to fetch a variable local to Q
LOAD *d*[L2]	– for procedure R to fetch a variable local to P

But no register points to the frame containing S's local variables, since R may not directly access these variables.

□

By arranging for registers L1, L2, etc., to point to the correct frames, we allow each procedure to access nonlocal variables. To achieve this, we need to add a third item to the link data in each frame. Consider a routine (procedure or function) *R* that is enclosed by routine *R´* in the source program. In a frame that contains variables local to routine *R*:

- The ***static link*** is a pointer to the base of an underlying frame that contains variables local to *R´*. The static link is set up when *R* is called. (This will be demonstrated in Section 6.5.1.)

The static links were shown in Figure 6.15. Notice that the static link in a frame for Q always points to a frame for P, since it is P that immediately encloses Q in the source program. Similarly, the static link in a frame for R always points to a frame for Q, and the static link in a frame for S always points to a frame for P. (The static link in a frame for P always points to the globals, but that static link is actually redundant.)

The layout of a stack frame is now as shown in Figure 6.16. Since there are now three words of link data, the local variables now start at address displacement 3. Figure 6.17 shows the layout of frames for the procedures in Figure 6.14.

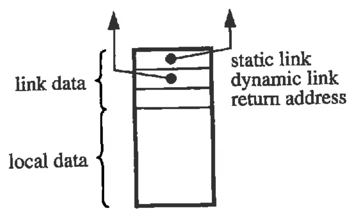


Figure 6.16 Layout of a frame (with dynamic and static links).

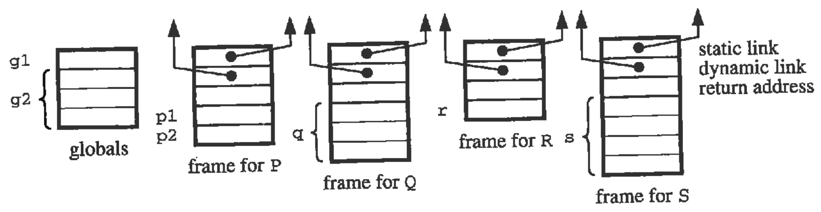


Figure 6.17 Layout of globals and frames for the program of Figure 6.14 (with static links).

The static links allow us to set up the registers L1, L2, etc. LB points to the first word of the topmost frame, which is the static link and points to a frame for the enclosing routine. Therefore:

$$L1 = \mathit{content}(LB) \tag{6.25}$$

where *content*(*r*) stands for the content of the word to which register *r* points. In turn, L1 points to the next static link. Therefore:

$$L2 = \mathit{content}(L1) = \mathit{content}(\mathit{content}(LB)) \tag{6.26}$$
$$L3 = \mathit{content}(L2) = \mathit{content}(\mathit{content}(\mathit{content}(LB))) \tag{6.27}$$

...

These equations are invariants: L1, L2, etc., automatically change whenever LB changes, i.e., on a routine call or return.

At any moment during run-time:

- Register SB points to the global variables.
- Register LB points to the topmost frame, which always belongs to the routine *R* that is currently running.
- Register L1 points to a frame belonging to the routine *R*′ that encloses *R* in the source program.
- Register L2 points to a frame belonging to the routine *R*″ that encloses *R*′ in the source program.

And so on.

The collection of registers LB, L1, L2, ..., and SB is often called the ***display***. The display allows access to local, nonlocal, and global variables. The display changes whenever a routine is called or returns.

The critical property of the display is that the *compiler* can always determine which register to use to access any variable. A global variable is always addressed relative to SB. A local variable is always addressed relative to LB. A nonlocal variable is addressed relative to one of the registers L1, L2, ... The appropriate register is determined entirely by the nesting levels of the routines in the source program.

We assign routine levels as follows: the main program is at *routine level 0*; the body of each routine declared at level 0 is at *routine level 1*; the body of each routine declared at level 1 is at *routine level 2*; and so on.

Let *v* be a variable declared at routine level *l*, and let *v*'s address displacement be *d*. Then the current value of *v* is fetched by various parts of the code as follows:

If $l = 0$ (i.e., *v* is a global variable):

`LOAD d[SB]`	– for any code to fetch the value of *v*

If $l > 0$ (i.e., *v* is a local variable):

`LOAD d[LB]`	– for code at level *l* to fetch the value of *v*
`LOAD d[L1]`	– for code at level *l*+1 to fetch the value of *v*
`LOAD d[L2]`	– for code at level *l*+2 to fetch the value of *v*
...	

Storing to variable *v* is analogous.

6.5 Routines

A ***routine*** (or *subroutine*) is the machine-code equivalent of a procedure or function in a high-level language. Control is transferred to a routine by means of a ***call*** instruction (or instruction sequence). Control is transferred back to the caller by means of a ***return*** instruction in the routine.

When a routine is called, some ***arguments*** may be passed to it. An argument could be, for example, a value or an address. There may be zero, one, or many arguments. A routine may also return a ***result*** – that is if it corresponds to a function in the high-level language.

We have already studied one aspect of routines, namely allocation of storage for local variables. In this section we study other important aspects:

- protocols for passing arguments to routines and returning their results
- how static links are determined

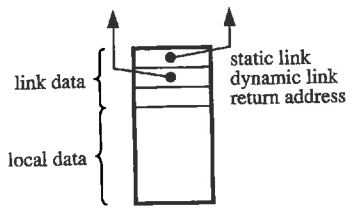


Figure 6.16 Layout of a frame (with dynamic and static links).

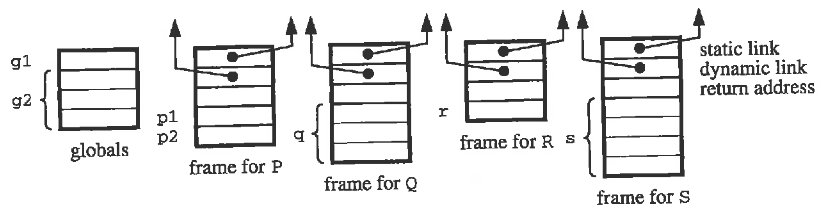


Figure 6.17 Layout of globals and frames for the program of Figure 6.14 (with static links).

The static links allow us to set up the registers L1, L2, etc. LB points to the first word of the topmost frame, which is the static link and points to a frame for the enclosing routine. Therefore:

$$\text{L1} = \textit{content}(\text{LB}) \qquad (6.25)$$

where *content*(*r*) stands for the content of the word to which register *r* points. In turn, L1 points to the next static link. Therefore:

$$\text{L2} = \textit{content}(\text{L1}) = \textit{content}(\textit{content}(\text{LB})) \qquad (6.26)$$
$$\text{L3} = \textit{content}(\text{L2}) = \textit{content}(\textit{content}(\textit{content}(\text{LB}))) \qquad (6.27)$$
...

These equations are invariants: L1, L2, etc., automatically change whenever LB changes, i.e., on a routine call or return.

At any moment during run-time:

- Register SB points to the global variables.
- Register LB points to the topmost frame, which always belongs to the routine *R* that is currently running.
- Register L1 points to a frame belonging to the routine *R*′ that encloses *R* in the source program.
- Register L2 points to a frame belonging to the routine *R*″ that encloses *R*′ in the source program.

And so on.

The collection of registers LB, L1, L2, ..., and SB is often called the ***display***. The display allows access to local, nonlocal, and global variables. The display changes whenever a routine is called or returns.

The critical property of the display is that the *compiler* can always determine which register to use to access any variable. A global variable is always addressed relative to SB. A local variable is always addressed relative to LB. A nonlocal variable is addressed relative to one of the registers L1, L2, ... The appropriate register is determined entirely by the nesting levels of the routines in the source program.

We assign routine levels as follows: the main program is at *routine level 0*; the body of each routine declared at level 0 is at *routine level 1*; the body of each routine declared at level 1 is at *routine level 2*; and so on.

Let *v* be a variable declared at routine level *l*, and let *v*'s address displacement be *d*. Then the current value of *v* is fetched by various parts of the code as follows:

If $l = 0$ (i.e., *v* is a global variable):

```
    LOAD d[SB]        – for any code to fetch the value of v
```

If $l > 0$ (i.e., *v* is a local variable):

```
    LOAD d[LB]        – for code at level l to fetch the value of v
    LOAD d[L1]        – for code at level l+1 to fetch the value of v
    LOAD d[L2]        – for code at level l+2 to fetch the value of v
    ...
```

Storing to variable *v* is analogous.

6.5 Routines

A ***routine*** (or *subroutine*) is the machine-code equivalent of a procedure or function in a high-level language. Control is transferred to a routine by means of a ***call*** instruction (or instruction sequence). Control is transferred back to the caller by means of a ***return*** instruction in the routine.

When a routine is called, some ***arguments*** may be passed to it. An argument could be, for example, a value or an address. There may be zero, one, or many arguments. A routine may also return a ***result*** – that is if it corresponds to a function in the high-level language.

We have already studied one aspect of routines, namely allocation of storage for local variables. In this section we study other important aspects:

- protocols for passing arguments to routines and returning their results
- how static links are determined

- the arguments themselves
- the implementation of recursive routines.

6.5.1 Routine protocols

When a routine is called, the arguments are computed by the caller, and used by the called routine. Thus we need a suitable ***routine protocol***, a convention to ensure that the caller deposits the arguments in the place where the called routine expects to find them. Conversely, the routine's result (if any) is computed by the routine, and used by the caller. Thus the routine protocol must also ensure that, on return, the called routine deposits its result in the place where the caller expects to find it.

There are numerous possible routine protocols. Sometimes the implementor has to design a protocol from scratch. More often, the operating system dictates a standard protocol to which all compilers must conform. In every case, the choice of protocol is influenced by the target machine, such as whether the latter is a register machine or a stack machine.

Example 6.18 Routine protocol for a register machine

In a register machine, the routine protocol might be:

- Pass the first argument in R1, the second argument in R2, etc.
- Return the result (if any) in R0.

Such a simple protocol works only if there are fewer arguments than registers, and if every argument and result is small enough to fit into a register. In practice, a more elaborate protocol is needed. (See Exercise 6.20.)

□

Example 6.19 Routine protocol for a stack machine

In a stack machine, the routine protocol might be:

- Pass the arguments at the stack top.
- Return the result (if any) at the stack top, in place of the arguments.

This protocol places no limits on the number of arguments, nor on the sizes of the arguments or result.

□

The stack-based routine protocol of Example 6.19 is simple and general. For that reason it is adopted by the abstract machine TAM. Variants of this protocol are also adopted by machines equipped with both registers and stacks (such as the Pentium). Due to the popularity of this protocol, we shall study the TAM routine protocol in detail.

Some routines (functions) have results, whereas others (procedures) do not. For the sake of simplicity, we shall discuss the protocol in terms of the more general case, namely a routine with a result. We can treat a procedure as a routine with a 0-word 'result'. (Compare the use of a **`void`** function in C or Java, or a `unit` function in ML, to achieve the effect of a procedure.)

Before calling a routine, the caller is responsible for evaluating the arguments and pushing them on to the stack top. (Since expression evaluation is done on the stack, as in Section 6.2, the stack top is where the arguments will be evaluated anyway.) After return, the caller can expect to find the result at the stack top, in the place formerly occupied by the arguments. This is shown in Figure 6.18. The net effect of calling the routine (ignoring any side effects) will be to replace the arguments by the result at the stack top.

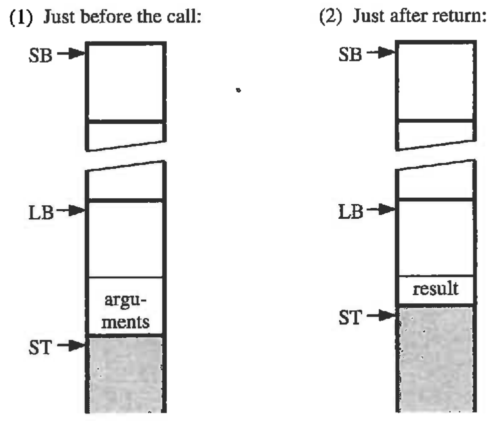


Figure 6.18 The TAM routine protocol.

The called routine itself is responsible for evaluating its result and depositing it in the correct place. Let us examine a call to some routine *R*, from the point of view of the routine itself (see Figure 6.19):

(1) Immediately before the call, the arguments to be passed to *R* must be at the stack top.

(2) The call instruction pushes a new frame, on top of the arguments. Initially, the new frame contains only link data. Its return address is the address of the code following the call instruction. Its dynamic link is the old content of LB. Its static link is supplied by the call instruction. Now LB is made to point to the base of the new frame, and control is transferred to the first instruction of *R*.

(3) The instructions within *R* may expand the new frame, to make space for local variables and to perform expression evaluation. These instructions can access the arguments relative to LB. Immediately before return, *R* evaluates its result and leaves it at the stack top.

(4) The return instruction pops the frame and the arguments, and deposits the result in the place formerly occupied by the arguments. LB is reset using the dynamic link, and control is transferred to the instruction at the return address.

TAM has a single call instruction that does all the work described in step (2). Some other machines have a less powerful call instruction, and we need a *sequence* of instructions to do the same work. TAM also has a single return instruction that does all the work described in step (4).

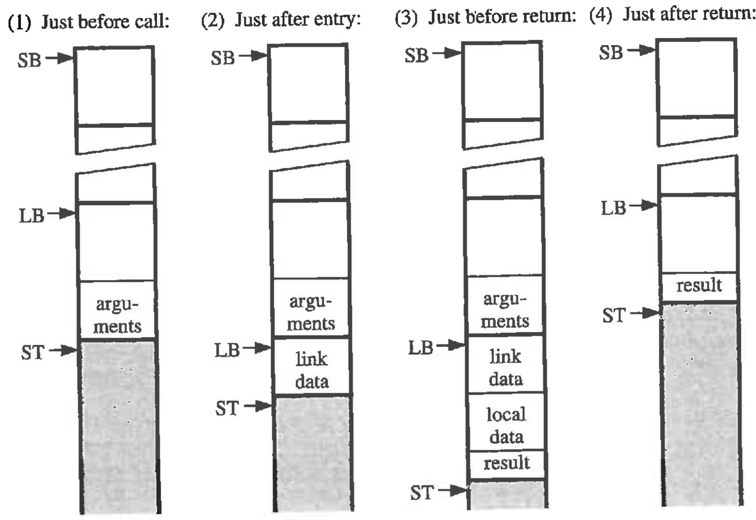


Figure 6.19 TAM routine call and return (in detail).

Example 6.20 Passing arguments

Consider the following Triangle program, containing a function `F` with two parameters, and a procedure `W` with one parameter:

```
let var g: Integer;

   func F (m: Integer, n: Integer) : Integer ~
      m * n;
```

```
   proc W (i: Integer) ~
      let const s ~ i * i
      in
         begin
         putint(F(i, s));
         putint(F(s, s))
         end

in
   begin
   getint(var g);
   W(g+1)
   end
```

This (artificial) program reads an integer, and writes the cube and fourth power of its successor.

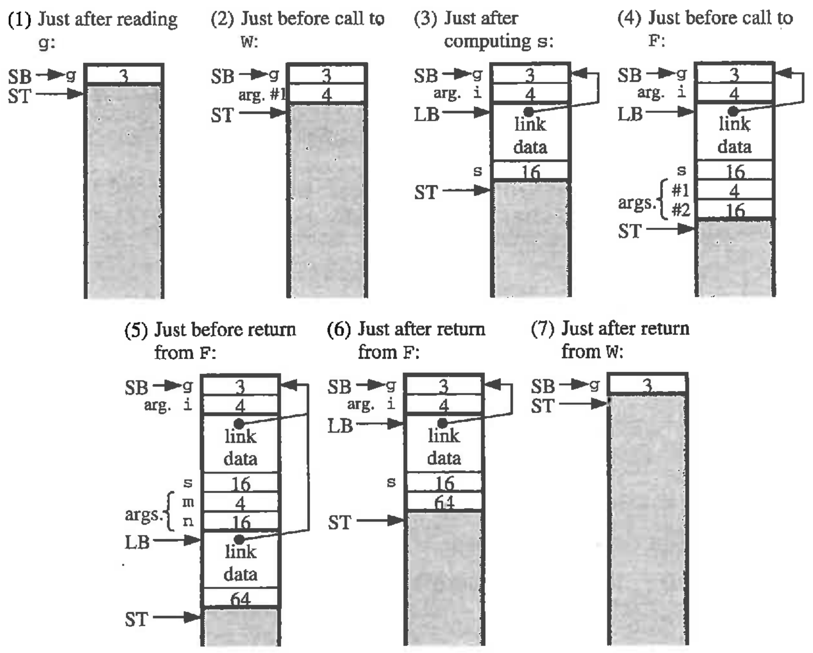

(1) Just after reading g:
(2) Just before call to W:
(3) Just after computing s:
(4) Just before call to F:
(5) Just before return from F:
(6) Just after return from F:
(7) Just after return from W:
SB
ST
LB
g
arg. #1
arg. i
s
args.
#1
#2
m
n
link data
3
4
16
64

Figure 6.20 Arguments and results in Example 6.20.

Figure 6.20 shows a sequence of stack snapshots. The main program first reads an integer, say 3, into the global variable g – snapshot (1). Then it evaluates 'g+1', which yields 4, and leaves that value at the stack top as the argument to be passed to procedure W – snapshot (2).

On entry to procedure W, a new frame is pushed on to the stack top, and the argument becomes known to the procedure as i. The constant s is defined by evaluating 'i*i', which yields 16 – snapshot (3). Next, the procedure prepares to evaluate 'F(i, s)' by pushing the two arguments, 4 and 16, on to the stack top – snapshot (4).

On entry to function F, a new frame is pushed on to the stack, and the arguments become known to the function as m and n, respectively. F immediately evaluates 'm*n' to determine its result, 64, and leaves that value on the stack top – snapshot (5). On return from F, the topmost frame and the arguments are popped, and the result is deposited in place of the arguments – snapshot (6). This value is used immediately as an argument to putint, which writes it out.

Similarly, W evaluates 'F(s, s)', yielding 256, and passes the result as an argument to putint. Finally, on return from W, the topmost frame and the argument are popped; this time there is no result to replace the arguments – snapshot (7).

It is instructive to study the corresponding object code. It would look something like this (using symbolic names for routines, and omitting some minor details):

```
    PUSH       1        – expand globals to make space for g
    LOADA      0[SB]    – push the address of g
    CALL       getint   – read an integer into g
    LOAD       0[SB]    – push the value of g
    CALL       succ     – add 1
    CALL(SB)   W        – call W (using SB as static link)
    POP        1        – contract globals
    HALT

W:  LOAD       -1[LB]   – push the value of i
    LOAD       -1[LB]   – push the value of i
    CALL       mult     – multiply; the result will be the value of s
    LOAD       -1[LB]   – push the value of i
    LOAD       3[LB]    – push the value of s
    CALL(SB)   F        – call F (using SB as static link)
    CALL       putint   – write the value of F(i,s)
    LOAD       3[LB]    – push the value of s
    LOAD       3[LB]    – push the value of s
    CALL(SB)   F        – call F (using SB as static link)
    CALL       putint   – write the value of F(s,s)
    RETURN(0)  1        – return, replacing the 1-word argument
                            by a 0-word 'result'
```

```
F:  LOAD       -2[LB]   – push the value of m
    LOAD       -1[LB]   – push the value of n
    CALL       mult     – multiply
    RETURN(1)  2        – return, replacing the 2-word argument pair
                            by a 1-word result
```

Here the instruction 'LOADA *d*[*r*]' (load address) pushes the address *d* + register *r* on to the stack, and 'RETURN (*n*) *d*' returns from the current routine with an *n*-word result, removing *d* words of argument data. (*Note:* In TAM, operations like addition, subtraction, logical negation, etc., are performed by calling primitive routines – *add*, *sub*, *not*, etc. This avoids the need to provide many individual instructions – ADD, SUB, NOT, etc.)

□

6.5.2 Static links

One loose end in our description of the routine protocol is how the static link is determined. Recall that the static link is needed only for a source language with nested block structure (such as Pascal, Ada, or Triangle). The scope rules of such a language guarantee that, at the time of call, the correct static link is in one or other of the display registers. The caller need only copy it into the newly-created frame.

Example 6.21 Static links

Consider the outline Triangle program of Figure 6.14. Some stack snapshots were shown in Figure 6.15.

When P calls Q, the required static link is a pointer to a frame for P itself, since P encloses Q in the source program, and the caller can find that pointer in LB – snapshots (1) and (2). Similarly, when P calls S, the required static link is a pointer to a frame for P itself, since P encloses S, and the caller can find that pointer in LB – snapshots (3) and (4).

When S calls Q, the required static link is a pointer to a frame for P, since P encloses Q, and the caller can find that pointer in L1 – snapshots (4) and (5).

If R were to call Q or S, the required static link would be a pointer to a frame for P, since P encloses Q and S, and the caller could find that pointer in L2 – snapshot (6).

Here is a summary of all the possible calls in this program:

```
CALL(SB)  P     – for any call to P

CALL(LB)  Q     – for P to call Q
CALL(L1)  Q     – for Q to call Q (recursively)
CALL(L2)  Q     – for R to call Q
CALL(L1)  Q     – for S to call Q
CALL(LB) R      – for Q to call R
CALL(L1) R      – for R to call R (recursively)

CALL(LB) S      – for P to call S
CALL(L1) S      – for Q to call S
CALL(L2) S      – for R to call S
CALL(L1) S      – for S to call S (recursively)
```

(In the TAM call instruction, the field in parentheses nominates the register whose content is to be used as the static link.)

□

In general, the *compiler* can always determine which register to use as the static link in any call instruction. A call to a global routine (i.e., one declared at the outermost level of the source program) always uses SB. A call to a local routine (i.e., one declared inside the currently running routine) always uses LB. A call to any other routine uses one of the registers L1, L2, ... The appropriate register is determined entirely by the nesting levels of the routines in the source program.

Let *R* be a routine declared at routine level *l* (thus the *body* of *R* is at level *l*+1). Then *R* is called as follows:

If *l* = 0 (i.e., *R* is a global routine):
```
    CALL(SB) R        – for any call to R
```
If *l* > 0 (i.e., *R* is enclosed by another routine):
```
    CALL(LB) R        – for code at level l to call R
    CALL(L1) R        – for code at level l+1 to call R
    CALL(L2) R        – for code at level l+2 to call R
    ...
```

(Compare this with the code used for addressing variables, at the end of Section 6.4.2.)

6.5.3 Arguments

We have already seen some examples of argument passing. We now examine two other aspects of arguments: how the called routine accesses its own arguments, and how arguments are represented under different parameter mechanisms.

According to the routine protocol studied in the previous subsection, the arguments to be passed to a routine are deposited at the top of the *caller*'s frame (or at the top of the globals, if the caller is the main program). Since the latter frame is just under the *called* routine's frame, the called routine can find its arguments just under its own frame. In other words, the arguments have small negative addresses relative to the base of the called routine's frame. In all other respects, they can be accessed just like variables local to the called routine.

Example 6.22 Accessing arguments

In the Triangle program of Example 6.17, the two routines accessed their arguments as follows:

```
LOAD -1[LB]     – for procedure W to fetch its argument i

LOAD -2[LB]     – for function F to fetch its argument m
LOAD -1[LB]     – for function F to fetch its argument n
```

□

We can easily implement a variety of parameter mechanisms:

- *Constant parameter* (as in Triangle and ML) or *value parameter* (as in Pascal, C, and Java): The argument is an ordinary value (such as an integer or record). The caller evaluates an expression to compute the argument value, and leaves it on the stack.
- *Variable parameter* (as in Triangle and Pascal) or *reference parameter* (as in C++): The argument is the address of a variable. The caller simply pushes this address on to the stack.
- *Procedural/functional parameter* (as in Triangle, Pascal, and ML): The argument is a (static link, code address) pair representing a routine. This pair, known as a ***closure***, contains just the information that will be needed to call the argument routine.

Constant parameters have already been illustrated, in Example 6.20. Value parameters differ in only one respect: the formal parameter is treated as a local *variable*, and thus may be updated. If procedure W had a *value* parameter i, the procedure body could contain assignments to i, implemented by 'STORE -1[LB]'. Note, however, that the word corresponding to i will be popped on return from P, so any such updating would have no effect outside the procedure. This conforms to the intended semantics of value parameters.

Example 6.23 Variable parameter

Consider the following outline Triangle program, containing a procedure S with a variable parameter n as well as a constant parameter i:

```
let
   proc S (var n: Integer, i: Integer) ~
      n := n + i;
   var b: record y: Integer, m: Integer, d: Integer end
in
   begin
   b := {y ~ 1978, m ~ 5, d ~ 5};
   S(var b.m, 6);
   end
```

Figure 6.21 shows some snapshots of the stack as this program runs.

The procedure call 'S(var b.m, 6)' works by first pushing the *address* of the variable b.m, along with the value 6, and then calling S.

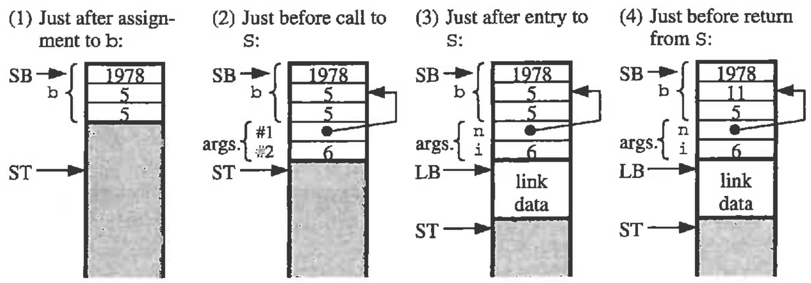
The procedure S itself works as follows. Its first argument is the address of some variable. S can access the variable by indirect addressing. It can fetch the variable's value by an indirect load instruction, and update it by an indirect store instruction.

We can see this by studying the TAM code corresponding to the above program:

```
    ...
    LOADL       1978
    LOADL       5
    LOADL       5
    STORE(3)    0[SB]      – store a record value in b
    LOADA       1[SB]      – push the address of b.m
    LOADL       6          – push the value 6
    CALL(SB)    S          – call S
    ...
S:  LOAD        -2[LB]     – push the argument address n
    LOADI                  – push the value contained at that address
    LOAD        -1[LB]     – push the argument value i
    CALL        add        – add (giving the value of n+i)
    LOAD        -2[LB]     – push the argument address n
    STOREI                 – store the value of n+i at that address
    RETURN(0)   2          – return, replacing the 2-word argument
                             pair by a 0-word 'result'
```

Here the instruction LOADI (load indirect) pops an address off the stack, and then fetches a value from that address. STOREI (store indirect) pops an address and a value, and then stores that value at that address.

□



(1) Just after assignment to b:
(2) Just before call to S:
(3) Just after entry to S:
(4) Just before return from S:
SB
b
1978
5
5
11
ST
args.
#1
#2
6
n
i
LB
link data

Figure 6.21 Variable and constant parameters in Example 6.23.

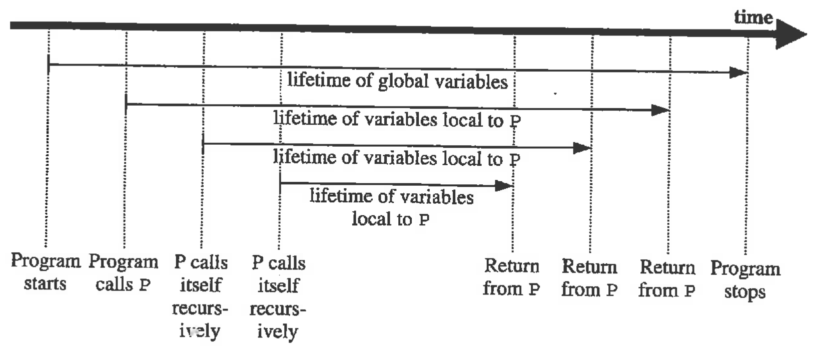
6.5.4 Recursion

We have already noted that stack allocation is more economical of storage than static allocation. As a bonus, stack allocation supports the implementation of *recursive* routines. In fact, there is nothing to add to the techniques introduced in Section 6.4; we need only illustrate how stack allocation works in the presence of recursive routines.

Example 6.24 Recursion

Consider the following Triangle program. It includes a recursive procedure, P, that writes a given nonnegative integer, i, to a given base, b, in the range 2–10:

```
let
   proc P (i: Integer, b: Integer) ~
      let const d ~ chr(i//b + ord('0'))
      in
         if i < b then
            put(d)
         else
            begin P(i//b, b); put(d) end;
   var n: Integer
in
   begin
   getint(var n); P(n, 8)
   end
```



time
lifetime of global variables
lifetime of variables local to P
lifetime of variables local to P
lifetime of variables local to P
Program starts
Program calls P
P calls itself recursively
P calls itself recursively
Return from P
Return from P
Return from P
Program stops

Figure 6.22 Lifetimes of variables local to the recursive procedure of Example 6.24.

Figure 6.22 shows the lifetimes of the variables in this program (and also formal parameters such as i and b, and declared constants such as d, because they too occupy storage). Note that each recursive activation of P creates a new set of local variables, which coexist with the local variables of continuing activations. In Figure 6.22, like Figure 6.10, all the variable's lifetimes are nested. This suggests that stack allocation will cope with recursion.

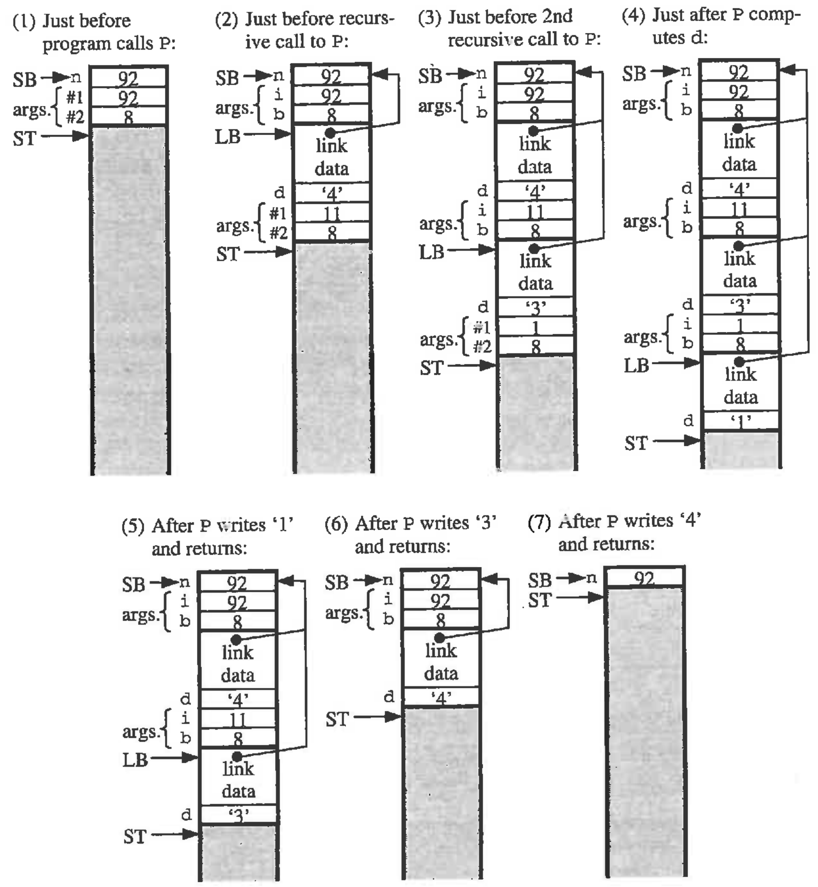

(1) Just before program calls P:
(2) Just before recursive call to P:
(3) Just before 2nd recursive call to P:
(4) Just after P computes d:
(5) After P writes '1' and returns:
(6) After P writes '3' and returns:
(7) After P writes '4' and returns:
SB
ST
LB
args.
n
i
b
d
#1
#2
92
8
11
1
link
data
'4'
'3'
'1'

Figure 6.23 Stack snapshots for the recursive procedure of Example 6.24.

Figure 6.23 shows some stack snapshots as this program runs. Having read a value into n, say 92, the main program pushes a pair of arguments, here 92 and 8, in preparation for calling P – snapshot (1). Inside P these arguments are known as i and b, respectively. In the constant definition, d is defined to be '4'. Now, since the value of 'i < b' is *false*, P pushes a pair of arguments, here 11 and 8, in preparation for calling itself recursively – snapshot (2). Inside P these arguments are known as i and b. At this point there are two activations of P, the original one and the recursive one, and each activation has its own arguments i and b. In the constant definition, using the *current* activation's i and b, d is defined to be '3'. Now, since the value of 'i < b' is again *false*, P pushes a pair of arguments, here 1 and 8, in preparation for calling itself recursively – snapshot (3). In this third activation of P, d is defined to be '1', but the value of 'i < b' turns out to be *true* – snapshot (4). So P merely writes '1', then returns to the second activation of itself – snapshot (5). This activation writes '3', and then returns to the original activation of P – snapshot (6). This activation writes '4', and then returns to the main program – snapshot (7).

□

6.6 Heap storage allocation

In Section 6.4 we saw how local variables are allocated storage. A lifetime of a local variable corresponds exactly to an activation of the procedure, function, or block within which the local variable was declared. Since their lifetimes are always nested, local variables can be allocated storage on a stack.

On the other hand, a ***heap variable*** is allocated (created) by executing an ***allocator*** (such as new in Pascal, malloc in C, or **new** in Java). The allocator returns a pointer through which the heap variable can be accessed. Later the heap variable may be deallocated, either explicitly by executing a ***deallocator*** (such as dispose in Pascal or free in C), or automatically (as in Java). The heap variable's lifetime extends from the time it is allocated until the time it is deallocated.

Thus heap variables behave quite differently from local variables. Consequently they demand a different method of storage allocation, called ***heap storage allocation***.

Example 6.25 Heap storage allocation

Consider the following outline of a Pascal program, which manipulates linked lists:

```
type IntList = ...;     {linked list of integers}
     Symbol  = array [1..2] of Char;
     SymList = ...;     {linked list of symbols}

var ns: IntList; ps: SymList;

procedure insertI (i: Integer; var l: IntList);
   ...;     {Insert a node containing i at the front of list l.}
```

storage). Note that each recursive activation of P creates a new set of local variables, which coexist with the local variables of continuing activations. In Figure 6.22, like Figure 6.10, all the variable's lifetimes are nested. This suggests that stack allocation will cope with recursion.

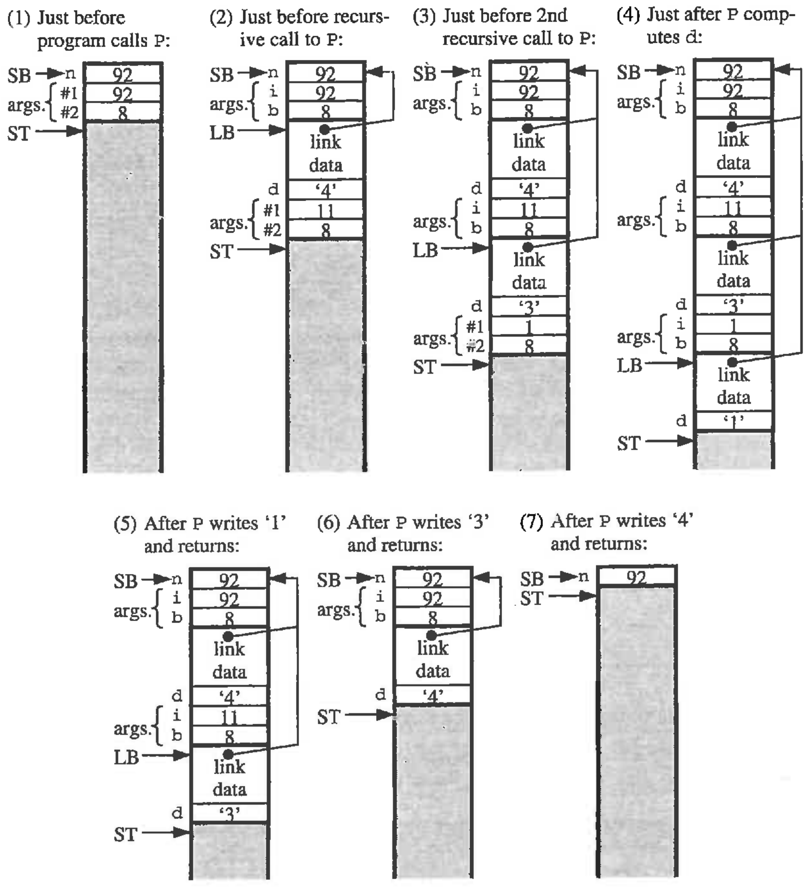

(1) Just before program calls P:
(2) Just before recursive call to P:
(3) Just before 2nd recursive call to P:
(4) Just after P computes d:
(5) After P writes '1' and returns:
(6) After P writes '3' and returns:
(7) After P writes '4' and returns:
SB
LB
ST
n
args.
#1
#2
i
b
d
link data
92
8
'4'
11
'3'
1
'1'

Figure 6.23 Stack snapshots for the recursive procedure of Example 6.24.

Figure 6.23 shows some stack snapshots as this program runs. Having read a value into n, say 92, the main program pushes a pair of arguments, here 92 and 8, in preparation for calling P – snapshot (1). Inside P these arguments are known as i and b, respectively. In the constant definition, d is defined to be '4'. Now, since the value of 'i < b' is *false*, P pushes a pair of arguments, here 11 and 8, in preparation for calling itself recursively – snapshot (2). Inside P these arguments are known as i and b. At this point there are two activations of P, the original one and the recursive one, and each activation has its own arguments i and b. In the constant definition, using the *current* activation's i and b, d is defined to be '3'. Now, since the value of 'i < b' is again *false*, P pushes a pair of arguments, here 1 and 8, in preparation for calling itself recursively – snapshot (3). In this third activation of P, d is defined to be '1', but the value of 'i < b' turns out to be *true* – snapshot (4). So P merely writes '1', then returns to the second activation of itself – snapshot (5). This activation writes '3', and then returns to the original activation of P – snapshot (6). This activation writes '4', and then returns to the main program – snapshot (7).

□

6.6 Heap storage allocation

In Section 6.4 we saw how local variables are allocated storage. A lifetime of a local variable corresponds exactly to an activation of the procedure, function, or block within which the local variable was declared. Since their lifetimes are always nested, local variables can be allocated storage on a stack.

On the other hand, a ***heap variable*** is allocated (created) by executing an ***allocator*** (such as new in Pascal, malloc in C, or **new** in Java). The allocator returns a pointer through which the heap variable can be accessed. Later the heap variable may be deallocated, either explicitly by executing a ***deallocator*** (such as dispose in Pascal or free in C), or automatically (as in Java). The heap variable's lifetime extends from the time it is allocated until the time it is deallocated.

Thus heap variables behave quite differently from local variables. Consequently they demand a different method of storage allocation, called ***heap storage allocation***.

Example 6.25 Heap storage allocation

Consider the following outline of a Pascal program, which manipulates linked lists:

```
type IntList = ...;     {linked list of integers}
     Symbol  = array [1..2] of Char;
     SymList = ...;     {linked list of symbols}

var ns: IntList; ps: SymList;

procedure insertI (i: Integer; var l: IntList);
   ...;      {Insert a node containing i at the front of list l.}
```

```
procedure deleteI (i: Integer; var l: IntList);
    ...;     {Delete the first node containing i from list l.}

procedure insertS (s: Symbol; var l: SymList);
    ...;     {Insert a node containing s at the front of list l.}

procedure deleteS (s: Symbol; var l: SymList);
    ...;     {Delete the first node containing s from list l.}

...
ns := nil;          ps := nil;             (1)
insertI(6, ns);     insertS('Cu', ps);
insertI(9, ns);     insertS('Ag', ps);
insertI(10, ns);    insertS('Au', ps);     (2)
deleteI(10, ns);    deleteS('Cu', ps);     (3)
insertI(12, ns);    insertS('Pt', ps);     (4)
```

Here, the heap variables are nodes of linked lists. Procedures `insertI` and `insertS` allocate nodes, and procedures `deleteI` and `deleteS` deallocate nodes.

Figure 6.24 shows the lifetimes of the heap variables. Observe that there is no particular pattern to their lifetimes: the program allocates and deallocates them whenever it chooses. Those not deallocated by the program cease to exist when the program stops.

□

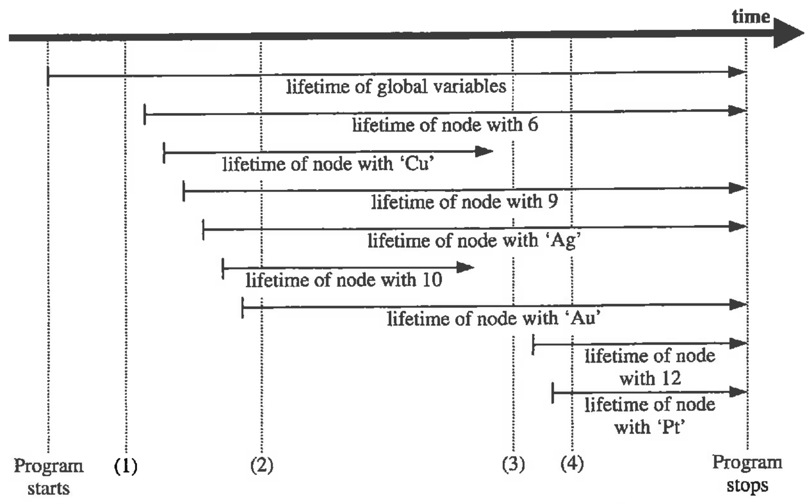


Figure 6.24 Lifetimes of heap variables in Example 6.25.

6.6.1 Heap management

Since heap variables can be allocated and deallocated at any time, their lifetimes bear no particular relationship to one another. So these variables are allocated on a ***heap***, a storage region managed differently from a stack.

The heap will expand and (occasionally) contract as the program runs. Let us assume that registers HB (Heap Base) and HT (Heap Top) point to the boundaries of the heap. (Note the analogy with SB and ST, which point to the boundaries of the stack.)

Since the stack and the heap both expand and contract, it is a good idea to place them at opposite ends of the available storage space. Contraction of the stack leaves more space for the heap to expand, and *vice versa*. It is only when the stack and heap collide that the program must fail due to storage exhaustion.

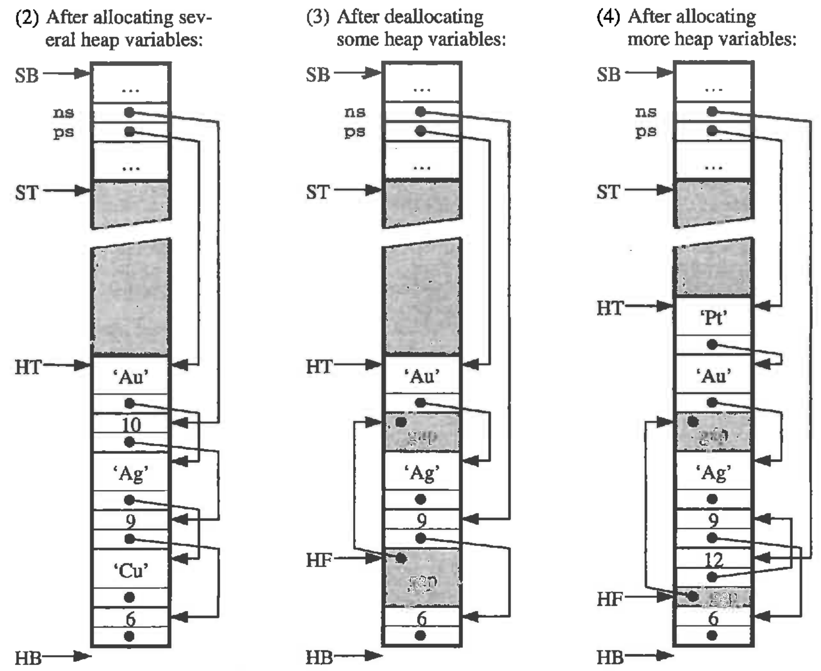

(2) After allocating several heap variables:
(3) After deallocating some heap variables:
(4) After allocating more heap variables:
SB
ns
ps
ST
HT
HF
HB
...
'Au'
10
'Ag'
9
'Cu'
6
'Pt'
12

Figure 6.25 Snapshots of the heap in Example 6.25.

Figure 6.25 shows several snapshots of the heap as the program of Example 6.25 runs. The heap is initially empty, but expands as nodes are allocated – snapshot (2).

When nodes are deallocated, gaps appear – snapshot (3). Some of these gaps may be partly or wholly refilled as further nodes are allocated – snapshot (4).

Deallocation of a heap variable at the heap top causes the heap to contract. But deallocation elsewhere in the heap leaves a ***gap***, i.e., a piece of unused storage surrounded by used storage. (This never happens in the stack, where deallocation always takes place at the stack top.) Gaps may appear in the heap at any time, and they have to be managed. Thus the object program must be supported by a run-time module called the ***heap manager***.

The heap manager privately maintains a ***free list***, which is a linked list of gaps within the heap. Each gap contains a size field, and a link to the next gap. The size fields are necessary because the gaps are of differing sizes. The heap manager needs a pointer to the first gap in the free list; we shall call this HF (Heap Free-list pointer). In Figure 6.25, the free list is initially empty – snapshot (2), but later accumulates some gaps – snapshots (3) and (4).

A simple heap manager works as follows. To *allocate* a heap variable of size *s*, the heap manager searches the free list for a gap of size at least *s*:

- If it finds a gap whose size is *s* exactly, it removes that gap from the free list.
- If it finds a gap whose size is greater than *s*, it replaces that gap in the free list by the residual gap.
- If there is no gap big enough, it expands the heap by the amount *s*.
- If there is no room to expand the heap, storage is exhausted and the program fails.

To *deallocate* a heap variable, the heap manager simply adds it to the free list.

All this seems very straightforward, but in practice such a simple heap manager does not work very well. One major problem is ***fragmentation***. As many allocations and deallocations take place, gaps tend to become smaller and more numerous. (This can already be seen in snapshot (4) of Figure 6.25.) When the heap manager tries to allocate a heap variable, there might be no *single* gap big enough, although the *total* amount of free space is sufficient.

There are several techniques for reducing fragmentation:

- When allocating a heap variable, the heap manager could choose the *smallest* gap that is big enough (rather than choosing just any gap). This technique implies an increased time overhead on allocation, because the entire free list must be searched – unless the heap manager keeps the gaps sorted by size, which implies an increased time overhead on deallocation. This technique helps to preserve large gaps for when they are really needed, but also tends to make many very small gaps.[4]

[4] This is a *best-fit* allocation algorithm. A *worst-fit* allocation algorithm is also worth considering. It makes very small gaps less frequent, but also tends to split up large gaps.

- When deallocating a heap variable, the heap manager could *coalesce* the heap variable with any adjacent gap(s). In a naive implementation, this technique implies an increased time overhead on deallocation, because the entire free list must be searched. (See Example 6.26 and Exercise 6.22.) A more sophisticated algorithm is possible that uses additional space to allow adjacent regions to be coalesced without the need to search the free list (Standish 1998).
- The heap manager could occasionally *compact* the heap by shifting heap variables together. This technique is quite complicated to manage: to shift a heap variable, the heap manager must find and redirect all pointers to that heap variable. The technique implies a large time overhead whenever the heap is compacted, because the whole heap is affected. (See Example 6.27 and Exercise 6.23.)

Example 6.26 Coalescence in the heap

Figure 6.26 illustrates coalescence of gaps in the heap. Deallocating heap variable *c* allows the space it occupied to be coalesced with an adjacent gap. Deallocating heap variable *b* is even more effective, since the space it occupied can be coalesced with two adjacent gaps. (What would the free list look like without coalescence of gaps?)

□

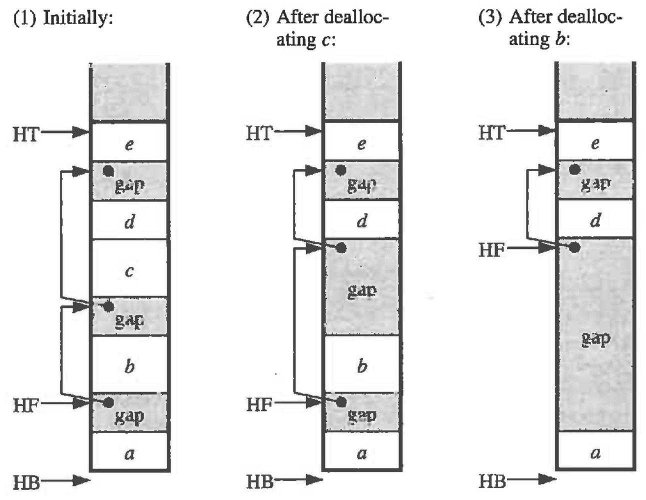


Figure 6.26 Coalescing deallocated heap variables with adjacent gaps in the heap.

Example 6.27 Heap compaction

Figure 6.27 illustrates heap compaction. To understand this, start by convincing yourself that the states of the heap before and after compaction are equivalent.

Since a pointer is represented by the address of the heap variable it points to, moving a heap variable to a different address implies that every pointer to it must be adjusted. There are two pointers to heap variable *c*: one in the stack, a second in another heap variable, *b*. Indeed, *b* and *c* point to each other. All these pointers have to be adjusted consistently.

□

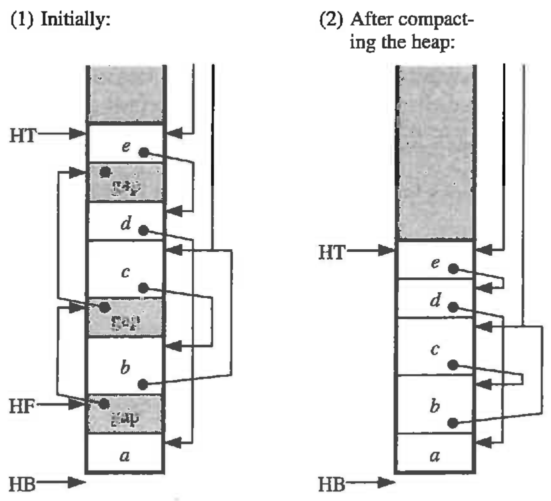


Figure 6.27 Compacting the heap.

Example 6.27 illustrates the complications that can arise in heap compaction. There may be several pointers to the same heap variable. Pointers may be located both in the stack and in other heap variables. *All* such pointers must be found and adjusted.

To implement heap compaction, the heap manager must create a table containing the old address and new address of each heap variable. Then, for every pointer in the stack or heap, it must use the table to replace the old address by the new address. Finally it can actually copy the heap variables to their new addresses.

Compaction is usually combined with garbage collection, so we defer a more detailed explanation until Section 6.6.3.

6.6.2 Explicit storage deallocation

Programming languages differ in how they allow heap variables to be deallocated. In this subsection we study ***explicit storage deallocation***. A program may explicitly deallocate a heap variable by, for example, calling `dispose` in Pascal, or `free` in C.

Example 6.28 Explicit storage deallocation

The procedure `deleteI` of Example 6.25 might be implemented as follows:

```
procedure deleteI (i: Integer; var l: IntList);
   {Delete the first node containing i from list l.}
   var p, q: IntList;
   begin
   ...;     {Make q point to the first node containing i in list l,
              and make p point to the preceding node (if any).}
   if q = l then
      {If q is at the start of the list, then delete it by making
       the head of the list point to q's successor. }
      l := q^.tail
   else
      {Otherwise remove node q by making the previous node p
       point to q's successor. }
      p^.tail := q^.tail;
   {Node q^ is now no longer part of the list and the space associated
    with it can be deallocated.}
   dispose(q)
   end {deleteI}
```

□

Explicit storage deallocation is efficient, and allows the programmer fine control over heap storage allocation.

In Examples 6.25 and 6.28 explicit storage deallocation was used in a controlled manner. In practice, however, explicit storage deallocation is notoriously error-prone. Two problems arise frequently in practice: garbage accumulation and dangling pointers.

A heap variable is ***inaccessible***, or ***garbage***, if there exists no pointer to it. If such a heap variable has not been deallocated, the space it occupies is wasted.

Example 6.29 Garbage in the heap

Figure 6.28 illustrates how garbage can appear. At first `p` and `q` point to different heap variables, *a* and *b*, respectively – snapshot (1). After the assignment '`p := q`', both `p` and `q` point to *b* – snapshot (2). Now there exists no pointer to *a*, so the latter is garbage. Worse still, there is no way to retrieve this situation; without a pointer to *a*, it cannot be explicitly deallocated.

This situation could have been averted by greater care on the programmer's part: *a* should have been deallocated *before* the assignment '`p := q`'.

□

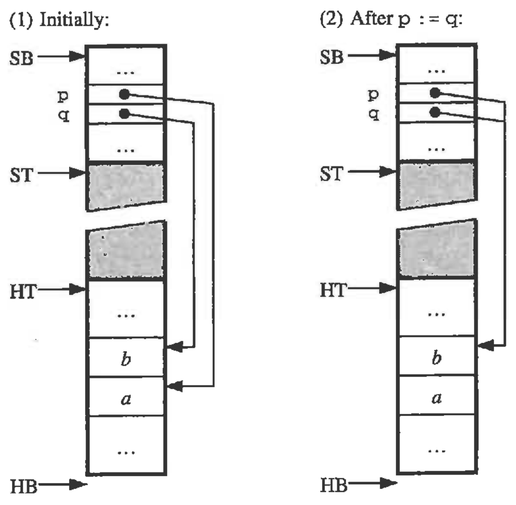


Figure 6.28 Garbage in the heap.

Since pointers can be copied, several pointers to the same heap variable might exist at the same time. When one of these pointers is used to deallocate the heap variable, the other pointers are left pointing to a gap. They are called ***dangling pointers***.

The program might accidentally use a dangling pointer to update a heap variable that no longer exists. The effect will be to corrupt the gap left by deallocation, or perhaps to corrupt a new heap variable subsequently allocated (by chance) in that gap.

Example 6.30 Dangling pointer

The following Pascal program fragment illustrates a possible effect of a dangling pointer:

```
var p, q: ^T1; r: ^T2;
...
new(p);   p^ := value of type T1;
q := p;                                    (1)
```

```
...;
dispose(p);                                (2)
...;
new(r);   r^ := value of type T2;          (3)
...;
q^ := value of type T1;                    (4)
```

Figure 6.29 shows the effect. At first, both `p` and `q` point to the same heap variable, which contains a value of type `T1` – snapshot (1). Now the program uses `dispose(p)` to deallocate `p^`, which adds this heap variable to the free list, and which (in a typical implementation) changes `p` to *nil*. But of course `dispose` knows nothing about `q`. (How could it?) So `q` still contains the same address, which is a dangling pointer – snapshot (2). Any assignment to `q^` now would corrupt the gap.

Later the program executes '`new(r)`', and then stores a value of type `T2` in the newly allocated heap variable – snapshot (3). This new heap variable might (purely by chance) be located at the same address as the old one, as shown in snapshot (3).

This is a situation in which a disaster is just waiting to happen. The program might attempt to inspect `q^`, expecting to find a value of type `T1`. Worse still, it might attempt to store a value of type `T1` in `q^`, which would corrupt the value already there.

□

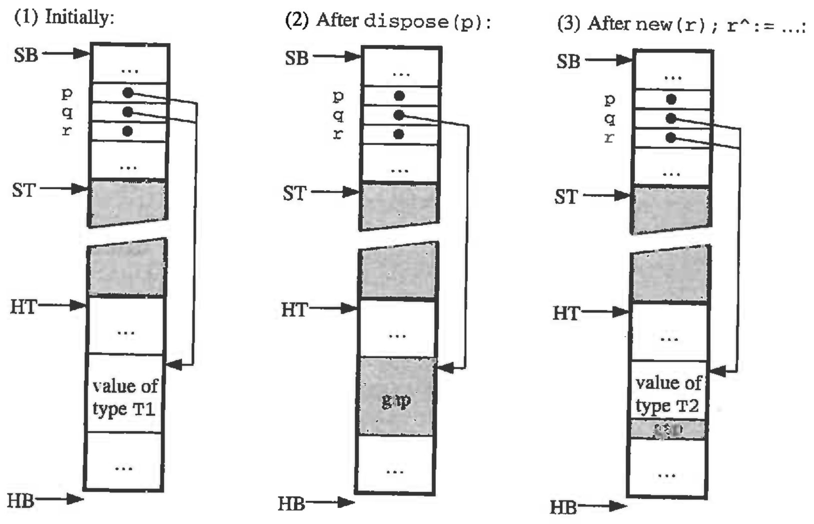


Figure 6.29 Effect of a dangling pointer.

The situation illustrated in Figure 6.29 is no less than a violation of Pascal's type rules, which are supposed to guarantee that every pointer of type ^*T* is either *nil* or points to a heap variable of type *T*. To restore the guarantee, a Pascal implementation would have to take extreme measures. One measure would be never to allocate a heap variable in space released by deallocation. But this would prevent the heap from ever contracting. An alternative measure would be to make `dispose` find, and change to *nil*, *all* pointers to the deallocated heap variable. But this would imply a large time overhead, negating the main advantage of explicit storage deallocation.

6.6.3 Automatic storage deallocation and garbage collection

In a programming language that supports explicit storage deallocation, the appearance of garbage is usually a consequence of a programming error. In a language that does not support explicit storage deallocation, garbage must inevitably appear. A heap variable becomes inaccessible when the last pointer to it is overwritten (as in Example 6.29) or otherwise ceases to exist.

Fortunately, ***automatic storage deallocation*** of inaccessible heap variables is possible. The space they occupied can then be recycled by being added to the free list. The recycling process is called ***garbage collection***, and is performed by a heap manager routine called the ***garbage collector***.

Garbage collection is a feature of the run-time support for some imperative languages (such as Ada), most object-oriented languages (including Java), and all functional languages (such as Lisp and ML). It is generally not provided for languages (such as Pascal and C) that have explicit storage deallocation.

Many garbage collection algorithms have been invented. ***Mark–sweep garbage collection*** is a simple and commonly-used algorithm. The idea is to mark as accessible every heap variable that can be reached (directly or indirectly) by pointers from the stack. All other heap variables are inaccessible and may be deallocated.

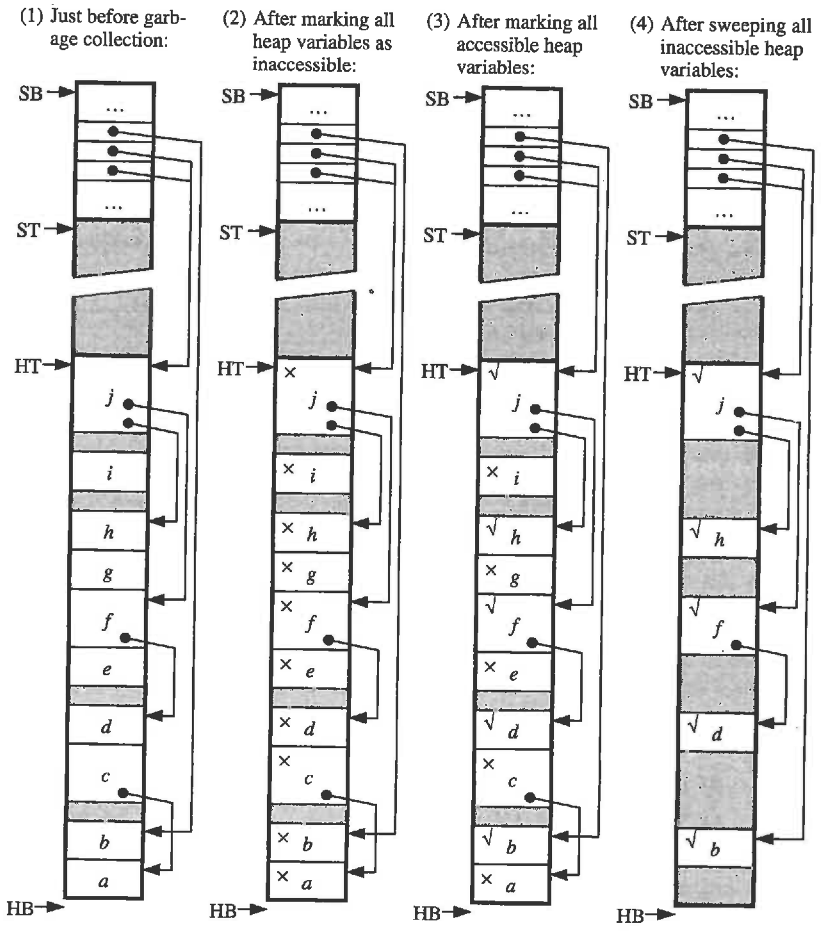
Example 6.31 Mark–sweep garbage collection

Figure 6.30 illustrates mark–sweep garbage collection. The initial state of the heap shows typical patterns. The stack contains pointers to some of the heap variables (*b* and *j*). These heap variables in turn contain pointers to other heap variables (*f* and *h*), and so on (*d*). But there are also some inaccessible heap variables to which no pointers exist (*c*, *e*, *g*, and *i*), and others to which the only pointers are themselves in inaccessible heap variables (*a*).

The garbage collector starts by marking *all* heap variables as inaccessible (shown by ×).

Next, the garbage collector follows all chains of pointers from the stack, marking each heap variable it reaches as accessible (shown by √). By following the first pointer

from the stack it reaches *b*. It marks *b* as accessible. By following the second pointer from the stack it reaches *j*; by following the pointers in *j* it reaches *f* and *h*; and by following the pointer in *f* it reaches *d*. It marks all these heap variables as accessible. By following the third pointer from the stack it reaches *j*; but it has already marked *j* as accessible, so it need take no further action there.



(1) Just before garbage collection:
(2) After marking all heap variables as inaccessible:
(3) After marking all accessible heap variables:
(4) After sweeping all inaccessible heap variables:
SB
ST
HT
HB
...
j
i
h
g
f
e
d
c
b
a
×
√

Figure 6.30 Mark–sweep garbage collection.

The garbage collector finishes by scanning the heap for heap variables still marked as inaccessible: *a*, *c*, *e*, *g*, and *i*. These really *are* inaccessible, so the garbage collector deallocates them.

□

Mark–sweep garbage collection may be seen to be a simple graph algorithm. The heap variables and the stack are the nodes of a directed graph, and the pointers are the edges. Our aim is to determine the largest subgraph in which all nodes can be reached from the stack node. The *mark–sweep garbage collection algorithm* can be expressed recursively as follows:

```
Procedure to collect garbage:
    mark all heap variables as inaccessible;
    scan all frames in the stack;
    add all heap variables still marked as inaccessible to the free list.

Procedure to scan the storage region R:
    for each pointer p in R:
        if p points to a heap variable v that is marked as inaccessible:
            mark v as accessible;
            scan v.
```

For this algorithm to work, it must be able to visit all heap variables, it must know the size of each, and it must be able to mark each as accessible or inaccessible. One way to meet these requirements is to extend each heap variable with the following hidden fields: a size field; a link field (used to connect all heap variables into a single linked list, to permit them all to be visited); and a 1-bit accessibility field. These hidden fields are used by the garbage collector but invisible to the programmer.

Another requirement is that pointers must be distinguishable from other data in the store. (This is a requirement not only for garbage collection, but also for heap compaction as described in Section 6.6.2.) This is an awkward problem: pointers are represented by addresses, which typically have exactly the same form as integers. Some clever techniques have been devised to solve this problem – see Wilson (1992).

6.7 Run-time organization for object-oriented languages

Object-oriented (OO) languages give rise to interesting and special problems in run-time organization. An object is a special kind of record. Attached to each object are some methods, each method being a kind of procedure or function that is able to operate on that object. Objects are grouped into classes, such that all objects of the same class have identical structure and identical methods.

We shall assume the following more precise definitions:

- An ***object*** is a group of instance variables, to which a group of instance methods are attached.
- An ***instance variable*** is a named component of a particular object.
- An ***instance method*** is a named operation, which is attached to a particular object and is able to access that object's instance variables.
- An ***object class*** (or just *class*) is a family of objects with similar instance variables and identical methods.

In a pure OO language, all instance variables would be private, leaving the instance methods as the *only* way to operate on the objects. In practice, most OO languages (such as Java and C++) allow the programmer to decide which of the instance variables are public and which are private. Anyway, this issue does not affect their representation.

An instance-method call explicitly identifies a particular object, called the *receiver object*, and a particular instance method attached to that object. In Java, such a method call has the form:

$E_0 . I(E_1, \ldots, E_n)$

The expression E_0 is evaluated to yield the receiver object. The identifier I names an instance method attached to that object. The expressions $E_1, \ldots, E_n$ are evaluated to yield the arguments passed to the method.

Although an object is somewhat similar to a record, the representation of an object must reflect the close association between the object and its instance methods. From an object we must be able to locate the attached instance methods. In turn, each instance method must somehow 'know' which object it is attached to.

Example 6.32 Java object representation (single class)

Consider the following Java class:

```
    class Point {
        // A Point object represents a geometric point located at (x, y).

        protected int x, y;

(1)     public Point (int x, int y) {
            this.x = x; this.y = y;
        }

(2)     public void move (int dx, int dy) {
            this.x += dx; this.y += dy;
        }

(3)     public float area () {
            return 0.0;
        }
```