- 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.

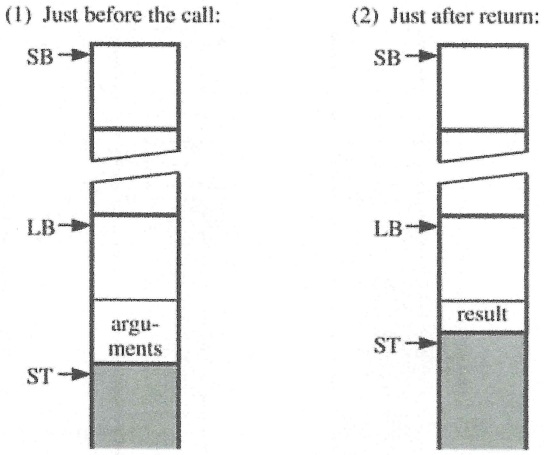

Figure 6.18 The TAM routine protocol.

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).

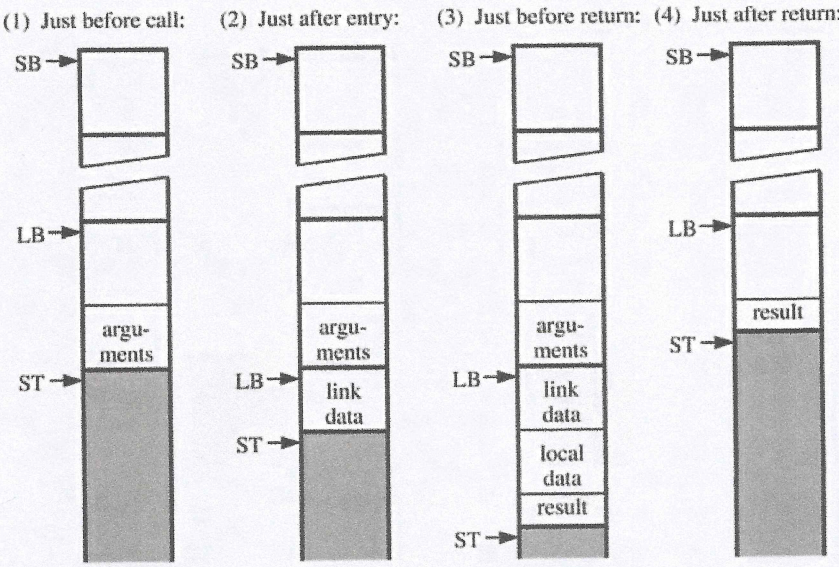

Figure 6.19 TAM routine call and return (in detail).

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 it successor.

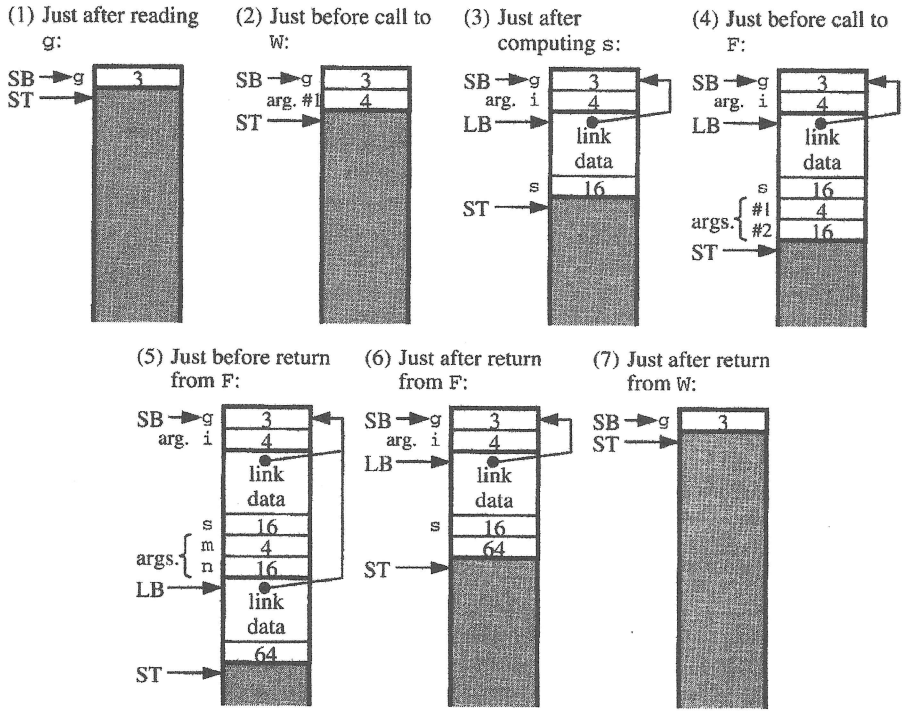

Figure 6.20 Arguments and results in Example 6.20.

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.

□

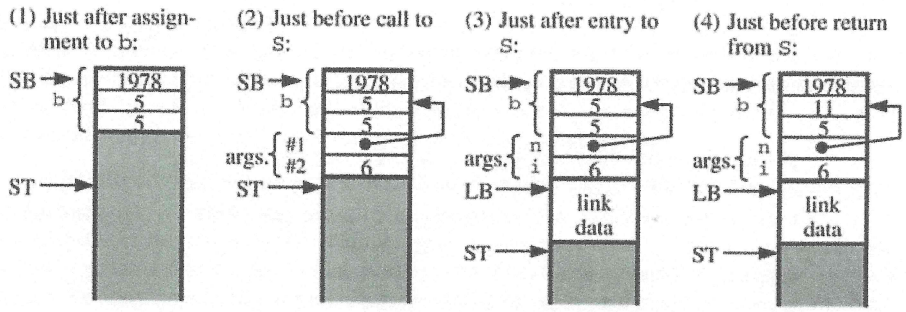

Figure 6.21 Variable and constant parameters in Example 6.23.

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
```

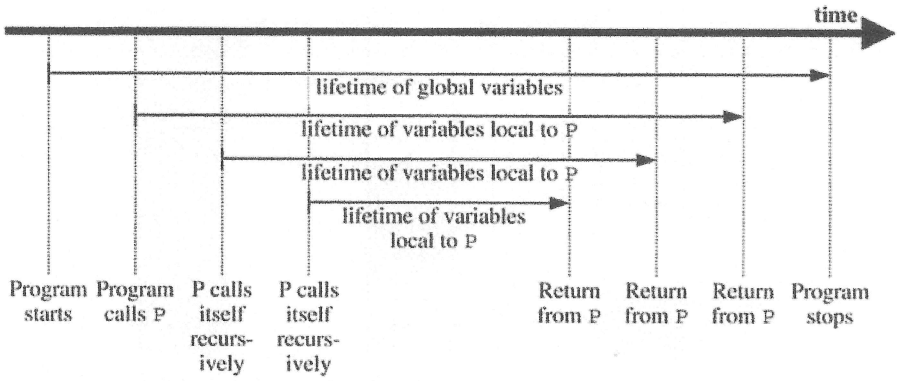

Figure 6.22 Lifetimes of variables local to the recursive procedure of Example 6.24.

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.

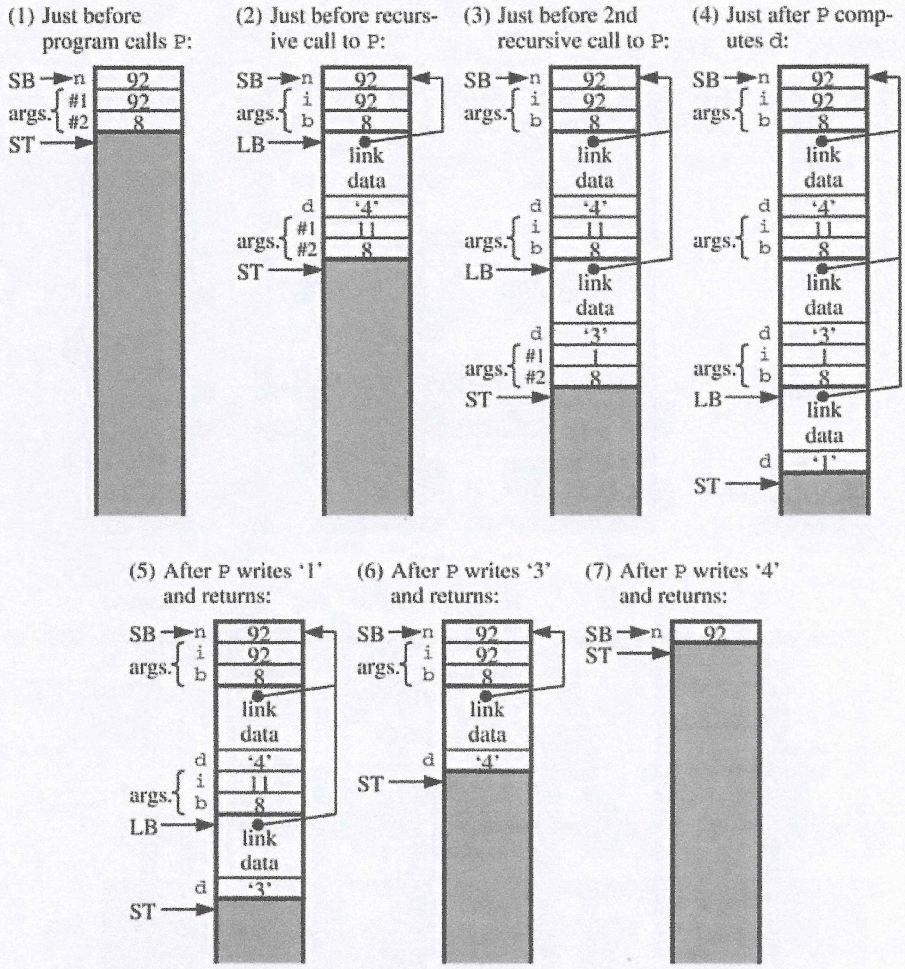

Figure 6.23 Stack snapshots for the recursive procedure of Example 6.24.

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.}
```