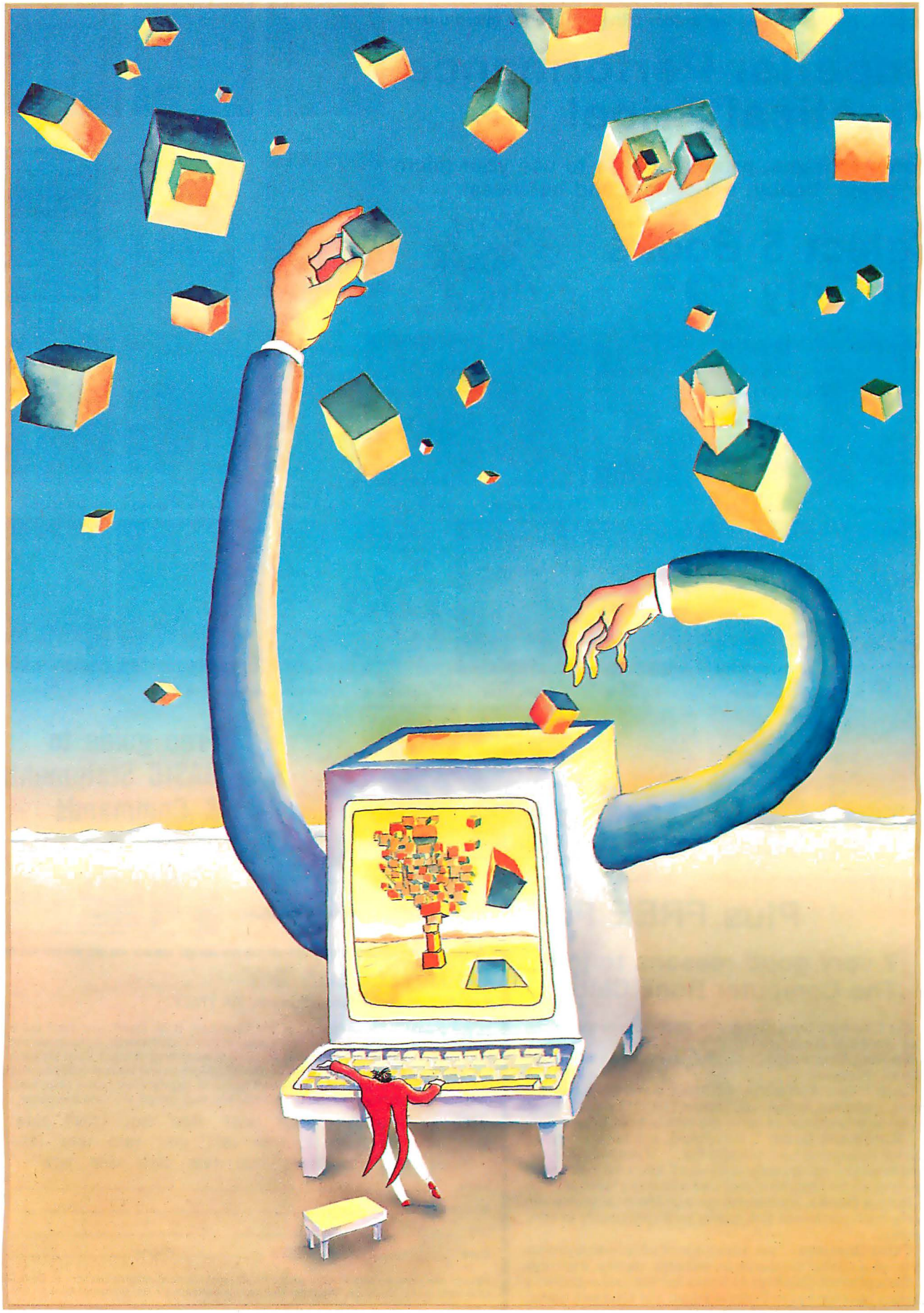
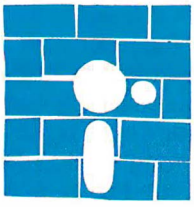


ILLUSTRATED BY JOHN S. DYKES

A SIMPL COMPILER PART 2: PROCEDURES AND FUNCTIONS

BY JONATHAN AMSTERDAM

Procedures and functions are a boon for programmers, but they're tricky to compile



Last month, I described the construction of a compiler for the high-level language SIMPL, but I omitted any description of the part of the compiler that handles procedures and functions. This month, I'll fill that gap.

The SIMPL compiler I wrote translates SIMPL, a Pascal-like language, into VM2 assembly language. VM2 is a hypothetical computer that I wrote a simulator for in "Building a Computer in Software" (October 1985 BYTE, page 112). I described an assembler for VM2 in my November 1985 article (page 112). The routines—my collective term for procedures and functions—of SIMPL are similar to those of Pascal, except that a value is returned from a function using a RETURN statement rather than by assignment to the function name. The syntax of routines is presented in figure 1, and a SIMPL program using a function can be found in listing 1a.

THE CHALLENGE OF ROUTINES

What makes compiling routines so difficult? Listing 1 shows a SIMPL program that calculates the factorial of a number, using a function called fact. The factorial of a non-negative integer *n* is $n*(n-1)*(n-2)*\ldots*1$. The fact function is recursive; it says that the factorial of *n* is equal to *n* times the factorial of $n-1$ and that the factorial of 0 is defined to be 1. To see what has to be done to compile this program, first consider what the run-time behavior of the program ought to be. The following four things have to be done when calling fact.

1. When the statement WRITE(fact(n)) is executed, control has to transfer to the code constituting fact.
2. The argument *n* has to be passed to the function. Somehow, the actual parameter, the value of *n* in the call to fact, must be connected (or bound) to the formal parameter, *n*, that appears in the function definition.
3. It is necessary that fact return to the proper place in the main program and that its result be made available. A function call should act as if it were replaced by its result in the program text. If the call to fact produced the result 6, the program should behave as if the call to fact were simply replaced by the number 6, yielding the statement WRITE(6).
4. Storage has to be found for fact's local variable, temp.

(continued)

Jonathan Amsterdam is a graduate student at the Massachusetts Institute of Technology Artificial Intelligence Laboratory. He can be reached at 1643 Cambridge St. #34, Cambridge, MA 02138.

To handle the control transfer, a simple BRANCH instruction will suffice. If we provide fact with a return address—the memory address of the instruction just after the call—fact will know where to branch to when it's finished. That takes care of transferring control.

How about storage allocation? One solution, often used in FORTRAN compilers, is to allocate enough space with each procedure or function to hold that routine's arguments and locals, plus an additional word of storage for the result of a function. In this case, three words would be allocated: one for the argument *n*, one for temp, and one for fact's return value. The compiler would assign these memory locations while compiling fact; it would remember them in the symbol table and use them to generate references to the argument, local, and return value.

```
routine   ::= proc | func
proc      ::= PROCEDURE id {formals} ; {vars} <routine> block;
func      ::= FUNCTION id {formals} : type ; {vars} <routine> block;
formals   ::= ( <decl ;> decl)
```

Figure 1: *The syntax of* SIMPL *routines. A block is a list of statements surrounded by* BEGIN *and* END*; a decl is a variable declaration; and vars indicate the keyword* VAR *followed by one or more decls. Curly braces around an item indicate that the item is optional. Angle brackets indicate zero or more repetitions of the item are permitted.*

Listing 1: *(a) A* SIMPL *program for calculating the factorial of a number. (b)* VM2 *assembler code generated by the compiler from (a).*

(a)

```
PROGRAM factorial;
VAR n:INTEGER;

FUNCTION fact(n:INTEGER):INTEGER;
VAR temp:INTEGER;
BEGIN
    IF n = 0 THEN
        RETURN 1;
    ELSE
        temp := fact(n-1);
        RETURN n*temp;
    END;
END;

BEGIN
    READ(n);
    WRITE(fact(n));
END.
```

(b)

```
    BRANCH factorial
n:  0
fact:
    SETSP   1
    PUSHL   0, 3        ; n
    PUSHC   0
    EQUAL
    BREQL   L1
    PUSHC   1
    FRETURN 1
    BRANCH  L2
L1:
    PUSHL   0, 3        ; n
    PUSHC   1
    SUB
    CALL    fact, 1
    POPL    0, -1       ; temp
    PUSHL   0, 3        ; n
    PUSHL   0, -1       ; temp
    MUL
    FRETURN 1
L2:
    PUSHC   0
    FRETURN 1
factorial:
    RDINT
    POPC    n
    PUSH    n
    CALL    fact, 0
    WRINT
    HALT
```

This design is simple and elegant. Unfortunately, it does not handle recursion. Because this scheme assigns a fixed amount of memory to each routine, it implicitly assumes that a routine can use only one set of arguments and locals at a time. Each time a routine is called recursively, a new invocation is set up using the same code but different values for the arguments and locals. In the simple scheme above, the values of the first invocation of a recursive routine will be overwritten by the values of the second invocation.

It is necessary to allocate new memory locations each time a recursive routine is called. But it's impossible for the compiler to predict the amount of storage a recursive routine might need, because the compiler can't determine how many recursive calls of a given routine would occur when the program is run. Therefore, this storage allocation must take place at run time, not compile time. You need to decide at compile time how to reference the arguments and locals of the routine and compile the references into the code for the routine. How can this be done?

Activation Records

The solution to this problem involves a data structure called an activation record, which is a contiguous region of memory that contains all the variable information needed for a routine's invocation. It holds the arguments, locals, and a space for the return value for functions. It also holds the return address and some pointers to other activation records I'll describe later. All the activation records for a given procedure have the same format, but their contents differ from invocation to invocation.

The run-time behavior of a program with routines is as follows: Each time a routine is called, storage for a new activation record is allocated. After the activation record is allocated, it is filled with the values of the arguments passed by the call and with the return address. Control then transfers to the called routine. When the routine

(continued)

Listing 2: *(a) A SIMPL program illustrating nested routines. (b) VM2 assembler code generated by the compiler from (a).*

(a)

```
PROGRAM P;
VAR a, b:INTEGER;

  PROCEDURE Q;
  VAR b, c:INTEGER;

    PROCEDURE R;
    VAR b, d:INTEGER;

    BEGIN { R }
      b := 3;
      d := 3;
      WRITE('\n','R',':',' ',a,b,c,d);
      IF c > 1 THEN
        c := c-1;
        R;
    END;
    END;

  BEGIN { Q }
    b := 2;
    c := 2;
    R;
    WRITE('\n','Q',':',' ',a,b,c);
  END;

BEGIN {P}
  a := 1;
  b := 1;
  Q;
  WRITE('\n','P',':',' ',a,b,'\n');
END.
```

(b)

```
    BRANCH  P
a:  0
b:  0
R0:
    SETSP   2
    PUSHC   3
    POPL    0, -2   ; b
    PUSHC   3
    POPL    0, -1   ; d
    PUSHC   '

    WRCHAR
    PUSHC   'R
    WRCHAR
    PUSHC   ':
    WRCHAR
    PUSHC   '
    WRCHAR
    PUSH    a
    WRINT
    PUSHL   0, -2   ; b
    WRINT
    PUSHL   1, -1   ; c
    WRINT
    PUSHL   0, -1   ; d
    WRINT
    PUSHL   1, -1   ; c
    PUSHC   1
    GREATER
    BREQL   L1
    PUSHL   1, -1   ; c
    PUSHC   1
    SUB
    POPL    1, -1   ; c
    CALL    R0, 1
L1:
    RETURN  0
Q:
    SETSP   2
    PUSHC   2
    POPL    0, -2   ; b
    PUSHC   2
    POPL    0, -1   ; c
    CALL    R0, 0
    PUSHC   '

    WRCHAR
    PUSHC   'Q
    WRCHAR
    PUSHC   ':
    WRCHAR
    PUSHC   '
    WRCHAR
    PUSH    a
    WRINT
    PUSHL   0, -2   ; b
    WRINT
    PUSHL   0, -1   ; c
    WRINT
    RETURN  0
P:
    PUSHC   1
    POPC    a
    PUSHC   1
    POPC    b
    CALL    Q, 0
    PUSHC   '

    WRCHAR
    PUSHC   'P
    WRCHAR
    PUSHC   ':
    WRCHAR
    PUSHC   '
    WRCHAR
    PUSH    a
    WRINT
    PUSH    b
    WRINT
    PUSHC   '

    WRCHAR
    HALT
```

returns, the storage for the activation record is deallocated.

How are references to arguments and locals handled? Instead of wiring an absolute address into the routine's code, the compiler generates an offset from the current activation record. The offset is added to the address of the current activation record to get the address of the variable being referenced. Since all activation records for a given routine have the same format, a given offset will pick out the same variable regardless of the invocation.

The current activation record is referenced with a new register I have added to the VM2 machine. This register is called the frame pointer (FP). The FP always points to the current activation record. Each time a routine is called, VM2 needs to save the current value of the FP and set the FP to point to the new activation record. It is convenient to save the old FP in the new activation record. When the routine returns, VM2 sets the FP back to the old value. These manipulations ensure that the FP always points to the activation record of the routine currently being executed.

A new activation record must be allocated on each call of a routine, and it should be freed when the routine returns, otherwise *all* the machine's memory would eventually be consumed. Activation records can be allocated on a stack—the same stack VM2 uses for almost everything else it does—and can be freed by simply popping the stack. In fact, another name for an activation record is a stack frame, from which the name "frame pointer" comes. You may recall from my discussion of stacks in "Building a Computer in Software" that pushing and popping involve little more than incrementing and decrementing the stack pointer. You could hardly hope for a more simple and efficient storage-allocation scheme.

NESTED ROUTINES

The scheme for compiling routines as outlined so far does not handle

(continued)

SIMPL's feature of nested routines. Take a look at the SIMPL program in listing 2a, which makes use of nested routines. It illustrates how you can place the definitions of other routines between the local-variable declarations and the body of a routine, just as you can place routines between the global-variable declarations and the main program body. Nesting affects the scope or visibility of identifiers, that is, which identifiers—variables and routine names—are available to different parts of the program. Let us define the lexical level of a point in the program as its depth of nesting. In listing 2a, global variables are declared at lexical level 0, variables local to procedure Q are at lexical level 1, and variables local to procedure R are at lexical level 2. Then, the rules governing scope in SIMPL are easily stated: A routine has available to it all identifiers declared in the routines that enclose it and none of the identifiers declared in routines nested within it. Furthermore, if two identifiers have the same name, the one at the highest lexical level is the one that is visible to a routine.

Running the program in listing 2a results in the following output:

```
R: 1323
R: 1313
Q: 121
P: 11
```

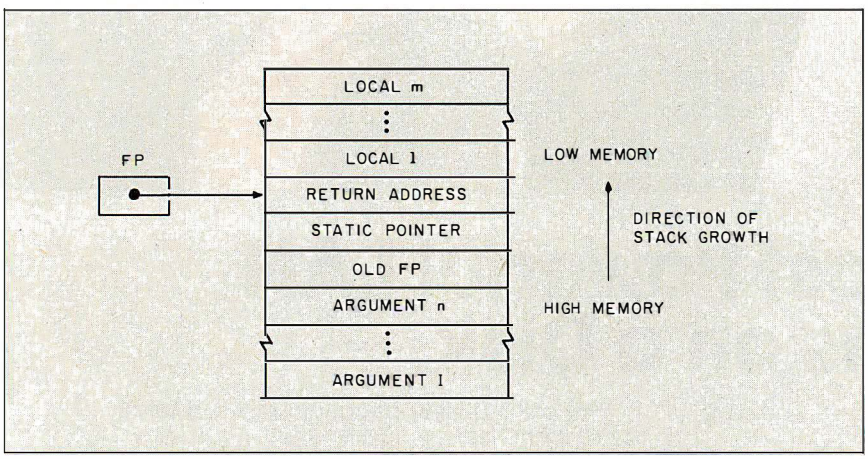


Figure 2: *The structure of an activation record for a routine with n arguments and m local variables.*

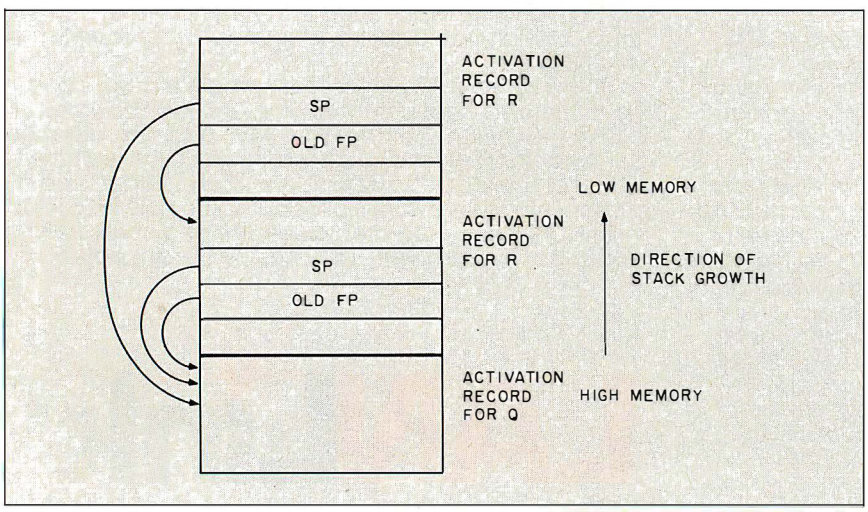


Figure 3: *The structure of the stack when* Q *calls* R *and then* R *calls itself showing the static* (SP) *and dynamic* (FP) *pointers.*

Procedure R, being the innermost procedure, can access the global variable *a*; the variable *c*, which is local to procedure Q; and its own local variables *b* and *d*. R can also call both itself and the procedure in which it is nested, Q (R does not call Q in this example). Q cannot access any of R's variables, but it can access *a* and its own locals, and it can call R. The main program can access only global variables and can call Q. The variable *b* provides an example of how variables with the same name hide, or shadow, one another. Each of the three occurrences of *b* in the program refers to a different variable. The appearances of *b* within R and Q refer to local variables of those procedures and have the values 3 and 2, respectively. The occurrence of *b* in the main program refers to the global variable *b*, and its value is 1.

R's access to *c* causes a problem for the routine-calling scheme I outlined above. If *c* were a global variable, it would be accessible directly by name; if it were local to R, it could be found at some fixed offset from the FP. But *c* is neither local to R nor globally visible from P. I have not indicated how such nonglobal nonlocal variables can be accessed.

Before I proceed to the solution, note that at the time it is accessed by R, the variable *c* must be residing somewhere on the stack because, by the visibility rules discussed above, Q has to be called before R can be; it is only from within the definition of Q that R is visible at all.

You may recall that the activation record for R contains the value of the FP for R's caller. In this case, R's caller is Q, so the old FP is a pointer into Q's activation record. It would seem you need only follow the old FP to get to nonlocal nonglobal variables.

This will not work, however, because other routines besides Q can call R; in particular, R can call itself. In this case, the old FP for the second invocation of R points to the activation

record for the first invocation of R, not to Q's activation record, so following the old FP would not get us to Q, but merely to another copy of R. We would need to follow the chain of frame pointers back twice to get to Q.

Static Pointers

In essence, the problem with following the FPs to find nonlocal nonglobal variables is this: The saved FPs indicate the dynamic structure of the program, its run-time behavior: who calls whom. To find variables, the so-called static structure is needed: who's defined inside whom.

The solution I have adopted is to maintain a static pointer (SP) in each activation record in addition to the value of the caller's FP (sometimes called the dynamic pointer). The SP always points back to the most recent activation record of the routine in which the current routine was defined; for instance, the SP in R's activation record always points to an activation record for Q, regardless of who called R. The activation-record format for a routine with *n* arguments and *m* local variables is shown in figure 2. Figure 3 illustrates the structure of the stack when static and dynamic pointers are used. Note that it is sometimes necessary to follow several static pointers to get to the desired variable. For example, if a procedure S were defined inside R and accessed the variable *c*, the SP in S's activation record would be followed, leading to an activation record for R; then, R's SP would be followed, leading to the desired activation record for Q. The number of static pointers to follow is the difference in lexical levels between the point of call and the callee.

Calling Mechanism in Action

Now that all the pieces of the routine-calling scheme have been described, let's put them into place by seeing what happens when the program in listing 2 is executed. You may want to glance at figure 4 during this discussion.

The main program begins by calling Q. First, the current value of the FP is pushed, followed by the SP, and the FP is set to the current value of the stack pointer. Since Q is called from the main program, it is not necessary to save the FP on the stack or to compute the SP, but I do it anyway since it's easier to implement this calling mechanism if a call from the main program isn't treated as a special case. Next, the return address, which can be calculated from the value of the program counter at the time of the call, is pushed onto the stack, and the computer branches to the beginning of Q (see figure 4a).

Q begins by pushing two zeros onto the stack. This serves to allocate a word on the stack for each of Q's local variables and at the same time to initialize those variables to 0. The body of Q begins execution by setting its local variables, *b* and *c*, to 2. Then, R is called by the same mechanism as before: First, the FP is pushed onto the stack, the SP is computed by following the chain of static pointers as many times as the difference in lexical levels between the point of call and R, and the value of the FP for the activation record at that place in the stack is pushed. Since the definition of R is at the same lexical level as the body of Q, no static pointers need be

(continued)

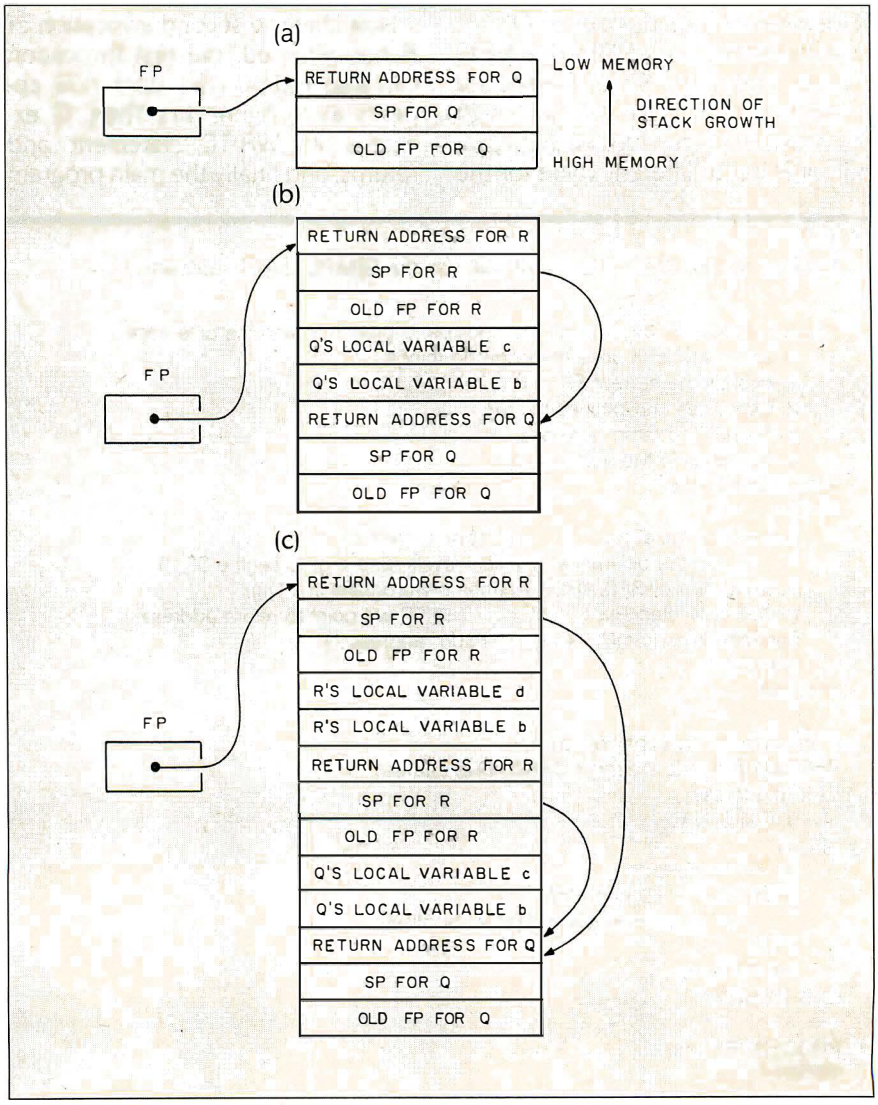


Figure 4: *The run-time behavior of the program in listing 2. See the text for details.*

followed, and the FP value for Q is pushed as the SP for R's activation record. FP is then set to the stack pointer, the return address is pushed, and control transfers to procedure R (see figure 4b).

After pushing and initializing its local variables, R executes its WRITE statement, then tests the value of Q's local variable *c*. Since Q set *c* to 2, the statements within the IF statement are executed. First, *c* is decremented, then R is called recursively. To begin the recursive call on R, the FP is again pushed onto the stack and the new SP calculated. Now, since this activation record for R is one lexical level deeper than R's definition, a single SP is followed; this leads back to Q's activation record, so the FP value for Q is again used as the SP for this second invocation of R. Note that although the activation record for each invocation of R has a different value for the old FP, they have the same value for the SP. Next, the FP is set to the current value of the stack pointer, the return address is pushed, and control transfers to the body of R for the second time (see figure 4c).

In the second invocation of R, R's local variables are pushed onto the stack and then the WRITE statement is again executed. R tests *c*, but this time it is not greater than 1, so the code within the IF statement isn't executed. The return process is the inverse of the call: The stack pointer is set back to where it was before the call, and the FP is restored to its old value. At this point, the stack again looks as it does in figure 4b.

Now that the second invocation of R has returned, the first invocation can also return. (The stack now appears as in figure 4a.) Then, Q executes its WRITE statement and returns, and finally the main program does a WRITE and the program ends.

What I've just described differs in two minor ways from the scheme as I originally presented it. First, although for reasons of conceptual simplicity I described the activation record as being allocated all at once, it in fact is allocated piecemeal, a push at a time: the arguments (although in this example there were none), the FP, the SP, and the return address. Second, it's somewhat more convenient for my purposes to have the FP point to the middle of the activation record instead of to the beginning. This means that some offsets from the FP will be negative and others positive.

Listing 3: *The Modula-2 source code for the* SIMPL CALL *statement.*

```
(* CALL takes two arguments, the address to branch to and the difference
    in lexical levels. It does the following things:
    1. Pushes the current FP
    2. Computes and pushes the SP
    3. Pushes the return address
    4. Branches to the address. *)
PROCEDURE call;
BEGIN
    pushWord(framePtr);          (* save current FP *)
        (* use the difference in lexical levels (2nd arg) to set the SP *)
    pushWord(followSP(CARDINAL(memory[programCtr + 1])));
    framePtr := stackPtr;              (* FP will point to return address *)
    pushWord(programCtr + 2); (* return address *)
    branch;
END call;

(* Follows the static-pointer chain. *)
PROCEDURE followSP(num:CARDINAL):address;
VAR fp:address;
    n:CARDINAL;
BEGIN
    fp := framePtr;
    FOR n := 1 TO num DO
        fp := address(memory[fp + SPoffset]);
    END;
    RETURN fp;
END followSP;

PROCEDURE branch;
BEGIN
    programCtr := address(memory[programCtr]);
END branch;
```

Some New Instructions

A compiled program's code would be long and messy indeed if it had to worry about every manipulation of static pointers and activation records. Instead, I'm going to push all this complexity down into the virtual machine, VM2, and hide it behind five new VM2 instructions.

The first and most complicated is CALL, which takes two arguments: the memory address of the beginning of the routine's code and the difference in lexical levels between the caller and callee. It performs all the operations necessary when one routine calls another: saving the return address and FP on the stack, setting the SP, setting the FP register, and branching to the routine. Because CALL is so complex, I have provided the Modula-2 source code for it in listing 3.

The instructions PUSHL and POPL are used to access all but global variables; the "L" is for "local." They each take two arguments: the difference in lexical levels between the variable and the accessing routine and the offset of the variable. Each follows the chain of static pointers a number of times equal to the difference in lexical levels and then uses the offset to access the variable. PUSHL pushes the value of the variable onto the stack; POPL pops the top of the stack into the variable.

(*continued*)

Two instructions, RETURN and FRETURN, handle returns. Both take one argument, the number of words of actual parameters (arguments) pushed onto the stack by the caller. They need this value to determine where to set the stack pointer. RETURN merely sets the stack pointer to where it was before the call, effectively popping the activation record off the stack. FRETURN (function return) first pops the top of the stack, which should contain the value to be returned by the function, then resets the stack pointer as with RETURN, and finally pushes the returned value back onto the stack.

To get a sense of the code generated by my compiler, you may want to look at listings 1b and 2b; they show the compiler's output for listings 1a and 2a, respectively.

Compiler Issues

Paradoxically, I have spent nearly all the second part of the compiler project describing a mechanism that is implemented in VM2. Of course, the mechanism would have been unnecessary were it not for the peculiar problems that arise in compiling high-level languages with nested procedures and functions. But it is now time to move to the compiler proper.

The basic action of the compiler when it sees a routine is as follows: First, the routine name is entered into the symbol table. Then, the list of formal parameters is parsed; each formal parameter is entered separately into the symbol table, and the whole list of formals is attached to the routine's symbol-table entry as well to aid in checking calls to make sure they supply the right number and types of arguments. If the routine is a function, its type is then parsed and placed in the routine's symbol-table entry. Next, the local variables are parsed and entered into the symbol table. The compiler's routine-compiling procedure then calls itself recursively to handle any nested routines.

Finally, the body of the routine is compiled. The compiler first outputs a label, which is the routine's name. Then, the code to place the local variables onto the stack is generated. I do this by outputting a PUSHC 0 instruction for each local; as I said earlier, it has the effects of allocating a word on the stack and initializing the variable to 0. Lastly, the code for the body is generated. In SIMPL, if no RETURN statement is executed in a procedure, that procedure returns after its last statement is executed; to handle this, the compiler needs to generate a RETURN instruction after the code for the procedure. Functions, on the other hand, have to return values explicitly. It should be an error if they don't.

A few things are needed to embellish this basic compiling process. First, the compiler needs to remember the lexical level at which each identifier in the program is defined. It does this by means of a counter, lexicalLevel, which starts at 0, is incremented whenever a routine definition occurs, and is decremented when the compiler has finished compiling a routine. Each time a routine name or variable is defined, the current lexical level is stored with it in its symbol-table record. In order to get the visibility of routine names right, the counter must be incremented just after the routine name is seen but just before the formals are. Formals are treated as being local to the routine in which they occur.

Second, formals and locals need to be given offsets from the FP. If you take a look at the form of an activation record in figure 2, you'll see that the first local variable is one word below where the FP points to, so it should be given an offset of −1. The second local should be given an offset of −2, and so on. Things are a bit more tricky with formals, however. The compiler handles the arguments in a routine call from left to right, pushing the first argument onto the stack first. Hence, the first argument will be farthest from the FP, so it should have the highest offset. To assign offsets to formals, the compiler must read them all in first, count how many there are, then go back through them and assign the offsets. Because

(continued)

of the way I've set up the activation record, the last argument—the one closest to where the FP points—will have an offset of 3.

One final consideration is that after the compiler is done with a routine, all identifiers local only to that routine should be removed from the symbol table. This is so that a later part of the program can't possibly succeed in referencing one of these identifiers. I'll now describe how to compile the various constructs that arise in dealing with routines.

Routine Calls

To compile a procedure or function call, the arguments are treated as expressions and each is compiled. When an expression is compiled, code is generated that will result in the value of the expression being left on the stack at run time, so compiling the arguments as expressions is just what the routine-calling mechanism requires. After the arguments are compiled, a CALL instruction is generated with the name of the routine being called and the difference between the lexical level of the called routine and the current lexical level. The compiler also performs several checks: The called routine must be a function if the call occurs in an expression, otherwise it must be a procedure; and the number of arguments and their types must match with the list of formal parameters.

SIMPL RETURN Statement

When the compiler sees a RETURN statement followed by an expression, it checks to make sure it is in the process of compiling a function; if so, it generates code for the expression (which will result in the expression's value being pushed onto the stack at run time) and generates an FRETURN instruction. When the compiler sees a RETURN statement with no following expression, it makes sure it is compiling a procedure, then it generates a VM2 RETURN instruction.

Variable Access

When a variable is used in the code, the compiler looks it up in the symbol table. If it is global, its name is used. If not, a PUSHL or POPL instruction is generated, as appropriate, with the variable's offset and the difference in lexical levels between the current one and the one in which the variable was defined.

Name Management

Two minor problems remain for the compiler, both having to do with managing the names of identifiers. The first one concerns routine name clashes. Say you have two routines, P and Q. Inside P you can define another routine, R, and inside Q you can also define a routine called R. The problem is that you can't use the routine names as labels in the assembly-language program, since then you would have two "R" labels, and that's illegal in my assembler. The easiest solution is to generate a new label for every routine and record the label in the routine's symbol-table entry for use when the routine is called.

(continued)

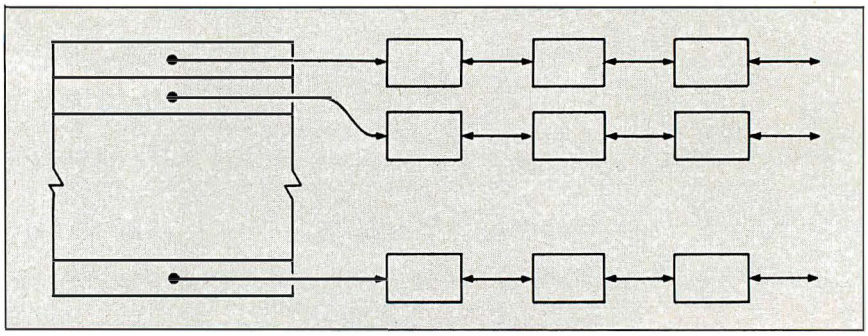


Figure 5: *The structure of the symbol table: an array of pointers to doubly linked lists of symbol-table entries.*

The second problem concerns the process of looking up an identifier in the symbol table. Recall that the second scoping rule states that when an identifier that is declared in a nested routine is also declared inside the routine in which it is nested, the innermost identifier shadows the other. So the *b* referred to by procedure R in listing 2a is the variable local to R, not the ones local to Q or P. How can we implement the identifier lookup routine so that this scoping rule is enforced?

The obvious solution is to examine all the identifiers with the same name and choose the one defined at the highest lexical level. This solution will work, but a simpler one suggests itself if you notice that identifiers in lower lexical levels are declared before those in higher ones. That is, as the compiler reads the program from top to bottom, it will first install global variables into the symbol table, then variables at lexical level 1, and so on. If the symbol table were merely a list of entries, and if new entries were inserted at the beginning of the list, the lookup routine could simply take the first identifier whose name matched the one being looked up; since that identifier was the most recently inserted of all those with the same name, it must have been defined at the highest lexical level.

In practice, though, a single list is too inefficient a representation for a symbol table—the lookup time is proportional to the length of the list, and if there are many identifiers, the list will be long. It would be great if the symbol table could combine the efficiency of a hash table with the nice lookup property of a list. That's possible if each element of the hash table, instead of containing a single symbol entry, contains a pointer to a list of entries. Instead of one long list, the symbol table consists of an array of shorter lists; and since identical strings hash to the same location in the array, all the identifiers with the same name will be on the same list. The lookup routine hashes the name of the identifier it is searching for, indexes the array to find the appropriate list, and searches the list in order, taking the first match it finds. To facilitate the removal of entries, the list is doubly linked. The structure of the symbol table is illustrated in figure 5.

CONCLUSION

The Modula-2 source code for my SIMPL compiler, including the code to handle routines, along with the VM2 assembler and VM2 monitor, are available for downloading from BYTEnet Listings. The telephone number is (617) 861-9764. In part 3 next month, I'll extend the compiler by adding some useful features like arrays. ■