Some programmers question certain claims made for threaded-code systems. The author defends these claims by tracing such a system from its origin in simple concepts.

An Architectural Trail to Threaded-Code Systems

Peter M. Kogge, IBM Federal Systems Division

Interest in software systems based on threaded-code concepts has grown remarkably in recent years. Advocates of such systems regularly make claims for them that many classical programmers regard as bordering on the impossible. Typically, proponents claim that it is possible to construct, in 5K to 10K bytes of code, a software package that

- is conversational like APL, Lisp, or Basic;
- includes a compile facility with many high-order language, structured-programming constructs;
- exhibits performance very close to that of machine-coded program equivalents;
- is written largely in itself and consequently is largely portable;
- places no barriers among combinations of system, compiler, or application code;
- can include an integrated, user-controlled virtual memory system for source text and data files;
- permits easy user definition of new data types and structures; and
- can be extended to include new commands written either in terms of existing commands or in the underlying machine language (an assembler for the underlying machine can be implemented in as little as a page of text of existing commands).

Such systems do exist—Forth is perhaps the best known—and have attracted widespread attention, as evidenced by countless implementations, publications, products, and standards groups.

The purpose of this article is to convince the skeptic that these claims are, in fact, achievable without resorting to obscure software magic or arm-waving. The approach is to build a logical (not historical) trail from some rather simple concepts to the outlines of a software system that meets the above claims. A series of generalizations added to a simple view of software leads to the final result, a package approaching the combination of

- a simple instruction-set architecture, or ISA;
- the concepts and syntax of a high-order language, or HOL; and
- a set of commands for a conversational monitor.

The ISA (such as it is) is for an abstract machine that is very efficiently implemented by short code segments in almost any current machine architecture. The HOL concepts include hierarchical construction of programs, block structures, GOTOless programming, and user-definable data structures. The commands for the conversational monitor permit direct interaction with objects defined by the programmer in terms he defines, not simply in the terms of the underlying machine (i.e., core dumps). Further, the command set is directly executable in a conversational mode and includes virtually all commands available in the system; when so executed, they function just as they would if found in an executed program. Finally, programs defined by the programmer are automatically part of the system's set and can be used in any combination with other commands in either a conversational or program-definition mode. This permits a programmer to develop application-oriented interfaces that are compatible with and run as efficiently as the basic monitor commands.

Much argument exists as to whether these results define a machine, a language, an architecture, a system, etc. We sidestep this issue by declaring it to be a self-extendable software package.

Discussion base

One of the major tenets of modern software development is the value of modular, structured solutions to problems. In such solutions, the final product is a hierarchy of procedures or subroutines, each with well defined interfaces and short, easily understood bodies that exhibit minimum side-effects.

When written in either assembly or higher-order language, such structured applications tend to look like this:

0018-9162/82/0300-0022$00.75 © 1982 IEEE

```
APPLICATION    CALL        INITIALIZE
               CALL        INPUT
               CALL        PROCESS
               CALL        OUTPUT
                      .
                      .
                      .
INPUT          CALL        OPEN
               CALL        READ
               CALL        CLOSE
                      .
                      .
                      .
PROCESS        CALL        STEP 1
               CALL        STEP 2
               CALL        STEP 3
```

(Code that handles passing of operands has been deliberately ignored for the present; it will be discussed later.)

The key thing about such "programs" is that they largely consist of addresses of procedures, with each address preceded by a CALL opcode.

Generalization 1: Removal of opcode. Consider first programs at the next-to-last level of such a hierarchy, those in which all called routines are actually pure machine code with no further nested calls. One way to simplify representation of such programs is to replace the list of call instructions with a simple list of addresses; a very small machine-language routine would go through this list sequentially, making indirect branches at each step. This clearly reduces storage at some cost in performance but, by itself, seems to yield nothing else. The benefits of this move will be brought out in later generalizations.

As an example, assume that Steps 1 to 3 above are pure machine code. Thus, PROCESS could be replaced by:

```
PROCESS   I←Address of pointer to
          first step (i.e., PLIST)
          Branch to NEXT
PLIST     DW STEP 1
          DW STEP 2
          DW STEP 3
             .
             .
             .
STEP i    Code
          Branch to NEXT
NEXT      W←Memory(I)        Machine code to start se-
          I←I+1              quencing through list. I
          Branch to (W)      points successively to each
                             entry.
```

The code at the beginning of PROCESS is termed its *prologue*. In each of the called steps, any RETURN instructions would be replaced by a branch to NEXT.

In the literature, this threading of a sequence of subroutines into a list of their entry addresses is termed direct threaded code, or DTC. The small routine NEXT is termed an *address interpreter* (it has also been called an *inner interpreter*). In many architectures, it is a single "jump indirect with post increment" instruction.

By convention, *I* and *W* (as well as *X, Y,* and TABLE, to be introduced later) are assumed to be registers or dedicated memory locations of the underlying machine. Through the address interpreter sequences of the machine, these registers or locations direct the machine to carry out the appropriate procedures.

Register *I* points to the next address in the list of procedures to be executed, and *W* contains that address. Figure 1a summarizes their relationship.

Generalization 2: Higher levels in the hierarchy. This same technique can be used at higher levels in the hierarchy by using a stack to keep track of *I* values and by preceding each procedure that is above the next-to-last level with a piece of code that stacks the current value of *I* and resets it to the new list. A new entry is added at the end of each procedure list; it points to a routine that pops the stack to retrieve the *I* value of the procedure's caller.

```
INPUT     Push I to STACK      New prologue
          I←NEW LIST
          Branch to NEXT
NEW LIST  DW OPEN              Define each word
          DW READ              as an address
          DW CLOSE
             .
             .
             .
          DW RETURN
RETURN    Pop from STACK to I
          Branch to NEXT
```

This stack is often called the *return stack.*

This technique effectively reverses the position of the information that tells the machine it is entering a new procedure. In the classical approach, the opcodes associated with the addresses are interpreted *before* the change to a new procedure. In the above approach, the prologue code serves the same function, but is invoked *after* the address in the list has been used. What has been defined is, in effect, a mechanism for a *called* routine to determine what kind of routine it is and how to save the return information; in contrast, the classical approach demands that the *calling* procedure provide this information. In a sense, the machine does not know what kind of routine it is entering until it gets there. This self-definition capability is the keystone for many of the capabilities to be discussed below.

Generalization 3: Commonality of prologue. As defined above, every single procedure constructed from a list of calls to other procedures must have its own carbon copy of the prologue code. A programmer's typical response to such a situation is to look for ways of avoiding this duplication. In one common approach, he stores at the entry point of the procedure not a copy of the prologue code, but the *address* of a single routine that carries out the function of the prologue code. Thus, all procedures that use the same prologue start with an address that points to the actual routine.

Placement of an address rather than actual code at the entry to a procedure requires a change in the address interpreter. Instead of branching directly to the start of the next procedure, the address interpreter must branch indirectly through the first word of that procedure. This corresponds to something like the following:

```
NEXT          W←MEMORY(I)
              I←I+1
              X←Memory (W)
              Branch to (X)
```

If a procedure really is to be implemented in machine code, the address at its entry typically points to the very next word.

Figure 1b summarizes the relation between *I, W,* and *X,* and Figure 2 diagrams a fairly complex case. The prologue is called ENTER; EXIT is the procedure to return to the caller. One important aspect of this is that most of the code is largely machine-independent. Only at the lowest level do we see real machine/architecture dependencies. All other levels use addresses, which can conceivably be transported unchanged to another machine with an entirely different architecture.

Code written for this new address interpreter is often called *indirect threaded code,* or ITC. It is slightly slower than the earlier version, but provides savings in storage and machine independence and permits some of the interesting generalizations discussed below. If more speed is desired, this function and that of any of the other address interpreters can be microcoded or added to the hardware of the underlying machine.

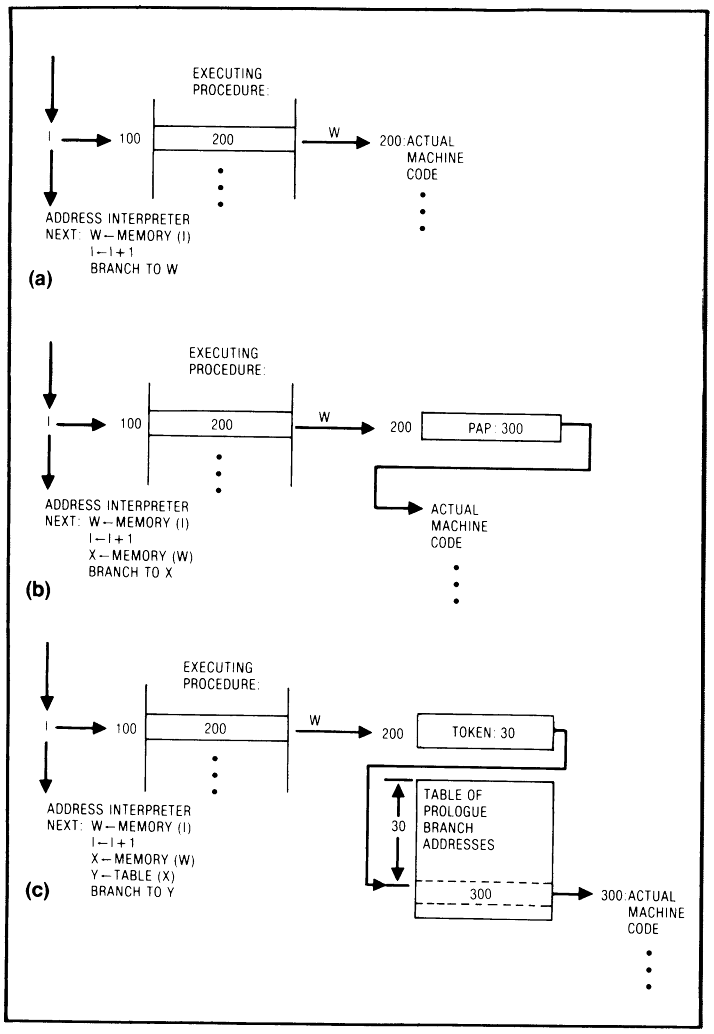

Figure 1. Types of address interpreters (or, Where is the machine code now?): (a) direct threaded code, (b) indirect threaded code, and (c) indirect token-threaded code.

The word addressed by *W* is the *prologue address pointer,* or PAP (very often referred to as the *code address word* or CAW). Again, *W* is the address of the PAP. In the above description of NEXT, *X* is the contents of the PAP. This value is the address of the prologue machine code for the new procedure.

A variation of this approach replaces this entry address PAP with a *token,* which serves as an index into a table of possible prologue codes. This adds one more level of indirection to the address interpreter, but provides even more machine independence to the total program. The technique is often called *indirect token-threaded code,* or ITTC. Figure 1c diagrams it in more detail. Table 1 compares DTC, ITC, and ITTC.

Generalization 4: A parameter stack. These lists of addresses are beginning to resemble a program in an ISA, where each instruction consists of only an opcode—the address of a routine. What is needed to help complete this ISA is some way to pass operands. The simplest such mechanism—one used in many other ISAs that have no explicit operand specifiers—is a second stack. Any instruction that needs operands as inputs takes them from the top of this stack; any data produced goes back onto the stack, which is commonly called the *parameter stack.* It is distinct from the return stack discussed above.

As a trivial example, the machine code for a procedure to add two numbers looks like this:

```
ADD DW * + 1           Prologue address pointer
    Pop PSTACK to Z    Get operand
    Pop PSTACK to Y
    Add Z to Y         Do operation
    Push Y to PSTACK   Save result
    Branch to NEXT
```

The PAP points to the actual machine code. This, in turn, pops two values off the stack, adds them, and returns the result. The asterisk marks the address of the current location Thus, * + 1 points to one location beyond the PAP.

Many actual stack-oriented ISAs keep all information on a single stack; the advantage of two stacks lies in both simplicity of implementation and the reduced conceptual complexity they present to a human programmer. Operands are always found on the parameter stack and procedure call information on the return stack. The advantage of this will become increasingly apparent to the reader in the following sections.

Generalization 5: Reverse Polish notation. Existence of a parameter stack permits use of reverse Polish notation,

Table 1.
Comparison of DTC, ITC, ITTC.

TYPE OF ADDRESS INTERPRETER	VALUE OF INTERPRETER REGISTERS I	W	X	Y	ADDRESS OF EXECUTED MACHINE CODE
DTC	100	200	—	—	200
ITC	100	200	300	—	300
ITTC	100	200	30	300	300

or RPN, for specification of series of operations. In the written form of such a notation, operands come before the operations and evaluation proceeds left to right, one operation at a time. There are no parentheses, and no precedence is given to one operator over another. This notation is used in solving problems on many calculators.

As an example, consider the polynomial evaluation

$$AT^2+BT+C=(AT+B)T+C$$

When expressed in a written RPN, this becomes

$$A\ T*B+T*C+$$

Such an expression is read from left to right in the following manner:

(1) Put A on the stack.
(2) Put T on the stack.
(3) Multiply the top two entries (A and T).
(4) Put B on the stack.
(5) Add the top two entries (AT and B).
(6) Put T on the stack.
(7) Multiply the top two entries ($AT+B$ and T).
(8) Put C on the stack.
(9) Add the top two entries ($(AT+B)T$ and C).

If we could come up with a way of defining threaded-code procedures to access A, B, C, and T, the above computation could be carried out by a series of nine addresses in exactly the above order. No general registers, accumulators, or other constructs would be needed. This simplification is the key to maintaining the concise user interface described below.

Generalization 6: Named entities. Despite the advantages of parameter stacks and reverse Polish notation, in many classical programs and programming languages a need remains for binding the values of named variables, arrays, records, etc., to explicit storage locations that are less transient than entries on a stack. These can be added to our developing threaded-code architecture by defining special prologue routines that use information from the address interpreter, namely address W of the PAP, to identify unique storage locations.

As an example, a procedure that delivers a constant to the parameter stack might have two entries: the PAP pointing to a special prologue and a word holding the desired value. The prologue code might look like this:

```
CONSTANT   Z←Memory(W + 1)    Get the value
           Push Z to PSTACK   Push it
           Branch to NEXT
```

Again, as many constants as needed can be defined. Each will have its unique value, but will contain a PAP pointer to the same prologue.

Cases in which the values change frequently can be handled similarly; such structures are called variables. Here the prologue code stacks not the contents of location $W+1$ but its address:

```
VARIABLE   Z←W + 1
           Push Z to PSTACK
           Branch to NEXT
```

To find the value of a variable, a second procedure must be called. This procedure, often called @ (AT), uses the top entry of the stack as an address and replaces it with

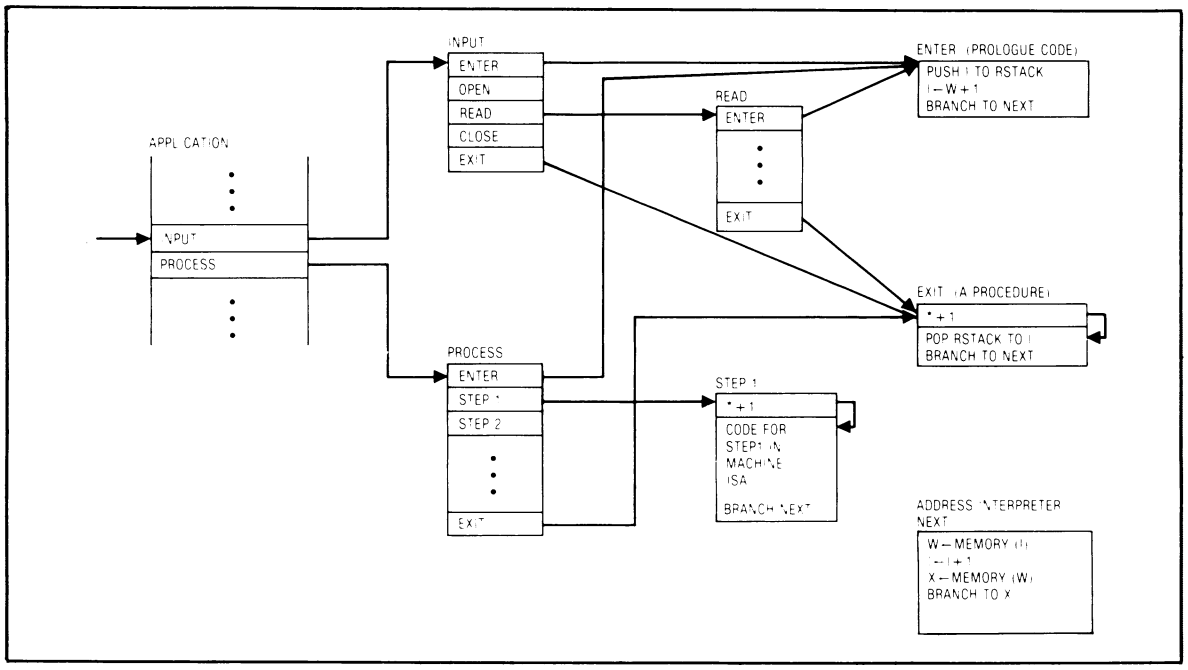

Figure 2. Indirect threaded code. Asterisk marks address of this memory location.

the value of that location. If defined as a machine-coded primitive, it might look like this:

```
AT    DW * + 1              PAP
      Pop PSTACK to Z
      Y←Memory (Z)
      Push Y to PSTACK
      Branch to NEXT
```

Changing the value of a variable also requires a new procedure, often called ! (STORE). This procedure removes two operands from the parameter stack; one is the value; the other the address, as delivered by an earlier VARIABLE. Defined in machine code, it would look like this:

```
STORE    DW * + 1              PAP
         Pop PSTACK to Z    Get address
         Pop PSTACK to Y    Get new value
         Memory (Z)←Y        Save it
         Branch to NEXT
```

The important concept behind CONSTANT and VARIABLE, as defined above, is that the access to a data structure can be defined as a prologue code that is both addressed by the PAP and works on a data area starting at location $W+1$ (one word beyond the PAP). Under this definition, it is possible to define any kind of a data structure. For example an array procedure could use the top of the parameter stack as an index, the contents of $W+1$ as a dimension limit, and $W+2$ as the start of the body of the array. The prologue code referenced at the beginning of an array definition might look like this:

```
ARRAY   Pop PSTACK to Z       Get index
        If Memory(W + 1) ≤ Z
         then error
        Y←W + 2 + Z           Get address of array (Z)
        Push Y to PSTACK      Save it
        Branch to NEXT
```

Extensions of this allow a user to define any kind of a data structure he might need and integrate it very cleanly with the rest of the architecture via the common parameter stack.

A detailed example

At this point, we review the concepts developed above by illustrating them in a detailed example. Figure 3 diagrams a hierarchy of procedures and codes needed to support polynomial evaluation as defined earlier. Assume that, at entry, T is at the top of the parameter stack, A is a constant, B is a variable, C is the third element in an array, and the result is to be placed in the variable Z.

Generalization 7: Conditional operators. So far, we have included no capability for decision-making and conditional processing. While this is not hard to add, there are several possibilities to consider. The simplest and most common is to invent a branch-on-zero procedure. This rather elementary operation tests the top of the parameter stack, and if it is zero, replaces the current value of I by the sum of I and the contents of the word following the original call to the branch procedure. If the top of the stack is nonzero, the operation increments I over the displacement. If the ITC address interpreter is used, then I, in the body of branch, points to the displacement directly.

```
  .
  .
  .
I BRANCH0→  * + 1                 \   PAP
  DISP      Pop PSTACK to Z        |  Machine
   .        If Z = 0 then          |  code
   .        I←I + Memory(I)        >  for
   .        else I←I + 1           |  Branch on 0.
            Branch to NEXT        /
```

The above branch and its variations are equivalent to what is found in most classical ISAs. Another form, particularly useful for loops, is a conditional procedure exit. It pops the return stack into I only if the top of the parameter stack is zero. Otherwise, the return stack is left untouched. Coupled with this are procedures to mark the return stack with an address for future use by this conditional return.

Yet another form would be a conditional procedure call:

```
  .
  .
  .
I CALL0→     * + 1
  F00        Pop PSTACK to Z.
   .         If Z = 0 then
   .         Push I + 1 to RSTACK
   .         I←Memory(I)
             else I←I + 1
             Branch to NEXT
```

Here, F00 is executed only if the top of the parameter stack is zero.

Variations of this approach handle IF-THEN-ELSE and similar constructs directly, with the bodies of the THEN and ELSE defined as separate procedures.

Generalization 8: Iterative loops. The constructs described above are sufficient for handling not only IF-THEN-ELSE structures, but also DO-WHILE and UNTIL-DO forms of loops. Iteration loops (e.g., DO K = 1, 10) can also be handled, but variables or some other construct must be used to hold the iteration variable (the one whose value changes once for each iteration of the loop).

A rather elegant construct for iterative loops uses the return stack to hold the iteration variable and its limits. Before entering the loops, the appropriate values are pushed onto the return stack; after each iteration, the values on top of the return stack are incremented by the appropriate amount and compared with limits. If the loop is to be executed, the modified iteration value remains on the return stack. Completion of the loop causes these entries to be removed from the return stack. Such a construct is inherently reentrant and nestable to any level.

Given that procedures exist for moving values between the two stacks, any of the mechanisms described above can be used for the test and branch back. Alternatively, the whole process of loop control can be buried in a single procedure that does iteration variable modification, test, and branch back.

Access to the iteration variable can be obtained by simple procedures that copy from the top of the return stack to the parameter stack. Such facilities are provided in many threaded-code systems.

Partial summary. What we have so far is the outline of the architecture of a simple ISA that has

- an indirect code access mechanism,
- two stacks,
- reverse Polish notation,
- user-definable data structures,
- extensible operations,
- a hierarchical framework, and
- the ability to directly support many HOL constructs.

An application written in such a system would consist of

- a small set of small prologue routines;
- a basic set of procedures written in the ISA of the underlying machine; and
- a set of procedures constructed from pointers to prologues and sequences of addresses pointing to either this set or the basic set of procedures.

There is a performance difference between this and an application written totally in the underlying machine's ISA. It consists of the costs of the address interpreter and prologue codes and the possible loss of machine register optimization for parameter passing. The cost of the first two items tends to be small for many modern ISAs; that of the third must be weighed against the standards imposed for procedure calls (such as register saves and restores) in many software systems and the less than optimal code produced by many compilers and programmers required to produce a great deal of code.

Benchmarks show typical performance penalties for threaded-code versus direct assembly coding to run in the range of 2:1 for eight-bit microprocessors and 1.2:1 for minicomputers. This is in contrast to orders-of-magnitude loss when more conventional interpreters (such as those for Basic) are used.

This small loss in performance is usually countered by a decrease in program size and a gain in machine independence. The only machine-dependent parts of an application package are the prologue code and basic procedure set, each of which is relatively easy to transport to a different machine. The bulk of the application consists of addresses, which can, conceivably, be transported intact to another machine.

While these results meet some of the claims listed in the introduction, they do not address others. Compile capa-

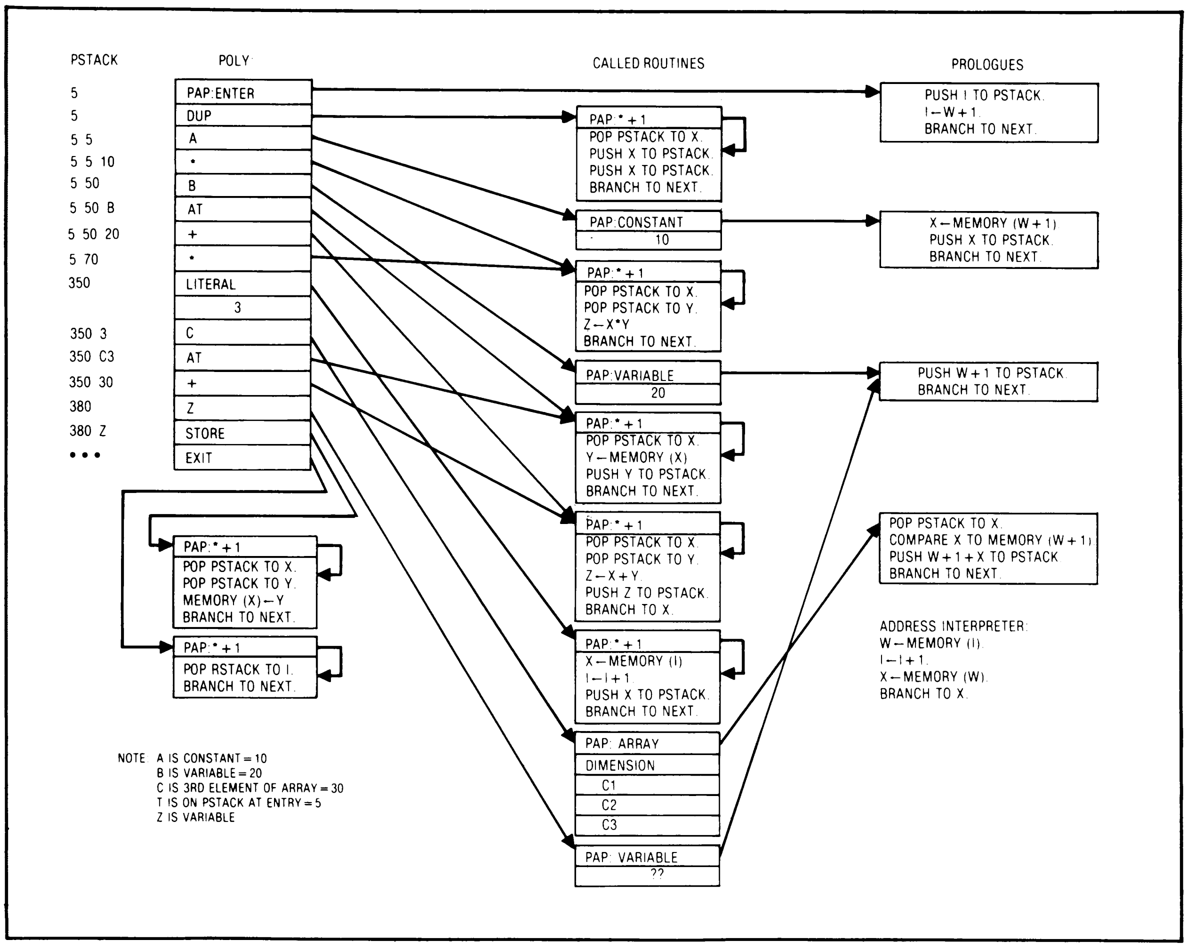

Figure 3. Procedure for polynomial evaluation; a detailed example that incorporates generalizations 1 through 6.

bility and interactiveness, for example, can only be achieved through certain programs that can be written in the architecture. The following subsections develop these programs and related concepts.

Generalization 9: The dictionary. The first step toward making a collection of threaded-code procedures interactive with a human is to provide some mechanism to translate the symbolic name of a procedure into its definition as a PAP and body. The simplest and most common mechanism is to precede each stored PAP with three items:

- the symbolic name of the procedure as a character string;
- a pointer to another procedure's name; and
- certain other information that is relevant at compile time.

The pointer field is used to link the names of the procedure set together in a list, called a *dictionary*. Starting with a symbolic name, a search of this dictionary can return a pointer to the appropriate PAP and body. All the standard search speed-up techniques, such as hash tables, are applicable and have been used in various systems. Figure 4 diagrams part of a possible dictionary.

The text interpreter. The combination of a dictionary and the text interpreter (sometimes called the *outer interpreter*) forms the basis of an interactive computing capability. The text interpreter is a simple program that accepts input characters from a terminal; as soon as a name is complete, it searches the dictionary for a match. If one is found, the entry in the dictionary can be executed. If no match is found, the program attempts to determine whether or not the name is a number. If it is, its translated value is left on the parameter stack. If not, an error message is sent. In either case, the text interpreter then returns for more input.

The text interpreter is extremely simple, but has two key aspects that give it a true interactive capability. First, the program is designed so that when it actually executes a user-specified procedure, there is nothing from the text interpreter on either the parameter stack or the return stack. The user therefore perceives both stacks as his and feels free to store information on them. Second, procedures are executed in the order in which they are typed in, from left to right. Thus, when a reverse Polish notation is used to express a computation, it corresponds exactly to the equivalent sequence of procedure names to be typed into the text interpreter. Thus, the sequence 10 20 30 * + PRINT enters 10, 20, and 30 onto the parameter stack, multiplies 20 by 30, adds 10 to the product, and uses the procedure PRINT to print the results.

In contrast, most other interactive systems, such as BASIC and APL, require significant syntactic processing on whole lines of text before execution can begin. There is no need for equivalent syntactic analysis in this system.

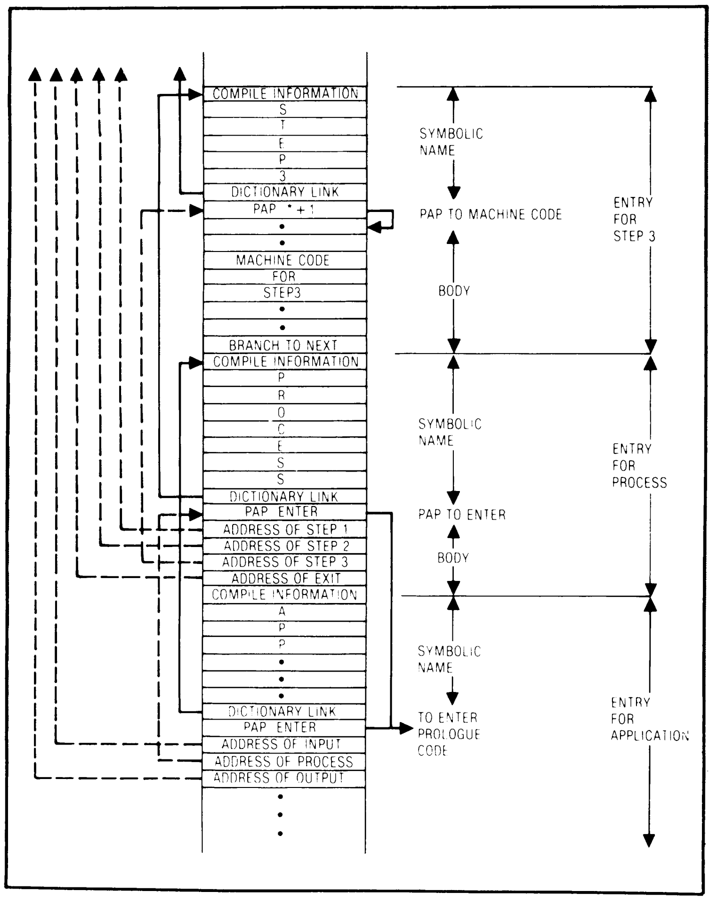

Figure 4. Sample dictionary format.

Generalization 10: The compile state. Since the system has no need for syntactic analysis, it is possible to implement a one-pass compile facility. Basically, when this facility is triggered by a user command, it establishes the name of a new procedure in the dictionary. Then, instead of executing the subsequent procedure name inputs, it simply stores the addresses of their PAPs in successive locations of the body of the new procedure. Detection of a termination command completes the compilation and returns the text interpreter to its normal state.

This compilation facility is typically implemented by modifying the text interpreter such that it has two states and then defining two procedures to govern transitions between states. State zero corresponds to the execution state, in which detection of a procedure name in the input stream causes immediate execution of its PAP and body. In state one, the compile state, detection of most procedure names causes not their execution, but simply the addition of a copy of the address of their PAPs to the body of a new procedure definition.

In many systems, the two special state-controlling procedures have the symbolic names : and ;. The procedure : (DEFINE), when executed, uses the next symbolic name from the input stream not in a dictionary search, but as the information needed to start a new dictionary entry. This new entry is linked to the previous dictionary entries in the appropriate fashion and is given a PAP that points to the ENTER prologue. The body is left open, and the state of the outer interpreter is set to one.

After execution of a :, the outer interpreter takes names from the input stream, looks up each in the dictionary, and (with certain exceptions) adds a copy of the address of its PAP to the new entry. This is an incremental compilation.

The procedure ; (END-DEFINE) is an example of a set of procedures that should be executed even in compile state. A typical way to make this determination is to include in each dictionary definition a *precedence bit*. If the bit is a one, it implies that the procedure should be executed in either state; if zero, it implies that it should be executed only in the execute state.

This precedence bit is set in the definition of ;. Thus, when ; is encountered in the input stream, it is always executed. Its execution causes completion of the current definition by adding a final address to the EXIT procedure (used in the previous examples) and by resetting the interpreter state to zero.

As an example, the following text is sufficient to define the polynomial evaluation routine shown in Figure 3.

```
: POLY DUP A * B @ + * 3 C @ + Z ! ;
```

The new procedure's name is POLY; assume that previous definitions have set up *A, B, C,* and *Z* as earlier defined. Figure 5 diagrams the changes in state and dictionary as this input stream is encountered.

Finally, Figure 6 is a flowchart of the text interpreter. The part in the dashed box is the entire code needed to achieve a basic compile facility. All of these procedures can themselves be defined (or redefined dynamically by the user) by a :...; procedure. For example, the entire text interpreter procedure can be written comprehensibly in six lines of source text (not counting the lower-level procedures it calls).

Generalization 11: Structured code. The ability to exectue procedures during compilation of another gives the system a powerful macro-like capability. Perhaps the most useful of such "macros" are those that implement most of the block structures found in many HOLs. Con-

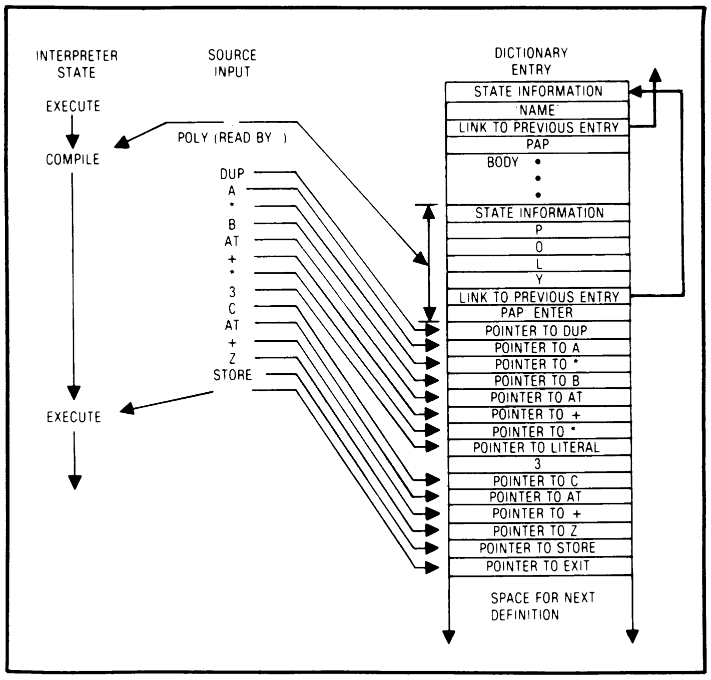

Figure 5. Sample compilation.

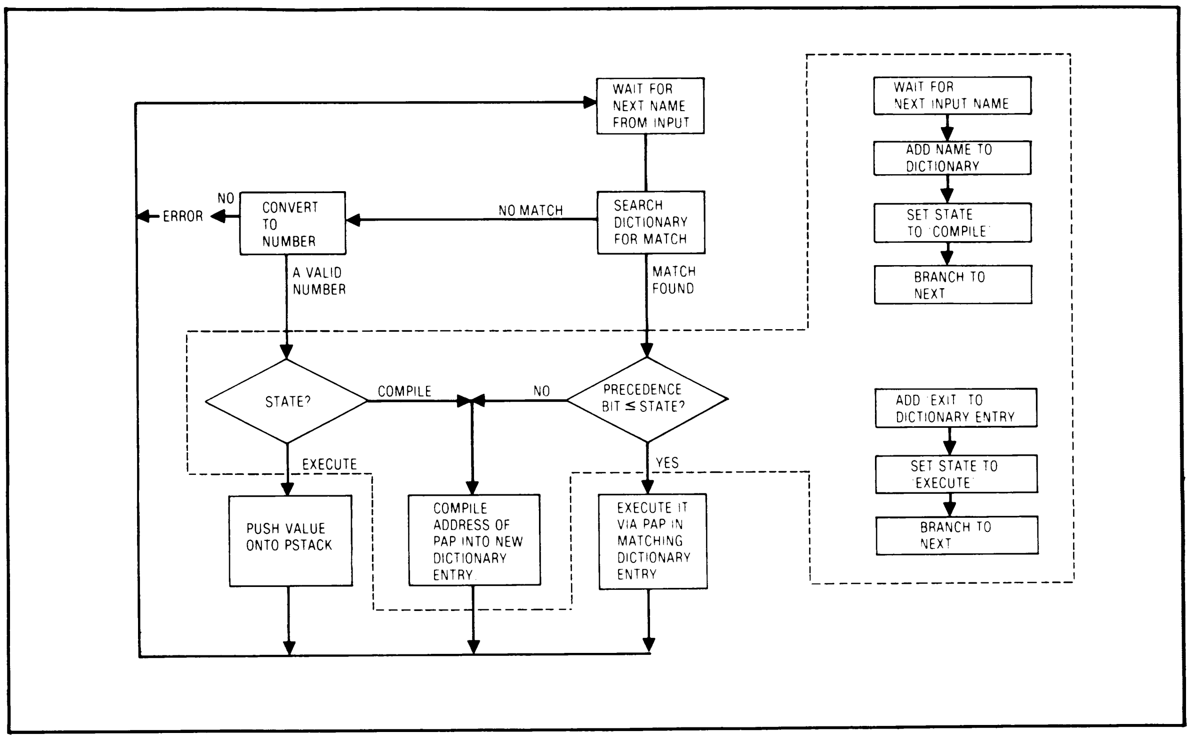

Figure 6. Text interpreter with compile facility. Dotted box contains all basic compile functions.

sider, for example, the following definition:

$$\text{: TEST } \underbrace{\text{X @ <0}}_{\text{Test X for <0}} \text{ IF } \underbrace{\text{0 X !}}_{\text{Then Part}} \text{ ELSE } \underbrace{\text{Z @ X !}}_{\text{Else Part}} \text{ ENDIF ;}$$

This tests the variable *X*. If *X* is less than zero, the definition zeros *X*. If *X* is not negative, it sets *X* to the current value of *Z*.

Of the words used in this definition, IF, ELSE, and ENDIF all have their precedence bits set, which means that they are executed during compile time. Typical definitions of each are as follows:

(1) *IF*. At compile time, this procedure adds to the current definition a pointer to the BRANCH0 procedure. It also leaves on the parameter stack the address of the memory word that follows the BRANCH0. This word should contain the displacement to the beginning of the ELSE code, but since the ELSE has not yet been encountered, it cannot be filled in. The parameter stack is a convenient place to store a marker noting this fact.

(2) *ELSE*. When ELSE is encountered, all the code for the THEN part of the IF-THEN-ELSE structure has been compiled. The ELSE procedure thus codes into the new definition the address of a BRANCH procedure, which should branch around the as yet undefined ELSE code. Again, since this displacement is not known, the ELSE procedure leaves on the parameter stack the address of the word that is to contain the displacement. Further, it takes the previous top of the parameter stack (the address of the displacement left from IF) and fills in the specified location with a displacement to the next free word in the new definition.

(3) *ENDIF*. When this is encountered, the IF-THEN-ELSE is complete. No new code is generated; instead, the branch at the end of the THEN code is fixed up. Again, the address of the word to be modified is on the parameter stack at compile time.

Figure 7 diagrams this process in action. The parameter stack is used at compile time to hold compiler-relevant data. One of the key advantages of using a stack is that these IF-THEN-ELSE blocks can be nested; the algorithms, as specified, handle the nesting just as one would expect.

Similar constructs can implement virtually all of the other block structures found in most HOLs, such as DO-UNTIL, DO-LOOP, and BEGIN-END. Their definitions are simple enough for a user to understand and even modify.

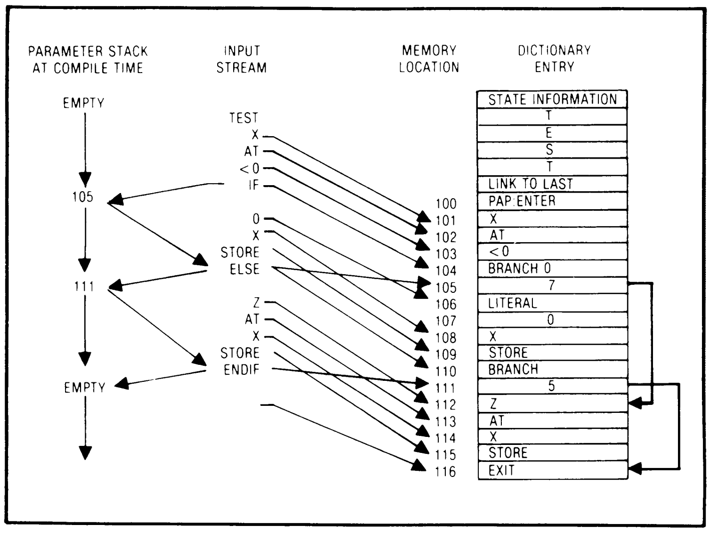

Figure 7. Sample compilation of the if-then-else block.

```
CODE MOVE      ( BLOCK 2-BYTE MOVE ROUTINE )
C L MOV        ( AT ENTRY, PSTACK=COUNT )
B H MOV        (                DESTINATION )
B POP          (                SOURCE . . . )
D POP
XTHL           ( GET PARAMETERS INTO MACHINE REGISTERS)
BEGIN          ( EXAMPLE OF A LOOP STRUCTURE )
  B A MOV      ( TEST COUNT )
  C ORA
WHILE
  M A MOV      ( MOV NEXT 2 BYTES )
  D STAX
  H INX
  D INX
  M A MOV
  D STAX
  H INX
  D INX
  B DCX        ( DECREMENT COUNT )
REPEAT         ( END OF LOOP -  GENERATES NEEDED BRANCHES )
B POP          ( RESTORE REGISTERS )
NEXT JMP       ( RETURN TO INNER INTERPRETER )
END-CODE       ( END OF CODE GENERATION )
```

Figure 8. Sample machine-coded procedure.

Generalization 12: Defining machine-coded procedures. One of the advantages of having the PAP identify the procedure type is that this makes it possible to design and debug an entire application in a totally machine-independent fashion by using the previously discussed compile facility. Then, one can go back and convert selected procedures to machine code for better performance. As long as the parameter-passing interfaces (i.e., the two stacks) are maintained, the rest of the code need not be modified in the slightest.

The process of generating a machine-coded procedure can, in fact, be much like that that of generating the compiler. Two procedures, often called CODE and END-CODE, work similarly to : and ;, except that they do not change the compile state. The PAP of the new entry started by CODE points not to ENTER, but to the start of the machine code to be produced; on exit, the machine code that branches to NEXT is added. Between CODE and END-CODE are operands and opcode mnemonics. Actually, these are previously defined procedure names that, when executed, produce the appropriate code. By reversing the order of operands and opcodes, the operand procedures can, at code generation time, leave on the parameter stack information as to the type of operands to be used. The opcode procedure can then use the information immediately. This eliminates the need for syntactic analysis. It also allows the programmer to perform arbitrary calculation at assembly time in support of the assembly. This, again, is a macro-like capability.

As in the compile mode, it is possible to build in block-structured procedures that automatically and directly generate branch code. For most cases, this matches the programmer's intent without requiring him to define branch labels or branch instructions.

Figure 8 diagrams a sample assembly for a typical microprocessor. For this machine, an entire assembler can be written in as little as one page of source code.

Advanced concepts

Many implementations of threaded-code systems include features that do not follow directly from any of the above discussions.

Vocabularies. The first of these features is the concept of vocabularies. While the reader might have assumed from previous discussion that the entire dictionary is linked into a single (perhaps hashed) list, this need not be the case. Collections of procedures that are part of an overall function, such as assembling machine code, can be separated from other procedures and given a name. This name is that of a specialized *vocabulary*. Users are quite free to build vocabularies for specialized applications where, for example, the entire interface specification to the eventual user is basically a description (names and functions) of the procedures in that vocabulary. This is similar to, although not as comprehensive as, the capabilities found in large software systems and such modern programming systems as DoD's Ada.

Another advantage of vocabularies is the high speed and ease with which source text can be compiled. This permits a programmer to keep the original source for specialized vocabularies on a cheap storage medium—like disk—and read and compile it into the dictionary only when needed. Support tools like linkage editors, loaders, and relocators are not needed.

Screens. Another concept used in many implementations is that of a memory hierarchy based on units of storage called screens, or blocks. Each unit is roughly equivalent to the amount of information that can be placed on a video display screen at one time. Many of these units are available to the programmer; each has a unique number. Although conceptually on disk, copies can exist within storage buffers in main memory. These, in turn, are managed like a cache, with references to the slower backing medium made only when necessary. Both text and data can be kept in these screens and accessed by simply calling up a small set of procedures. For example, the command 103 LOAD might compile the source code on screen 103 into the current dictionary. Commands on this screen might themselves direct compilation on yet other screens, allowing hierarchies, vocabularies, and libraries to be built up easily. Use of named constants permits a symbolic reference to units such as ASSEMBLER LOAD, where ASSEMBLER is a constant that returns the appropriate screen number.

History

Although the idea of threaded code has been around for a while, Charles F. Moore created nearly all the other concepts discussed in this article. He first produced a complete threaded-code system for real-time control of telescopes at the Kitt Peak Observatory in the early 1970's. This system, and many of its successors was called Forth, a foreshortening of *Fourth* for Fourth Generation Computer. A version of Forth is a standard international computer language for astronomers. Other major, but separate, implementations include Stoic, IPS, and Snap. Stoic has found use in hospital systems, IPS (developed in Germany) in earth satellite control, and Snap in several new hand-held computers.

Interest in such systems has increased dramatically since the introduction of personal computers; implementations are available for almost any make. Standards groups formed in the late 1970's are engaged in defining systems that are consistent and compatible over a wide range of hardware configurations. In the last two years, at least three books and dozens of articles on the subject have been published.

Interest in Forth developed more slowly in industry than it did in academic and personal computing, but seems to be picking up there, as well. Several groups, for example, are developing initial support software for a new military computer architecture and using such systems as a development tool. Others have found up to 80

• NEW FROM YOURDON PRESS •

WRITINGS OF THE REVOLUTION: SELECTED READINGS ON SOFTWARE ENGINEERING

Edited by Ed Yourdon, *Writings of the Revolution* is a collection of twenty-five hard-to-find articles that helped define the state of the art in analysis, design, and programming. It includes papers by Parnas, Dijkstra, Boehm, and Mills. Softcover, $30, 1982, ISBN: 0-917072-25-1.

Classics in Software Engineering, also edited by Ed Yourdon, is a collection of twenty-four landmark papers tracing the evolution of structured techniques. Softcover, $31, 1979, ISBN: 0-917072-14-6.

Both books are indispensable library additions for all who want a handy collection of the philosophy and guidelines of software engineering.

To order your copies (you can charge them to VISA or MASTERCARD), or for more information, call toll free 800 223-2452. Or, write

Yourdon Press
Dept. Comp
1133 Avenue of the Americas
New York, NY 10036

Reader Service Number 3 ►

percent commonality between the ISA of threaded-code systems and architectures like P-Code (a standard intermediate language for Pascal). This commonality permits consideration of hardware designed to efficiently execute either.

Conclusions

Threaded-code systems represent a synergistic combination of techniques that are, by themselves, of limited utility but together provide unique capabilities. Threaded code, particularly indirect threaded code, provides machine independence, portability, performance, and extensibility. Stacks and reverse Polish notation yield simplicity and the possibility of one-pass compilation. The ability to define certain routines in the underlying ISA provides performance when it is really needed and easy handling of new or diverse I/O devices and peripherals. Dictionaries and the text interpreter integrate these features into a highly interactive programming resource that the average programmer finds comprehensible from top to bottom.

In a very real sense, threaded-code systems represent neither a machine-level ISA nor a HOL and all the system support that goes with it. Instead, they represent a bridge between the two and give the programmer the features and flexibility of either or both, in the combination that best matches the application.

Although they are certainly not the ultimate or universal programming systems, threaded-code systems should find a secure niche where small-to-moderate specialized interactive application packages, real-time control of external devices and sensors, and engineering-type throw-away code dominate. Examples include video games, test equipment, distributed control systems, hardware simulation packages, and specialized software and hardware development tools. ■

Acknowledgment

I would like to thank D. Colburn for his helpful suggestions regarding terminology.

Bibliography and information sources

Threaded-code concepts

Bell, J. R., "Threaded Code," *Comm. ACM,* Vol. 16, No. 6, June 1973, pp. 370-372.

Dewar, R. B. K., "Indirect Threaded Code," *Comm. ACM,* Vol. 18, No. 6, June 1975, pp. 330-331.

Ritter, T., and Walker, G. "Varieties of Threaded Code for Language Implementation," *Byte,* Vol. 5, No. 9, Sept. 1980, pp. 206-227.

Language Implementations

Dewar, R. K., and McCann, A. P. "MACRO SPITBOL—A SNOBOL, 4 Compiler," *Software—Practice & Experience,* Vol. 7, No. 1, Jan. 1977, pp. 95-113.

Forth Interest Group, *Fig-FORTH Installation Manual and Assembly Listings,* San Carlos, Calif., 1979.

Loeliger, R. G., *Threaded Interpretive Languages, Byte* Books, Peterborough, N.H., 1981.

Meinzer, K., "IPS—An Unorthodox High Level Language," *Byte,* Vol. 4, No. 1, Jan. 1979, pp. 146-159.

Phillips, J. B., et al., "Threaded-code for Laboratory Computers," *Software—Practice & Experience* Vol. 8, No. 1, Jan. 1978, pp. 257-263.

Sachs, J., "An Introduction to STOIC," Technical Report BMEC TR001, Harvard-MIT Program in Health Science and Technology, Cambridge, Mass., June 1976.

Programming techniques and manuals

FORTH Standards Team, *Forth-79,* PO Box 1105, San Carlos, Calif. 94070, Oct. 1980.

James, J., "What is FORTH? A Tutorial Introduction," *Byte,* Vol. 5, No. 8, Aug. 1980, pp. 100-126.

Miller, A., and Miller, J., "Breakthrough into FORTH," *Byte,* Vol. 5, No. 8, Aug. 1980, pp. 150-163.

Using FORTH, Forth, Inc., Hermosa Beach, Calif., 1980.

Brodie, L. *Starting FORTH,* Prentice-Hall, Englewood Cliffs, N.J., 1981.

Feierback, G., "Forth—the Language of Machine Independence," *Computer Design,* June 1981, pp. 117-121.

Katzan, H., *Invitation to FORTH,* Petrocelli Books, New York, 1981.

History

Moore, C., "The Evolution of FORTH—An Unusual Language," *Byte,* Vol. 5, No. 8, Aug. 1980, pp. 76-92.

Moore, C., "FORTH: A New Way to Program a Minicomputer," *Astronomical Astrophysics Supplement,* Vol. 15, 1974, pp. 497-511.

Advanced concepts

Harris, K., "FORTH Extensibility: Or, How to Write a Compiler in Twenty-Five Words or Less," *Byte,* Vol. 5, No. 8, Aug. 1980, pp. 164-184.

Rather, E., and Moore, C., "The FORTH Approach to Operating Systems," *Proc. ACM '76,* Oct. 1976, pp. 233-240.

Interest group

Forth Interest Group, PO Box 1105, San Carlos, Calif., 94070.

Newsletter

Forth Dimensions, Forth Interest Group, San Carlos, Calif.

Peter Kogge, a senior engineer, is responsible for systems architecture in the MCF project at IBM's Federal Systems Division facility in Owego, New York. Prior to this, he was responsible for much of the organization of the IBM 3838 array processor and Space Shuttle input/output processor. He has taught at the University of Massachusetts and is currently an adjunct professor in the Computer Science Department at the State University of New York at Binghamton. His research interests include high-speed algorithms and computer organization. He is author of *The Architecture of Pipelined Computers,* published by McGraw-Hill in 1981.

Kogge received the BS degree in electrical engineering from the University of Notre Dame in 1968, the MS in systems and information sciences from Syracuse University in 1970, and the PhD in electrical engineering from Stanford University in 1973.