1-1-1983

Pistol.

Edward F. Bacon

PISTOL

A Threaded Interpretive Language

by

Edward F. Bacon

A Thesis

Presented to the Graduate Committee

of Lehigh University

in Candidacy for the Degree of

Master of Science

in

Computing Science

Department of Mathematics

Lehigh University

1983

This thesis is accepted and approved in partial fulfillment of the requirements for the degree of Master of Science.

July 17, 1983
Date

Professor in Charge

Chairman of Department

"If the pocket protector fits...wear it!"

-Jane Pali

Table of Contents

List of Figures

ABSTRACT

PISTOL (Portably Implemented STack Oriented Language) has been modeled after two threaded interpreters, FORTH and STOIC. This paper examines the nature of threaded languages and the particular installation of PISTOL for the Apple][.

1. Introduction

This paper looks at threaded intepreters, in particular the implementation of PISTOL. I begin with a section that will give the unfamiliar reader a brief overview of a stack oriented language and some of PISTOL's features. The second chapter discusses the internal workings of a threaded interpreter, using FORTH and STOIC as models. Chapter three deals with the nature of programming in these languages and what makes them unique. The final chapter covers the inner workings of PISTOL as written in C for the Apple.

1.1 A Look at PISTOL

PISTOL is an interactive language, commands are immediately interpreted and executed as they are entered at a terminal. Unlike a language like BASIC, it offers the user greater flexibility in naming variables and accessing more of the computer's capabilities. Features of PISTOL and other threaded languages include their extensiblity and the number of entry points offered by the system. From the command level the user may execute or define any number of routines. These languages are compiled in the sense that during actual execution (run time) none of the source code is rescanned.

PISTOL consists primarily of a dictionary of words.

Each word has a unique meaning and interpretation. For instance, the word CR means "carriage return" and will cause the ASCII character 13 to be sent. Initially PISTOL comes with a small dictionary of about seventy words, which constitute the basis to generate new commands in the language.

String literals take two forms in PISTOL. A string may be preceded by a single quote and terminated by spaces or tabs. A string may also be enclosed by double quotes. For example each of these are string literals:

```
'GOOD-DAY
"THIS IS A STRING "
```

The token, 146, is an example of a **numeric literal**, its value is determined by the number base the system is currently using. When 146 is typed its value is placed on a stack, thus allowing other words to access its value.

Most data passed between PISTOL procedures uses the **parameter stack.** PISTOL employs reverse polish notation for all its operations. RPN requires that operands precede operators, and eliminates the need for parenthesis. To get the equivalent of the algebraic expression

```
7 * ( 8 + 12 )
```

in PISTOL, type

```
7 8 12 + *
```

This will place 140 on the top of the stack, to see the result printed you must explicitly type = .

PISTOL words may be used directly as commands to the computer or may be compiled into the definition of new words. In fact, programming in PISTOL consists of defining new words in terms of existing words. As an illustration, the word TRIPLE will be defined to multiply the value at the top of the stack by 3 and print the result.

```
'TRIPLE : 3 * = ;
```

The special words : and ; indicate to PISTOL to begin defining a new word in terms of the enclosed words. For more illustrations of PISTOL programming see the file PBASE2, which when LOADed defines the common PISTOL commands.

A sizable collection of new words can be created by using a simple line editor, which is itself defined in terms of PISTOL words, or by creating them as an external text file that is LOADed into the dictionary. After the dictionary (set of defined words) has been enlarged an image of memory may be saved on disk by the word

COREDUMP. Later the image may be recovered by using RESTORE. Together with the capability to restrict users to limited vocabularies, a programmer has the machinery to create specialized application software in a customized PISTOL.

PISTOL provides the means to do top-down structured programming. The idea of a PISTOL word as a clearly defined module which has been constructed from other PISTOL words, facilitates the top-down design and modularization of complicated programs. The language also includes a complete set of control structures:

IF..THEN	a single branch
IF..ELSE...THEN	a two way test and branch
OFCASE...ENDCASE	a multiway test and branch
DO...LOOP DO...+LOOP	looping structures similar to PASCAL's FOR ... DO
BEGIN..IF...REPEAT	like WHILE...DO in PASCAL
BEGIN... .. END	like REPEAT... UNTIL NOT..

A feature of PISTOL is that these control structures may be executed at the commnd level as well as appear in the definition of new words. This is possible because PISTOL compiles every line into a buffer and then executes it. Such a scheme allows PISTOL to handle forward references to an address and to support recursive definitions.

User friendliness was a major design consideration for PISTOL. The language system includes a disassembler and trace facilities. The prompt displays the current number base, the number items on the parameter stack, and syntax level information. Each installation of PISTOL supports on-line help files and a tutorial.

2. Architecture of Threaded Code Languages

This chapter discusses the major characteristics of a threaded language system, namely:

- A simple instruction set for an abstract machine, written as short code segments in another current machine architecture.

- An interactive (conversational) monitor that permits direct execution of virtually all commands of the system and direct interaction with the user-defined objects. Programs (words) created by the user effectively extend the language and can be used either interactively or in new definitions.

The chapter outlines the mechanisms employed to implement a threaded language, stacks, headers, and the inner and outer interpreters. The basis for this chapter comes from examining FORTH, STOIC and PISTOL, as well from a book by Loeliger [13]. I tried to present a generic description of threaded languages, and at times will refer to the specifics of FORTH, STOIC or PISTOL. The examples of code in this section are from a ficticious machine (the BLT-90), and are meant to serve as outlines. A more detailed description of PISTOL can be found in chapter 4.

2.1 The Instruction Set Architecture

The essential idea of a threaded language is to create a simple yet useful and easily understood psuedo-machine from a real machine. An inner interpreter and at least two stacks control the execution of the machine. The instructions are either a small number of primitives or higher level secondary instructions. The economy and portability of these languages comes from the realization that these primitives and I/O routines are the only code that need to be written for the real machine.

Designers of various threaded languages differ on the function of the primitives. Versions of FORTH usually come with host-specific code for most of the single-length math operators and number formatting words, single-length stack manipulation operators, editor commands, branching and structure control words, the defining words, and the interpreters. There are many versions of FORTH for different computers, and hence a movement to standardize and formally define the language [7, 17]. STOIC [15] starts with an 8080 assembler, stack and arithmetic operators, and fewer control words, but can only run on the 8080 family of processors. PISTOL has roughly 70 primitives which supply a broad and universal set, on which all

implementations can be produced to run identically.

The higher order instructions in the threaded language consist of lists of pointers (addresses) to primitives or previously defined secondaries. Programs conceptually are tree structures, whose interior nodes are the addresses of secondary instructions and whose leaves reference primitive instructions[1]. The inner interpreter traverses the list of addresses in a depth first fashion until it encounters a primitive to be executed by the host processor. A return stack governs the flow of control, and a separate LIFO stack is used for passing parameters and for the temporary storage of local variables. By using a return stack, the psuedo-machine can execute instructions in the order in which they are encountered. The parameter stack effectively creates a zero-register machine, allowing procedures to be defined without formal arguments.

---

[1] Within a definition there is the possibility of multiple occurences of a word and in PISTOL there may be recursive calls. Technically therefore, some programs may not be trees, rather they are directed loop-multigraphs or psuedo-graphs [5].

2.1.1 Removing CALL Instructions -- The Inner Interpreter

The end result of a structured programming solution is a heirarchy of procedures (subroutines), each with well defined interfaces and concise understandable bodies (with minimal side affects). Ultimately such a program's executable code is mostly made up of addresses for the procedures proceeded by a CALL opcode. **Direct-threaded** [2] interpreters use only the list of addresses and an **address interpreter**, a machine-language routine, NEXT, that sequentially passes through the list making indirect branches at each address. To facilitate program control, any return from a routine is replaced by a branch to NEXT. As an illustration consider the following "application" consisting of level-1 routines that are defined only in terms of primitives, and level-2 routines made up of primitives and level-1 routines:

```
PROGRAM APPLICATION;

        PROCEDURE INITIALIZATION;
           BEGIN
              ... some code ...
           END;

        PROCEDURE GET_INPUT;
        { level-2 routine, calls on
          lower level procedures }
           BEGIN
                OPEN;
                READ;
                CLOSE;
           END;

        PROCEDURE GIVE_OUTPUT;
           BEGIN
              ...more code ...
           END;

        PROCEDURE PROCESS;
        { level-1 routine, calls on
          primitive instructions }
           BEGIN
                STEP_1;
                STEP_2;
                STEP_3;
           END;

    BEGIN
         INITIALIZE;
        GET_INPUT;
        PROCESS;
        GIVE_OUTPUT;
    END.
```

A threaded language represents a level-1 routine, e.g. PROCESS in figure 2-1, as a list of addresses for primitive instructions, here STEP1 through STEP3. Each primitive has code executable by the host machine, but rather than end with a return opcode the routine branches

```
PROCESS IP <- address of pointer
                to first step
        branch to NEXT
PLST    addr STEP1
        addr STEP2
        addr STEP3
        .
        .
        .
STEP1   code
        branch to NEXT
```

Figure 2-1: A level-1 routine, PROCESS

to NEXT .

Figure 2-2 outlines the action of the inner interpreter. IP and PW are registers or dedicated memory addresses of the underlying machine. IP, the **interpreter pointer**, points to the next address in the list of procedures to be executed and PW is the address of the instruction currently being interpreted. NEXT assigns to PW the contents of IP, increments IP by the machine word size, W, and indirectly branches to the contents of PW.

```
NEXT    PW <- Memory(IP)
        IP <- IP + W
        branch to (PW)
```

Figure 2-2: Inner Interpreter for direct threaded code

This method of control may be extended to higher level definitions in the threaded language, by using a **return stack** to keep track of the IP values . An initial

segment of code in each procedure at this level will stack the current value of IP and assign IP to point to the new list of instructions. This **prologue code** effectively forces execution to a lower level definition. At the end of each procedure list is an address that points to a routine, RETURN, that pops the return stack, in order to return to the higher level (calling) definition, see figure 2-3.

```
RETURN pop from STACK to IP
       branch to NEXT
```

Figure 2-3: RETURN routine

In the above example, GET_INPUT would call on lower level routines OPEN, READ and CLOSE as in figure 2-4.

```
INPUT  push IP onto STACK
       IP <- NEWLST
       branch to NEXT
NEWLST addr OPEN
       addr READ
       addr CLOSE
         .
         .
         .
       addr RETURN
```

Figure 2-4: A level-2 routine, INPUT

Rather than write a copy of the prologue code into each procedure at this level, we could store the address of the routine as the first entry of the definition. Since the primitives of the language should be executed

by the real machine and not interpreted, a different prologue is required at the lowest level. For an immediately executable routine, the inner interpreter should pass control to the host machine code that defines the primitive. One method sets the instruction pointer of the real machine, PC, to one word beyond the current address (see the example PLUS of figure 2-6). Furthermore, definitions of constructs such as variables and constants will require the interpreter to behave differently and therefore to expect a different prologue. We now have a collection of prologues for different types of definitions in the language, and require every definition begin with a pointer to the code for the appropriate prologue. These pointers to prologues require a modification in the address interpreter, which must now branch indirectly to the first word of the procedure.

```
NEXT    PW <- memory(IP)
        IP <- IP + W
        X <- memory(PW)
        branch to (X)
```

Figure 2-5: Inner Interpreter: indirect threaded code

The threading of a sequence of subroutines into a list of their entry addresses is termed **direct threaded code** in the literature [2, 6, 16]. **Indirect threaded**

code "consists of a linear list of words which contain addresses of routines to be executed" by Dewar's definition [6]. PISTOL is a variation on indirect threaded code, that uses lists of tokens which serve as an index into a table of routines to be executed. The **indirect token threaded code** offers even more machine independence at an expense in execution speed.

2.1.2 The Parameter Stack

To pass operands between the instructions a threaded language makes use of a parameter stack . Any routine that needs inputs takes them from the stack; any data returned by a routine goes back onto the top of the stack. Hence the need for general registers or accumulators can be largely eliminated. Furthermore, the parameter stack facilitates the use of **reverse polish notation**, RPN, to specify a series of operations. In RPN operands precede operators and evalation is from left to right. Parenthesis are not needed and no precedence is given to the operators. For example the primitive to add two integers is listed in figure 2-6.

Procedure calls (addresses) are maintained on the return stack but operands may only be found on the parameter stack. This use of multiple stacks greatly simplifies the implementation of the language and makes

```
PLUS    set PC to next word
        pop PSTACK to Z
        pop PSTACK to Y
        add Z to Y
        push Y to PSTACK
        branch to NEXT
```

Figure 2-6: Primitive to add two integers

program design conceptually easier for the user. The second is an important consideration, as the most obscure or unfamiliar aspect of programming in these languages is the stack manipulations. Separating the parameter and return stacks means the level of calls need not be taken into consideration when new words are used to rename existing routines. For example

```
'PLUS : + ;
'ADD : PLUS ;
```

are all equivalent; the only difference is a loss in execution speed. If the system used only one stack, the return addresses would interfere with the arguments.

The postfix stack architecture also creates some nice features for program development.

- To debug a module, the user explicitly places parameters on the stack and (interactively) executes the word she wants to test.

- Entering the variable's name places its

address on the stack, allowing various pointer calculations.

- Local variables need not be declared within a routine, just carefully placed and removed from the stack.

- Procedures may be written to accept a variable number of arguments, as in C.

2.2 The Outer Interpreter -- A Conversational Monitor

2.2.1 Headers

In order to make the collection of threaded-code instructions interactive with a human user, a mechanism to translate the symbolic name of a procedure into its definition as a prologue pointer and body is needed. A **header** preceeding the prologue addresses pointer is incorporated, and includes the following information:

- the symbolic name of the procedure as a character string;

- a pointer to another procedure's header; usually called the link field.

- other miscellaneous compile time or run time information;

The link field is used to chain the names of the procedure set together into a list, called a **vocabulary branch.** The **dictionary** consists of the collection of all

vocabulary branches. Starting with a symbolic name, a search of the dictionary will return a pointer to the header or body for the appropriate instruction.

2.2.2 FORTH and STOIC headers

FORTH and STOIC use essentially the same header. In the figure 2-7 each horizontol block represents one machine-word (2 bytes) of memory, and each dictionary entry has a three character maximium name field. Note that the newer and more general FORTH-79 standard permits up to thirty-one characters in the name field and allows the order of the fields to be implementation dependent. STOIC employs a 5 character name field that is null filled, if necessary. PISTOL's header is described in section 4.2.

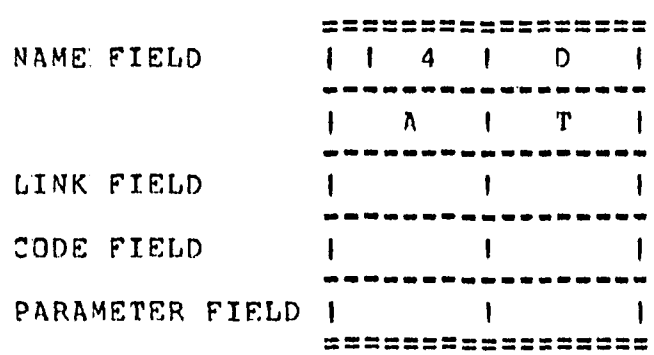

Figure 2-7: Sample Dictionary Header

Name field. The first byte contains the character count for the name of the defined word. There are special bits of this byte called **precedence bits.** Micro-Motion's FORTH [8] uses the next to most significant bit, bit six, to indicate whether the word is to be executed or to be compiled into the new definition during compilation. Bit five is set when the word is being defined and reset when the definition is complete. This is referred to as "smudging" the name field. Note that only words with bit 5 reset may be compiled into a definition, therefore a word may not refer to itself within its definition. The next three bytes contain the ASCII representation of the first three characters of the word's name.

Link field. The link field contains the address of the previous definition, thus chaining the word into the dictionary. To install a new word the compiler sets the new word's link field to point to the last entry in the dictionary and updates the system variable CURRENT to the address of the new word. To search for a name, start at the end of the dictionary and follow the pointers backward comparing name fields until a match or until the sentinal, 0, stored in the first word's link field is encountered.

Code field. This cell contains a pointer to the

appropriate prologue code, which distinguishes variables, constants and (colon) definitions. Briefly here is an outline of what the various prologues do at "run time"

- For variables push the address of the variable onto the parameter stack.

- For constants push the value of the constant onto the parameter stack.

- For colon definitions begin interpreting the word by setting the interpreter pointer, IP, to the parameter field.

- For primitives begin executing the native code by setting the hosts program counter to the next memory location[2].

Parameter field. The parameter field begins the data or code area used by the FORTH word . If the word is a variable or a constant this area is only one cell in length, it contains the value of the variable or the constant. In STOIC and PISTOL the address of the variable is stored in the parameter field, not the value. In primitive and higher level definitions the parameter field merely contains the first instruction in the body of the definition, and is where execution of the word

---

[2] In STOIC this action is performed by NEXT

begins.

2.2.3 Interpret State

The outer interpreter is a simple program that gets characters from a buffered input line. Upon recognizing a complete token, it searches the dictionary. If a match is found, the entry in the dictionary is executed (by the psuedo-machine inner interpreter). If no match is found, the program attempts to convert the token to a number in the current base. If the conversion is successful, the value is pushed onto the parameter stack. If the token is not numeric, some threaded languages (PISTOL and STOIC) will try to convert the token as a string and push a pointer to the string onto the stack. If all the conversions fail, an error message is printed and the program reset. This simple text interpreter design allows execution to occur in the order in which procedures are typed, from left to right, hence capitalizing on RPN conventions and greatly reducing the need for syntactical analysis.

2.2.4 Compile State

With little need for syntax checking, it is possible to compile new definitions in one pass. Compilation is triggered when the user types a defining word, such as :, the outer interpreter changes its state and function. Instead of executing the subsequent procedures, the compile facility enters the list of their starting addresses into the new procedure. When a terminating command is encountered, such as ;, the RETURN instruction is compiled into the definition, the new word is entered into the dictionary and the interpreter returns to its normal state. It is apparent that the word ; to terminate a definition should be executed and not compiled.

In general two types of behavior may be exhibited by a FORTH word: **run time** actions occur when the word is executed (in the interpret state), and **compile time** actions occur during the compile state. Some words behave in both ways and fall into the two general classes, usually referred to as defining words or compiling words [8]. Defining words specify the compile time and run time behavior for a family of words, for example the defining words CONSTANT and VARIABLE. When a user enters the definition

```
2 CONSTANT TWO
```

the compiler constructs a new dictionary entry called TWO and enters the value 2 in its parameter field. If the user subsequently types

```
TWO
```

the run time behavior of CONSTANT is executed and the value is 2 is pushed onto the stack.

Compiling words are used inside colon-definitions and cause the compiler to take specific actions, such as touching-up forward refences, thus ultimately affecting the run time execution. The compiler does not compile the address of the compiling word, but executes it instead. These **immediate** words are distinguished by the precedence bit in their name field.

The above scheme for a monitor restricts code generation to the compile state. And as execution in the interpret state is sequential, forward references and touchup must be prohibited while in the interpret state. FORTH for this reason limits the use of LOOP and IF-THEN statements to be within the definition of a word. To avoid this short coming, PISTOL and STOIC use a buffer to store compiled code, which is then executed by the interpreter. All addresses contained in the compiled

code are either absolute addresses of words in the dictionary or offsets relative to the IP. The code is position independent and will execute correctly in the compile buffer or when relocated in the dictionary.

3. Programming Philosophy of Threaded Language Systems

FORTH, the most popular threaded language, has gained a large group of advocates, who seem to have given this slightly unconventional language a cult status. They state many outrageous claims to its versatility and uniqueness, professing that it is THE way to program micro-computers. Clearly FORTH and other such language systems change the way a programmer thinks about her machine, her problem and the set of possible solutions.

These language systems supply the total environment to develop and execute programs. They contain an interpreter for interactive execution, compiler, built in utilities, and often their own operating system. Each of which may be modified or extended to some degree. This means that the artificial constraints to a problem that grow out of a software development system can be side stepped by changing the environment.

3.1 Top-down design; Bottom-up testing

Threaded language systems support the top-down analysis and design of a solution. The programmer expresses a complex task in terms of simpler set of less complex words, each of which can be refined (defined) still further until he reaches constructs of the basic language. However, it is most advantageous to use a

bottom-up order for implementation and testing. That is the lowest level modules (words) are written and tested before the top level modules.

In a language like PASCAL there is only one entry point, namely the main program, which then calls on procedures and functions to perform subordinate tasks. To test the top level module before lower level modules are created, requires the programmer to provide routines that do nothing when executed (except perhaps return simulated data). When these dummy routines are replaced by fully implemented modules, the top level must be retested. Alternatively the bottom-up order tests only the implemented modules as they are created, and does not require retesting as others are written and put in place. To achieve bottom-up testing and implementation, a language system must be interactive and allow enty points at any level of the program. The bottom-up implementation and testing offers easier debugging capabilities and faster overall program development.

3.2 Module structure

A tenet of structured programming is that a complex task should be decomposed into simpler sub-tasks or modules. Harris discussed the orginization and size of FORTH modules in [10]

- Each module should carry out a single action

- Each module should have a simple interface to others

- The modules should be grouped into layers of equal complexity.

- The layers should be ordered by complexity such that the bottom layer contains the simplest functions and the top would have the most complex.

- Modules should be small, generally not referencing more than nine others.

Harris states that the reasoning behind the last restriction comes from the number of things a human can "simultaneously analyze, trade-off, or optimize." And that FORTH programs will be simpler and easier to understand if definitions are not more than a few lines long. Of course FORTH's screen editor encourages short modules by offering only 24 lines on the Apple II[3]. As a

---

[3] Mass storage units are "blocks" if they hold data or object code or "screens" if they hold source code. In FORTH-79, each block of mass storage can hold 1024 bytes of data. If the block is used as a screen, these 1024 bytes will usually be organized as 16 lines of 64 characters each. The Apple requires 24 lines of 40 characters with 64 inaccessible characters in each screen.

result there will be many of these short modules to build a large, complex program. An extensive application in this type of language may be as unwieldy as an assembly language program. A label is used to indicate the entry point to each routine, and within the routine there is a collection of jumps to other labeled statements. In assembly language there is not much harm in creating all these labels, however for a threaded language the result will be a swollen dictionary and far too many words for a user to remember. This suggests that FORTH-like languages may not be suitable for large programming applications.

To speed up searches and avoid conflicts between some common words, the programmer may form **vocabulary branches.** These are indepent linked lists within the dictionary that chain together words used in a special context. For example the assembler which accompanies many of these language systems, is a specialized vocabulary that is accessable only during CODE definitions. PISTOL includes the word UNLINK to make rarely used, obscure or dangerous words inaccessable to a user.

3.3 Exensibility of Language

Compilation is the process of converting a source language program into a form that a computer can use. Compilers for most popular languages, such as PASCAL, are large complicated programs designed to handle every imaginable variation of the language's syntax. These compilers must also include storage allocation and code generation routines. Alternatively threaded language systems use multiple compilers to handle the functions that a larger language might. The system views compiling a constant declaration as a distinct process from compiling the definition of a new executable word, and as such handles them by different defining words. Most threaded languages provide a mechanism to declare a word "immediate," or executable during compilation, in effect allowing the user to create new defining (compiling) words. Since user-defined words are treated the same as system-supplied words, a programmer can extend the capabilities of the language by adding simple and specialized compiling words. FORTH offers the CREATE and DOES> combination to specify the compile time behavior and the run time behavior of a word. For a detailed discussion of CREATE and DOES> (or <BUILD and DOES> in earlier versions of FORTH) see Harris' article [9].

3.4 Criticism of FORTH-like Languges

Whole issues of BYTE and Dr. Dobb's Journal have been devoted to FORTH. But with all the acclaim comes some serious criticism of this unconventional language [1, 11]. The most unappealling aspect of FORTH is that its code is virtually unreadable. To understand the definition of a word requires a pencil and paper simulation to follow the use of the stacks. The compact code begs for extensive documentation. But the 1K screen afforded by the FORTH operating system seriously restricts definitions from including many comments. This environment is not optimal for production systems that would involve more than two programmers or an application with a long life span. Languages that are supported by a host operating system (STOIC, PISTOL and LISP), allow the user to load external files created by a friendlier editor that would afford more space for documentation.

In practice many programmers shun the transparency of local variables and parameters offered as a feature of the language. Instead storage is allocated to variables, which as dictionary entries are global hence susceptible to side effects, furthermore they may begin to bloat the dictionary. By adding floating point routines, graphics capabilities and other specialized vocabularies, the language becomes less compact and even sluggish. Some

users of FORTH systems have filled the available memory in a 48K Apple.

A postfix language may be difficult for even experienced programmers to learn. It requires a closer understanding of how a computer works. The language does not make the transformation from the way a human might define a programming solution to what the machine executes, the programmer must. Further the control structures are awkward and in practice tend to be abused. Just as an APL programmer will forgo good structured programming techniques for an indecipherable one-liner, FORTH programmers worship the compact, efficient solution.

3.5 Alternatives

GraFORTH is a wholly compiled version of FORTH written for the Apple II by Paul Lutus. Execution is very fast, allowing the animation and graphic capabilities for which it was conceived. Externally graFORTH looks like FORTH to the user. It employs a parameter stack and postfix notation, as well as FORTH conventions for defining words. GraFORTH, however does not use the standard FORTH operating system, rather hooks into the Apple DOS. It also does not conform to the FORTH-79 standard in many other places. Internally

graFORTH offers a completely different look than FORTH. It compiles a line into a buffer and executes it, like STOIC and PISTOL. Significantly the body of a definition is less like threaded code. It mostly is made up of calls to subroutines and includes the 6502 opcode JSR addr. This has many FORTH advocates upset, claiming that graFORTH is not really FORTH.

John McCarthy developed LISP as a language to process symbolic rather than numeric data [14, 18, 19]. It has been the main vehicle for encoding processes that exhibit artificial intelligence. LISP is an extensible interpretive language that employs prefix notation. Because it resembles functional notation, prefix is more familiar than postfix notation for most users. However derivatives of LISP (REDUCE and LOGO for example) use algebraic (or infix) notation. Execution of LISP-like languages is slower because most words are partially reinterpreted each time they are called. But the reinterpretation and blurred distinction between data and program gives LISP its most distinctive character.

3.5.1 Lisp-like languages

Lisp data are called **s-expressions** (symbolic expressions). The simplest s-expression is an **atom**, which is a numeric or literal. Non-atomic s-expressions are dotted pairs, represented as a two compartment cell whose left and right parts hold pointers to the left and right sub-expressions. The storage for the cells is an area of memory called the **heap**, and the value of a variable is a pointer to an s-expression in the heap.

The Lisp interpreter always tries to evaluate an expression and return a value; the value returned by an atom is itself. To defer evaluation by the interpreter use the quote, '. If an atom follows the quote, then a pointer to that atom is the value of the quoted expression. If a left parenthesis follows the quote, then a structure coresponding to the s-expression is created in the heap, and a pointer to this structure is the value of the quoted expression. The evaluation of an s-expression is done by the function **EVAL**, which recursively traverses the tree that represents the expression in a preorder. EVAL separates the expression into its left component, S, and its right component, called the **a-list** for associated list. If S is an atom return its value, namely return S. If is quoted return a pointer to S. If the first part of S (CAR S) is an

idiomatic Lisp form, eg. COND, perform the appropriate routine. Otherwise EVALuate the associated a-list and "apply" S to the returned value.

The syntax of Lisp for procedure calls requires **prefix notation**, that is the procedure name precedes its list of arguments. The body of a Lisp procedure is an expression and the value it returns is the value of that expression. Lisp functions are really data objects that are arguments to EVAL, and are reinterpreted every time they are called. This makes Lisp execution slow, but allows procedures that alter themselves while they are executed.

The property list of the item in the symbol table representing the function is the defining expression. The formal parameters are the second item in the expression and appear in a list that starts with LAMBDA. They receive their value from the actual parameters through "lambda binding." Arguments to the procedure are quoted in order to defer evaluation and to bind them to the lambda expression in the procedures definition.

4. PISTOL for the Apple][

PISTOL (Portably Implemented STack Oriented Language) was designed by Ernest E. Bergmann of the Physics Department at Lehigh University [3]. It is modeled after FORTH (Charles Moore, 1970) and STOIC (MIT and Harvard Bioengineering Center, 1977), but with a slightly different design philosophy. STOIC and FORTH were written to run on a micro- or mini-computer, but PISTOL was developed as a language to be used on large mainframe machines as well. A major goal included portability between machines with different word-lengths and instruction sets. Other criteria which directed the creation of PISTOL were: to add a greater degree of user friendliness, to bypass some of the bothersome short commings of FORTH, to be as self-contained and complete as possible, and to stress short simple and "stupid" routines.

Unlike FORTH, strings are a fundamental part of PISTOL. And as in STOIC, the name of a word being defined precedes the colon, hence achieving a greater degree of flexibility when defining new words.

PISTOL and STOIC compile every line into a buffer and do not require two modes of operation for the outer interpreter, as FORTH does.

PISTOL does not come with its own operating system,

but does have a resident line editor, a disassembler and trace facilities. To maintain portability between machines and insure that all defitions can disassemble completely, no facillity to write "CODE" definitions was included. However PISTOL does have in-line macro defining capabilities, and custom versions written in assembly language are planned to include CODE definitions [4].

PISTOL employs a different type of header than STOIC or FORTH. It uses a name field which points into the string area and recognizes a word by its entire name. PISTOL's header is also larger. By adding an extra field to the dictionary header that points to the end of the definition, PISTOL is able to implement macros which copy the code from the parameter field to the end of the definition directly into the compile buffer. This extra field also is used to indicate to the disassembler where to stop disassembling.

PISTOL has been written in BDS-C to run in a CP/M-80 based environment, and in PASCAL to run on the DEC-20. I have written the Apple II version in Aztec C as distributed by Manx Software. This chapter will discuss the inner workings of PISTOL as a threaded language. The examples of code that appear are taken from the implementations written in C.

4.1 The inner interpreter

In the implementation of PISTOL, the action of the prologue code and NEXT are combined in the function **interpret()** of figure 4-1. While the return stack is not empty (rptr >= 0), the interpreter increments **ip** by the machine word size, W. It then tests if the current instruction, **instr**, is a primitive; if yes, then execute the primitive, otherwise push the interpreter pointer, **ip**, onto the return stack and set it to instr. Finally the current instruction is set to the contents of ip.

```
#define         NFUNCS    74
#define         W         2
unsigned  ip, instr;
int       *Pw;
int       (*farray[NFUNCS])();
        .
        .
        .

interpret()
{       do {
           ip += W;
           if(instr < NFUNCS)  (*farray[instr])();
           else
                { rpush(ip); ip = instr; }
           Pw = ip;
           instr = *Pw;
           }
        while (rptr >= 0);
        ip -= W;
}
```

Figure 4-1: PISTOL's Inner Interpreter

There are aproximately 70 PISTOL primitives, each associated with an integer from 0 to NFUNCS. It is this

integer that is entered into a compiled definition and later assigned to instr. When a secondary reaches **interpret(), instr** holds the address of the secondary and will be larger than NFUNCS. Execution of a primitive comes from selecting a pointer to a function from **farray.** The interested reader should see section 5.12. of Kernighan and Ritchie [12] for a discussion of pointers to functions in C. The PASCAL implementation uses a large CASE statement to select the appropriate procedure.

4.2 PISTOL header

The header format for PISTOL consists of four fields:

ENDA	address of the end of the code body;
LFA	link field -- pointer to previous entry;
NFA	name field -- pointer into string area;
CFA	code field

ENDA most often points to the instruction which simulates a return, viz. the procedure **psemi().** The link field, LFA, points to the CFA of the previous entry; and the function **vfind()** follows these pointers attempting to match the current token from the text interpreter with the symbolic name of an instruction. NFA points into the string area, where strings are stored with a character

count and up to 127 characters. The CFA will contain different information depending on the word. Most primitives have the instruction **compme()**, which tells the compiler to copy from this point to the address pointed to by ENDA into the code of the word being defined (compiled). Secondary instructions contain the instruction **comphere()**, which tells the compiler to insert the address of the instruction into the code of the new word.

The ENDA permits two of PISTOL's unique features. The disassembler package uses it to decide when to stop disassembling a word. And the macro-defing words **$:** and **;$** rely on ENDA to bracket the definiton of a macro.

4.3 PISTOL's Outer Interpreter

The outer interpreter has been divided into two parts. The main loop of the program calls on **compline()** to enter a line into the compile buffer, and then executes the instructions in the buffer, see figure 4-2. **compline()** gets a buffered line of input, either from the console or an input file, then enters a loop to process the tokens. In this loop a pointer to the current token is pushed onto the stack, **find()** absorbs this pointer and searches through the dictionary for a match. If successful, **find()** leaves the CFA on the stack, otherwise

pushes 0 indicating no word was found in the dictionary. **compline()** then uses a nested conditional statement to decide how to handle the current token.

- If the address at the top of the stack (pad) is not zero, then **find()** succeeded and interpret the instruction in the CFA . Most often CFA contains the instruction compme or comphere. Compme() copies the entire definition into the compile buffer, and is used by primitive and macro definitions. While **comphere()** compiles the address of the word into the buffer.

- If **find()** did not succeed, try to convert the token to a numeric value using the current base. If **convert(-,-,-)** is successful, the instruction that indicates literal storage and the numeric value are entered into the compile buffer.

- Next try to recognize the token as a string literal. If the token begins with a single quote, process the string with **slit()**. Long strings are delimited by double quotes and are recognized by **longstring()**. Both functions return the address of the string, and after compiling the string literal storage instruction, **compline()** enters the address of the string into the compile buffer.

- If none of the conditions above are selected, the token cannot be deciphered. A message and the offending token are printed, control is returned to the main program loop where the pointer into the compile buffer, .C, is reset.

The main loop then interprets the instructions in the compile buffer.

As an example, suppose X is a variable that has been

```
compline()
{
     getline();
     ignrblnks();
     while ( nextcharptr != NEWLINE )
     {
          intoken();
          push( endofstrngptr );
          find();
          pad = pop();
          if (pad)  { instr = pad - 1; interpret(); }
            else { if (convert(endofstrngptr,base,&val) )
                     { compile(LIT); compile(val); }
                   else { if ( *Pc == '\'' )
                             {pad = slit();
                              compile(STRLIT);
                              compile(pad);
                             }
                          else { if (*Pc == '\"')
                                    {pad = longstring();
                                     compile(STRLIT);
                                     compile(pad);
                                    }
                                 else
                                /* token not deciphered */
                                   {message(endofstrngptr);
                                    printf(" ?\n");
                                    abort();
                                   }
                                 }
                             }
                   ignrblnks();
                   }
 }
```

Figure 4-2: PISTOL's Outer Interpreter

previously declared and given a value. If the user enters the lines

```
BEGIN
   X
   W@
   EQZ
END
```

compline() will proceed in the following manner:

- BEGIN is a primitive, so can be found in the dictionary. When **compline()** calls **interpret()** the routine **beginop()** is imediately executed. **Beginop()** pushes the compile buffer pointer, .C, onto the parameter stack for a future branch calculation.

- After matching X, the comphere instruction is passed onto **interpret()**, which places the address of the word X into the next location of the compile buffer.

- W@ is a primitive, whose CFA contains compme, which causes the token selecting **wat()** to be inserted into the compile buffer.

- EQZ is a secondary instruction, defined in PBASE2 to test the top of the stack for zero, and therefore has its address inserted into the compile buffer.

- The primitive END causes the instruction selecting **pif()** to be compiled into the buffer. END pops the address stored by BEGIN and computes the difference between that address and the current compile buffer pointer storing the result in the compile buffer.

Upon reaching the end of the line, control goes to the main loop which sets instr to the contents of the first entry in the compile buffer and calls **interpret()**.

When **interpret()** encounters the address of X, it realizes X is not a primitive; after saving a return address execution (interpretation) of X begins. The code for X pushes the address of the variable onto the parameter stack and **interpret()** returns to the compile buffer. The (primitive) code for **wat()** pops the parameter stack and pushes the contents of that address onto the stack. EQZ tests the top of the stack; if it is zero, pushes on TRU (-1), otherwise pushes on FALS (0). Next the interpreter finds the token for **pif()**, which pops the top of the parameter stack. If that value is 0, it then bumps the interpreter pointer to the next address. Otherwise **pif()** sets **ip** to the contents of the word to which ip is pointing, namely the value computed by the branch calculation.

PISTOL handles colon-definitions in much the same fashion. When **compline()**, the outer interpreter, encounters a **:**, it compiles the instruction pcolon into the buffer and calls on a routine to setup a forward reference. **fwdref()** pushes **.C** onto the parameter stack and compiles a 0 into the compile buffer which will be overwritten later during touchup. At the end of the new word's definition is **;**, which compiles psemicolon and calls **touchup()**. During interpretation of the compile buffer, **pcolon()** calls on **enter()**. This routine creates

a dictionary header with CFA containing the instruction comphere, and it updates the system variable CURRENT. **pcolon()** then moves the contents of the compile buffer into the dictionary area and finalizes the entry by patching up the word's ENDA.

Macro-definitions are delimited by the words $: and ;$. They cause similiar compilation and interpretation as colon-definitions, except in place of pcolon $: compiles pdollar, which during interpretation overwrites the CFA with compme.

A scheme to extend PISTOL to compile CODE definitions could incorporate an instruction like comphere. it would place the PFA in the compile buffer. During execution, the inner interpreter distinguishes between address list and machine code by "glancing up" at the header to see if CODE or comphere was used in the CFA. The interpreter would recognize that what follows is native code and pass control to the host processor.

4.4 Remarks on the Implementation

With portability as a design goal, the implementation language of PISTOL was chosen as C or PASCAL. The installation on a new machine that maintains one of these languages should be straight forward. However I had encountered some difficulties trying to put

PISTOL on the Apple][. The PASCAL version causes the internal stacks of the p-machine to overflow during compilation. I tried many combinations of units, include files, and swapping options, but never successfully compiled PISTOL using the Apple PASCAL.

The version written in C must be run with the Aztec C shell, an interpreter and operating system combination. When the relocatable code for PISTOL is linked to the libraries that support a "stand alone" program, the executable code grows very large and overwrites the DOS file buffers. The Aztec C shell adds another layer of interpretation which slows execution, particularly the I/O operations. I intend to write a version of PISTOL in 6502 code that will be more compact and faster than the present Apple version.

4.5 Conclusion

Threaded interpretive languages have made their mark in computing, particularly on minis and micros. Exhibiting great versatility, they have been used for many scientific and industrial applications. In the decade since Moore first developed FORTH, there has been a steady evolution of the languages. (Initially FORTH words were reinterpreted each time they were called.) PISTOL being the latest of the threaded languages, has

benefited the most from growing pains of FORTH. When a feature like string capabilities or the case-statement was added to FORTH, it was placed on top of the existing architecture often times in an awkward fashion. Rather than such a patch-work design, PISTOL started with a more flexible header and a compile buffer (ala STOIC). While PISTOL may sacrifice some execution speed and is not as compact as its predecessor, it offers more consistent and friendly aspects.

REFERENCES

[1] Barry, T.
On FORTH Failings: We need solutions not languages.
InfoWorld , October 11, 1982.

[2] Bell, J.R.
Threaded Code.
Communications of the ACM 16(6):370-372, June, 1973.

[3] Bergmann, E.E.
PISTOL A Forth-like Portably Implemented STack Oriented Language.
Dr. Dobb's Journal (76):12-15, February, 1983.

[4] Bergmann, E.E.
Private communication.

[5] Chartrand, G.
Graphs as Mathematical Models.
Prindle, Weber & Schmidt, Incorporated, Boston, MA, 1977.

[6] Dewar, R.B.K.
Indirect Threaded Code.
Communications of the ACM 18(6):330-331, June, 1975.

[7] EQRTH-79: A Publication of the EQRTH Standards Team
San Carlos, CA, 1980.

[8] EQRTH-79 Tutorial and Reference Manual Apple][version
MicroMotion, 12077 Wilshire Blvd West Los Angeles, CA 90025, 1981.

[9] Harris, K.
FORTH Extensibility.
BYTE 5(9):164-184, August, 1980.

[10] Harris, K.
The FORTH Philosophy.
Dr. Dobb's Journal (59):6-11, September, 1981.

[11] Hogan, T.
Demystify FORTH by facing the facts.
InfoWorld , October 11, 1982.

[12] Kernighan, B.W. and Ritchie, D.M.
The C Programming Language.
Prentice-Hall, Englewood Cliffs, NJ, 1978.

[13] Loeliger, R.G.
Threaded Interpretive Languages.
BYTE Books, Peterborough, NH, 1981.

[14] McCarthy, J., Abrahams, P., Edwards, D., Hart, T., Levin, M.
LISP 1.5 Programmer's Manual.
MIT Press, Cambridge, MA, 1962.

[15] Sachs, J.
STOIC (Stack Oriented Interactive Compiler)
Cambridge, MA, 1977.

[16] Sirag, D.J.
DTC versus ITC for FORTH on the PDP-11.
FORTH Dimensions 1(4), December, 1978.

[17] Ting, C.H.
Formal definition of FORTH.
Dr. Dobb's Journal (64):19-21, February, 1982.

[18] Winston, P.H.
Artificial Intelligence.
Addison-Wesley, Reading, MA, 1977.

[19] Winston, P.H., Berthold, K.P.
Lisp.
Addison-Wesley, Reading, MA, 1980.

Vita

Edward Francis Bacon was born on November 10, 1951. He attended Villanova University from 1969 to 1973, when he received a Bachelor of Science in Mathematics degree. In 1975, he received a Master of Science in Mathematics from Lehigh University. From 1976 to 1980, he taught mathematics at Stockton State College in Pomona, New Jersey. From 1980 to 1983 he attended Lehigh University as a graduate student in computer science and taught at Lafayette College in Easton, Pennsylvania.