Talk on interpreters

(handout at March 1982 DECUS meeting)
(Digital Equipment User's Group, San Antonio, Texas)

James C. Brakefield
Technology Inc.
San Antonio, Texas

Introduction:

The FORTH software system is rapidly becoming a hot item. Today we are going to examine the software region in which Forth resides. We will add perspective to the arena in which Forth resides (don't take nothing for granted).

We are going to cover variations on Forth. Ie., this is a talk on pertubations of a good idea. For detailed examination of Forth there is a recent paper in the IEEE Computer magazine (ref 1). There is also a very good introductory book available (ref 2).

The three corners of the arena are:

Forth: Interactive, primitive, complete, dictionary oriented

UNIX: Tools and files oriented, major language is C (bastard Pascal)

ADA: Super Pascal, complete developement system, lacks user experience

We owe DEC a vote of thanks for the hardware which provides the arena, for the PDP11 is the original hardware for Forth, USCD Pascal, and UNIX.

Preface:

The goal is to provide better tools for the professional programmer. The current situation is that one either uses the tools provided "as is" or one goes without. Tools are the software utilities available on a given system, ie. the assembler, the editor, the languages, the file sytem, the operating system, and the rest. The only tools the average programmer can build with ease are command files and libraries. Customizing the languages or the operating system is dependent on alot of time and skill.

Item of significance #1:

What follows is based on a particular use of memory and requires a computer which can support two stacks and has some remaining registers. The usual stack is used for temporaries and stack arithmetic and refered to within as the value stack. The second stack is used for return addresses and loop counters and refered to within as the control stack. This seperation of aritmetic and control information has far reaching effects on the favored programming discipline. In particular, the organization of a program can be better structured to reflect the structure of the solution of the problem that the program solves.

Let me say that the interest in interpreters derives from several factors. Most of the original Forth work was done on the PDP11 series of computers which as everybody knows has byte addressing and the autoincrement/decrement addressing modes which allow low overhead stack support. DEC did its original Fortran

compilers using a slightly different interpreter. The USCD Pascal project began using the LSI11 computer with yet another interpreter. One thing in common with all three of these efforts is that the language or tool has since been moved to other computers and that source programs are to a large extent portable between several makes and models of computers. If DEC wanted to, I am sure they could have a very portable Fortran.

Text:

There are for the purposes of this discussion two levels of interpreters, those at the lowest level (in Forth terminology, the inner interpreter) and those at the user interface level (the outer interpreter). Between them is the symbol table. The inner interpreter is designed with respect to the machine language of the computer. The outer interpreter is designed with respect to user facilities. The symbol table provides the communication between the two.

In DEC Fortran the outer level is the source language, and interactive compile, link, and execute is not supported. Most Basic's do support interactive developement, Forth definitely does, and Pascal usually does not.

History and such:

Interpreters are as old as programming. Many people have implemented them for a wide variety of purposes. I did not get interested until I had to do a ROM based LSI11/03 project. I knew I had to write it in machine language to keep the size down. I knew it would undergo revision. So, I used an interpreter for the part most likely to be modified in order to control the size and to make quick changes. It worked.

I continued to be employed using PDP11 computers. I did not do anymore ROM based LSI11 projects, so my interest in interpreters became more abstract. I became aware of the Forth language. There was USCD Pascal and its P-code. And DEC published information on their original Fortran on the PDP 11's. I also bought a home computer with many PDP11 features (the Motorola 6809).

I was studying how to make the fastest inner interpreter. I came to the conclusion that the RTS instruction will do the job if no interrupts are present. This was of such significance that I wrote it up and sent it off to the IEEE Computer magazine. Now two years later I have refined my ideas and have made a subject out of the study of inner interpreters.

The talk:

There are roughly four kinds of inner interpreters that are quite efficient. The first is micro-code. This is the coding of the instruction set of a computer. This level is generally unavailable to the indivigual and expensive at the small project level. The Burroughs 1700 series was designed for having a different micro-code for each language (Fortran, Cobol, Algol, etc). Several universities also tried it. My feeling is that micro-code has not yet become ecconomic at this level. Ie., micro-code is best used to implement machine language and that's

all.

The next is threaded code (TC). Threaded code has been around awhile. It has been used by many Fortran's and Basic's. On the PDP11 it is very efficient, on other computers it is relatively efficient. Threaded code is a list of "JSR sub" without the JSR. Ie, the opcode is dropped to save some memory and just the addresses are left. On the PDP11 one of the registers scans the address list. After each subroutine completes, the next address on the list is stuffed into PC. The means for this is JMP @(Rn)+. A one instruction interpreter. The interpreter is used inline. Instead of a subroutine ending with a RTS or JMP to the interpreter, it ends with the interpreter. Now RTS PC is equivalent to JMP @(SP)+. So this interpreter is essentially a RTS PC using another stack pointer register. This is what caught my interest. What if the RTS instruction was embellished. Could it not severe for any inner interpreter? I believe the answer is yes. I hope future computers will support this.

The thing that turns people away from threaded code is how do you handle things like variables and constants efficiently. The answer on the PDP11 is that each variable and constant has its own load and/or store subroutine. This takes more space than machine code. On the 6809 and on the VAX there is a way to avoid this redundancy. Each variable or constant has its own routine, but the routine is only a jump to the common routine for the load or store of a variable or constant of that type. On the PDP11 a jump to a routine takes two words. On the 6809 it can be done in one word. On the VAX it can be done in one address (32 bits). Space requirements:

	PDP11	6809	VAX
Variable	10	6	10
Constant	6	4	6

The 6809 comes out ahead because of direct page addressing. It assumes all the common load and store routines will fit in the direct page of 256 bytes.

Item of significance #2:

A timing comparison between machine code and threaded code is suprising. Threaded code actually executes faster. This is because the JMP @(Rn)+ does no stacking or unstacking as opposed to the combined JSR PC,loc -- RTS PC which does. Also, JMP @(Rn)+ is a single instruction as opposed to two. So threaded code is actually faster and more compact than what it replaces. If one carries this to its logical conclusion, one should only use machine code for the lowest level routines and threaded code for all higher level routines. Ie., all machine language programs (and I don't know of a program executing on a computer which is not machine code) should employ an inner interpreter for higher level functions.

The next step from threaded code is indirect threaded code (ITC). It goes back to the late 60's. It has been used in Snobol and in Forth. Its use in Forth is what has brought the

most attention. The idea is to add a level of indirection to threaded code. This allows all the load and store routines to be shared easily. Space requirements:

	PDP11	6809	VAX
Variable	6	6	10
Constant	4	4	6

The interpreter gets a little bigger. A MOV @(Rn)+,PC will do the job, but, the two instruction sequence MOV (Rn)+,R0 JMP @(R0)+ is usually used. The reason is that a variable or constant often follow the intermediate address link and may be easily accessed from R0 (at an offset of 0 or 2).

A memory optimization of TC and ITC is compressed threaded code (CTC). If the number of subroutines is less than 256 then a lookup table can be used to translate byte strings to addresses saving a factor of two on the PDP11 and a factor of four on the VAX. On the VAX if the number of subroutines is less than 65536 then a lookup table still saves a factor of two since addresses are 32 bits. Although not strictly the same, the Pascal P-code is similar in using 8-bit codes.

Another factor involued in interpreter design is memory organization. The approach used by Forth is very significant. Forth uses two working stacks. One is used for temporary operands and results and which I call the value stack. The other is used for "temporary" return addresses and loop counters. I call it the control stack. The significance of the second stack is that subroutine calls are simplified and unified. The traditional stack operator takes the top two items on a stack and combines them in some fashion. The traditional user operator (user written subroutine) places the two operands on the stack, does the JSR stacking a return address on top of the operands. This places a non-operand between the top of the stack and the operands, a departure from the pure stack discipline. Thus, the user or the compiler can not write subroutines obeying the pure stack discipline. However, with the return address on another stack, user and compiler subroutines do obey the pure stack discipline. This allows user subroutines to be considered extensions of the stack oriented instruction set (all instructions correspond to addresses).

Item of significance #3:

A previous paragraph recommending the use of interpreters for all higher level routines can now be seen in the double stack discipline as an extension of the machine language instruction set. Ie., the TC, ITC, and CTC interpreters under the double stack discipline serve as an extensible machine language. (Lament)For the last 20 years the programming language enthusiasts have thought about extensibility at the language level, never considering extensibility at the most basic level. If they had studied extensibility at the machine language level they might have learned how extensibility should be done at the language level.

Control constructs:

Control mechanisms are a little different. Loop counts are best kept on the control stack. This is possible since well nested program structure does not jump into the middle of loops. Both the loop index variable and the limit may be kept on the top of the control stack.

Conditionals and case statements come in several varieties depending on what is appropriate. Conditionals do a test on the top of the value stack and may either ascend, descend, or skip based on the test results. Case statements may utilize an inline jump table or may access a jump table stored elsewhere.

Outer interpreter:

We are now ready to leave the subject of inner interpreters and go on to the remaining subjects of outer interpreters and the symbol table. We will cover outer interpreters in brief to motivate symbol table structure and will return after describing the symbol table structure in detail.

There are three types of lexicographic routines for use by the outer interpreter. The simpilest is to only use single character symbols. It is left as an exercise to figure out how best to support constants. It can be done (Hint, the digits must be operators). For experimental work single character symbols works just fine.

The next level of complexity is variable length symbols with space, tab, or carriage return seperation. This is what Forth uses. This means something which would be an expression in Fortran or some other algebraic language is just a symbol. This is handy for naming intermediate calculations.

The final level is that provided by the traditional lexicographic routine. Characters are classified and symbols are delineated by class transitions. The usual classes are alphabetic, numeric, singles, and nonalphanumeric:

```
SUM  X5    45  3.14    , ( ) ;    := ** <<
```

The primary purpose of the outer interpreter within the context of this article is the conversion of symbol strings into address strings. The most basic conversion is one-for-one. This is what Forth does (most of the time). It is adequate. It results in left polish notation. The more complicated conversions use a grammar of one sort or another. This results in algebraic notation. My work uses precedence grammars.

The symbol table:

The most basic level of implementation of the inner interpreter requires no run time symbol table. Code strings will have no names. This is the most compact form. TC or CTC would probably be chosen, TC for execution efficiency or CTC for space efficiency (this mode requires edit-assemble-test style of debugging, ie. it is not interactive).

The next level provides a symbol table entry consisting of the symbol, and one or two semantic entries. Using ITC and one semantic entry this is identical to standard Forth. The symbol

length field can hold one to three flag bits (with symbol length correspondingly restricted to 127, 63, or 31 characters max).

```
  t$sym         EQU -2  right end of symbol
  t$syml        EQU -1  symbol length
  t$link        EQU 0   symbol table link
  t$alt         EQU 2/4 alternate semantic pointer
  t$norm        EQU 4/6 normal semantic pointer
  t$val         EQU 6/8 start of symbol value, if needed
  t$imm         EQU 200 @t$syml, compile flag as in Forth
  t$alfg        EQU 100 @t$syml, set if there is t$alt semantic
entry
  t$msk         EQU 37  mask for t$syml field, 31 char symbols
max
```

The third level adds type checking. This is provided by preceeding every semantic routine by an operand and result type list. The format is, proceeding right-to-left from the start of the semantic routine, length of operand type list, operand type list, length of result type list, result type list. Types are 8-bit constants. A type stack is maintained in addition to the value stack. It only functions while in the outer interpreter.

The fourth level adds aliasing and overloading. This is done by allowing a new kind of symbol table entry, the semantic list element. The parent symbol's semantic pointers are the heads of one or two semantic lists. The t$alt field of the semantic list element is used to link semantic list entries. For example the two semantic lists for subtract would be the "subtract" and "negate" lists. Each would contain entries for each type of numeric operands supported. Typically, this would be byte, word, double-word, and floating point arithmetic. As opposed to giving the same symbol several meanings (semantic routines), it is also possible to give the same semantic routine to several symbols. The "clear top of stack" semantic routine could both push a zero on the stack and extend an unsigned number to a longer size. Some additional flags are required calling for an additional byte in the entry:

```
  t$clas        EQU 2   classification byte
  t$clfg        EQU 40  @t$syml, set if there is a classification
byte; t$alt, t$norm, and t$val slid over one byte
  t$nsml        EQU 200 @t$clas, set if normal entry is head of
semantic list
  t$asml        EQU 100 @t$clas, set if alternate entry head of
semantic list
```

The final level adds parsing. This is provided by adding a precedence byte following the t$clas byte and adding more codes to t$clas:

```
  t$prec        EQU 3   precedence byte, present if PSI bit set;
t$alt, t$norm, t$val slid over another byte
  t$bkt         EQU 00  @t$clas, bracket operator; operator left
and right precedence sum to 256
  t$rtl         EQU 20  @t$clas, right-to-left parsed operator;
operator right precedence one less than left
```

```
  t$ltr         EQU 40  @t$clas, left-to-right parsed operator;
operator left precedence one less than right
  t$evn         EQU 60  @t$clas, group parsed operator;  operator
left and right precedence the same
  t$xmsk        EQU t$evn mask for the above codes
  t$$p          EQU 10  @t$clas, prefix flag
  t$$s          EQU 04  @t$clas, suffix flag
  t$$i          EQU 02  @t$clas, infix flag
  t$$imm        EQU 1   @t$clas, operator to be executed after
parse and not compiled
```

The above symbol table entry structure is modestly complicated. If support for parsing is not required, a more Forth like structure is possible which supports one/two semantic routines, overloading, and type checking. The third flag in t$syml is t$dir instead of t$clfg. It signifies that the t$norm and/or t$alt entries are heads of overloadings symbol lists. These lists use t$link for linking and normally do not have a t$alt entry. In fact, their only required fields are t$link and t$norm.

Item of significance #4:

It is amazing that Forth does so much with so little structure. It has tsym, tsyml, timm, tnorm, and t$val. It is difficult to see how one could get by with less.

Summary:

It is characteristic of interpreters that they are very portable. This is certainly the case with Forth and USCD Pascal. Many research lanaguages and operating systems use interpreters for this reason. Forth and Basic show that interpreters can be very interactive. I have tried to show that several kinds of interpreters are all part of the same general family.

References:

1 Kogge, Peter. Mar. 1982. An architectural trail to threaded-code systems. IEEE Computer, pp22-32. (survey, good references)

2 Brodie, L. 1981. Starting FORTH. Englewood Cliffs, N.J.: Prentice- Hall. (best introduction to Forth)

3 Ritter, T., and G. Walker. Sept. 1980. Varieties of threaded code for language implementation. Byte, pp.206-227. (about 40 references on interpreters)

4 Flynn, Michael. 1977. The interpretive interface: Resources and program representation in computer organization, pp41. In High Speed Computer and Algorithm Organization, Kuck, Lawrie, and Sameh (eds.). New York: Academic Press. (advanced compression techniques)

5 Brender, Ronald. 1978. Turning cousins into sisters: An

example of software smoothing of hardware differences, pp365. In Computer Engineer- ing, A DEC View of Hardware Systems Design, Bell, Mudge, and McNamara (eds.). Bedford, Mass: Digital Press. (Fortran runtime interpreter)

6 Bell, J.R. Jun. 1973. Threaded code. Comm. ACM, pp370-372.

7 Dewar, R.B. Jun. 1975. Indirect threaded code. Comm. ACM, pp330-331.

8 Fritzson, Richard. Feb. & Mar. 1981. Write your own FORTH interpreter. Microcomputing Journal. pp76 and pp44, respectively.

```
                VAX interpreters
TC      next=   JMP     @(Rn)+

ITC     next=   MOVL    (Rn)+,RØ
                JMP     @(RØ)+

CTC     next=   JMP     (Rm) -> CASEB   (Rn)+,#Ø,#limit
                tbl:    .WORD   ASD-tbl,LD-tbl,ST-tbl,...

                PDP11 interpreters
TC      next=   JMP     @(R4)+

ITC     next=   MOV     (R4)+,RØ
                JMP     @(RØ)+

CTC     next=   JMP     nxt ->  MOVB    (R4)+,RØ
                                ASL     RØ
                                JMP     @tbl(RØ)
                        tbl:    .WORD   ASD,LD,ST,...

                6809 interpreters
TC      next=   JMP     [,Y++]

ITC     next=   LDX     ,Y++
                JMP     [,X]    [,X++] also possible

CTC     next=   JMP     nxt ->  LDB     ,Y+
                                LDX     #tbl
                                ABX
                                ABX
                                JMP     [,X]
                        tbl     FDB     ASD,LD,ST,...
```

(TC-ITC and CTC-ITC hybrids also possible and useful)