FORTH for AI?

Harold Carr and Robert R. Kessler

Utah Portable Artificial Intelligence Support Systems Project
Department of Computer Science
University of Utah
Salt Lake City, Utah 84112

Abstract: Is FORTH a viable language for Artificial Intelligence (AI)? At first glance, one would be inclined to answer yes since FORTH has several features in common with Lisp and Prolog, the languages predominantly used in AI. All three have some form of runtime symbol table (dictionary, database) making it possible to incrementally (re)define functions. This promotes interactive programming, where statements in the language serve as the command language. They are all weakly typed: any variable (or stack location) may have as its value any object. This makes it possible to write routines that can work on any kind of data or do generic dispatches. Both Lisp and FORTH provide methods to create new defining words so the programmer is not limited to the defining words in the language kernel. Are these (and other) similarities enough? To answer this, we show the beginnings of a simple symbolic differentiation program in both Lisp and Prolog. We then discuss the built-in features of Lisp and Prolog that are used to an advantage by the program. The lack of these features makes standard FORTH a poor choice for AI research, at least for symbolic processing. However, we do point out that FORTH may be useful as a "delivery vehicle" for well understood programs.

Common Features: All three languages provide an interactive user interface. In FORTH, this interface is based on a reader which either compiles and installs words into the dictionary, or finds and runs previously defined words. This is analogous to Lisp's runtime symbol table and Prolog's database. This promotes a style of incremental program development in which partial programs are run and debugged, relying on stubs or the debugger to provide results from undefined functions. When a bug is found, the offending word/function/clause is fixed and the dictionary/symbol table/database is changed to include the corrected definition.

All three languages are weakly typed. This means that storage locations are not restricted to certain types: they may hold integers, characters, string pointers, etc. In Lisp and Prolog, runtime tags cause the system to dispatch to the correct routine. In FORTH, it is the programmer's responsibility to know how he wants to use an item at any particular time.

One of the most powerful features of FORTH is its ability to define new defining words. The standard example is defining a defining word CARRAY to create automatic vectors:

```
: carray    ( array-size --name)
   create allot
   does> ( index -- byte-address ) + ( Add index to base. ) ;
```

With this word you can define a 52 element array: `52 carray cards`. Storing and retrieving from this automatic array is then done by: `0 17 card c!` and `17 card c@`. Lisp's macro facility can be used to provide this capability:

```
(defmacro carray (name size)
  `(progn
     (setf ,name (make-array ,size))
     (defmacro ,name (index)
       `(aref ,',name ,index))))
```

The equivalent Lisp usage would be (without the need for addresses): `(carray cards 52)`

to create the array; `(setf (cards 17) 0)` and `(cards 17)` to store and retrieve.

All three language promote a style of programming in which programs are composed of many small functions. This is based on the fact that function calls are implemented very efficiently in FORTH, Lisp and Prolog. A more in-depth look at similarities between FORTH, Prolog and Lisp is beyond the scope of this paper, but we do point out that most of the common features are features that support interactive programming. Is this enough for AI programming? To answer this we now turn our attention to symbolic processing, a critical component of any AI language.

Support for Symbolic Processing: The key to symbolic processing is the ability to create and manipulate arbitrary structures of symbols. We begin this section with an example: symbolic differentiation - an operation that converts an algebraic expression into another algebraic expression. We write the derivative of expression `U` with respect to variable `x` as: `dU/dx`. Some "rules" or "transformation templates" for differentiation are:

```
dc/dx      =' 0 (c a constant or variable different from x)
dx/dx      = 1
d(U+V)/dx =   dU/dx +   dV/dx
d(U-V)/dx =   dU/dx -   dV/dx
d(U*V)/dx = U*dV/dx + V*dU/dx
```

Both Prolog and Lisp can easily express these transformations:

Prolog:

```
deriv(C,X,0)              :- number(C).
deriv(C,X,0)              :- atom(C), C \== X.
deriv(X,X,1).
deriv(U+V,X,A+B)          :- deriv(U,X,A), deriv(V,X,B).
deriv(U-V,X,A-B)          :- deriv(U,X,A), deriv(V,X,B).
deriv(U*V,X,U*B+V*A)      :- deriv(U,X,A), deriv(V,X,B).
```

Lisp:

```
(defun deriv (exp var)
  (cond ((numberp exp) 0)
        ((atom exp)
         (if (equal exp var) 1 0))
        ((equal exp var) 1)
        (t
         (case (car exp)
          (+ (list '+ (deriv (second exp) var)
                      (deriv (third  exp) var)))
          (- (list '- (deriv (second exp) var)
                      (deriv (third  exp) var)))
          (* (list '+
                   (list '* (second exp)
                            (deriv (third exp) var))
                   (list '* (third exp)
                            (deriv (second exp) var)))))))))
```

By running these programs on some sample input:

Prolog:

```
deriv(10,x,A).                  => A = 0
deriv(y,x,A).                   => A = 0
deriv(x,x,A).                   => A = 1
deriv(x+1,x,A).                 => A = 1+0
deriv(x*y,x,A).                 => A = x*0+y*1
deriv((x+10)*(x*y),x,A).        => A = (x+10)*(x*0+y*1)+x*y*(1+0)
```

Lisp:

```
(deriv 10 'x)                   => 0
(deriv 'y 'x)                   => 0
(deriv 'x 'x)                   => 1
(deriv '(+ x 1) 'x)             => (+ 1 0)
(deriv '(* x y) 'x)             => (+ (* X 0) (* Y 1))
(deriv '(* (+ x 10) (* x y)) 'x) => (+ (* (+ X 10)
                                       (+ (* X 0) (* Y 1)))
                                    (* (* X Y) (+ 1 0)))
```

we see correct results. We could add other operations (such as algebraic simplification, data-

driven dispatching, data abstraction, and a complete set of transformations) but these short prototype programs are sufficient for our purposes. The question we wish to answer is: *What built-in features of Lisp and Prolog are used in these programs, and are these features present in FORTH?*

Symbols: Both Lisp and Prolog have automatic facilities for the creation and manipulation of symbols. Symbols are distinct from variables. In the programs above: the `C` and `X` in `deriv(C,X,0)`, and the `exp` and `var` in `(defun deriv (exp var) ... )` are variables used to pass information into the procedures. The symbols are most clearly seen when we run the program. The `y` and `x` is `deriv(y,x,A)`, and the `*`, `x`, and `y` in `(deriv '(* x y) 'x)` are symbols. A FORTH programmer may declare named constants and then use their symbolic name throughout the program, but it is the programmer's responsibility to make sure that no two distinct symbols have the same value. Further, if it is possible for the user of the program to input any arbitrary symbol, the FORTH programmer will have to explicitly make provisions for this possibility, whereas this facility is built-in to both Lisp and Prolog. For example, we can input `deriv(a*b,a,A)` . to our `deriv` program and obtain an answer (`A = a*0+b*1`) even though the symbols `a` and `b` are not explicitly declared in the program. The system takes care of this for us. Structures of symbols may be used to represent relations as in: `(is-a-fruit pear)`, which brings us to our next point.

Lists: We see structures of symbols in both the input and output of our `deriv` program. Internally, these structures are represented by list structure. Any kind of structured data can be represented by this single general type. The Lisp and Prolog readers/printers automatically convert between the external and internal representation of lists so that a user may simple enter `'(* x y)` to create a specific list. List constructor and selector functions (`cons`, `car`, `cdr`) are an integral part of Lisp and Prolog systems as well as automatic reclaimation of lists no longer accessible by the program. There have been a number of implementations of the heap data structure in FORTH [Dress 86], but our point is that it is not part of the standard. Therefore FORTH programmers must take the time to find a copy of an implementation or redo it themselves.

Automatic Dereferencing: All values in Prolog and Lisp can be viewed as pointers to some object. These languages drop the phrase "a pointer to" since pointers are explicitly dereferenced everywhere in the program. We see this in the above program: when we input `'(* x 1)` to `deriv` we are passing a pointer to the internal list representing `'(* x 1)`. In the `deriv` function, references to the `exp` parameter automatically dereference the pointer. The same thing happens in Prolog. Contrast this with FORTH where pointers to variables or arrays must be explicitly dereferenced (`@`, `!`) to get at their contents.

Named Parameters: Although this feature is not directly related to supporting symbolic processing, we point out the fact that standard FORTH programmers must concern themselves with the intellectual burden of managing the parameter stack. Traditionally, AI programs have been large and complex. Lisp and Prolog support complex program development by freeing the programmer from this unnecessary detail by automatically handling the parameter stack.

Besides these salient features, we should mention even more powerful features found in Lisp and Prolog which are missing from FORTH. Prolog: bi-directional parameters; procedures may return results containing unbound variables; unification (which combines the effect of both conditionals and assignment); backtracking (which allows the generation of multiple solutions to a problem and easy restoration of the program state when failed paths have been pursued). Lisp: the combination of quoting with the user accessibility of `eval` and `apply` makes it possible to manipulate programs as data and to execute data as programs (which facilitates the implementation of embedded languages); syntax extensibility via readtables; full lexical scoping encapsulates data (which may be used to create infinite data objects [Odette 84]).

Since standard FORTH is missing the features discussed in this section one is inclined to conclude that it is a poor choice for AI programming. However, both Lisp and Prolog have some drawbacks which the use of FORTH can solve in some circumstances. That is the subject of the next section.

FORTH as a Delivery Vehicle: One advantage that FORTH has over Lisp and Prolog at this time is scalability: once a program has been developed it is fairly easy to produce a standalone, executable object from FORTH code, whereas both Lisp and Prolog require extensive runtime support. We would not recommend using FORTH for AI research because of the limitations cited above. But once a program is well-behaved and well-understood there may be some benefit rewriting it in FORTH for delivery on smaller machines. Another advantage of FORTH over Lisp and Prolog (although this advantage is shrinking with the availability of better Lisp and Prolog compilers [Kessler 86]) is its ability pack data into bit and multi-bit fields without superfluous pointers, and to access this data through "open-coded" functions.

Conclusion: Both Lisp and Prolog provide built-in facilities that promote symbolic programming. The lack of these features in standard FORTH makes symbolic programming a more time-consuming and error-prone task, and therefore a poor choice for AI programming. FORTH may be useful as a delivery vehicle for well-understood programs.

Acknowledgments: Work supported in part by the Burroughs Corporation, the Hewlett Packard Company, the International Business Machines Corporation, the National Science Foundation under Grant Numbers MCS81-21750 and MCS82-04247, and the Defense Advanced Research Projects Agency under contract number DAAK11-84-K-0017.

References:

[Dress 86] Dress, W. B.
A FORTH Implementation of the Heap Data Structure for Memory Management.
Journal of Forth Application and Research 3(3):39-49, July, 1986.

[Kessler 86] Kessler, R. R.; Carr, H.; et. al.
EPIC - A Retargetable, Highly Optimizing Lisp Compiler.
In *Proceedings of the SIGPLAN 1986 Symposium on Compiler Construction.* ACM SIGPLAN, June, 1986.

[Odette 84] Odette, L. L.
Computing with Streams.
Dr. Dobb's Journal 9(9):50-67, September, 1984.