Embeddings of Languages in Forth

R. D. Dixon

Department of Computer Science
Wright State University
Dayton, Ohio 45435

Introduction

Recent Forth literature contains so many papers on embeddings of other languages in Forth that the reader might ask, "Does anyone do real work in Forth anymore?"The answer is while a large amount of important production work is still going on in Forth, there is also a great deal of innovation, particularly in real-time applications. Many of these innovations lead to the definition of new or existing languages in Forth as the easiest and best way to do the job at hand ([DRE86]; [HAR85]; [PAL87]; [RED86]). The importance of real-time applications in manufacturing, science and national defense means Forth machines and other languages written in Forth are more significant than one might expect. This paper explores three important aspects of languages in which Forth and its extensions can play a prominent role: conciseness, conceptual models, and extensibility.

High Information Content in Languages

In the middle ages (1970s), computer science was chiefly a science of avoiding concise notation. Programming was an exercise in redundancy and low information content. Structure was all important as was protecting the user from his or her own intentions. We may speculate on the causes of this movement, but one fact seems clear: there were many programming tasks to be done and little man/womanpower available. Techniques that were successful for untrained personnel were favored.

Although low-density code still has its market and its place, it is doomed as the universal language of computing for several reasons. First, it is simply impossible to program some sophisticated, complicated applications in such fine-granule languages. Consider also the maintenance of a 1000-page program in Pascal versus a 10-page version of the same program in Prolog or LISP. Such condensation is possible, but, of course, understanding a single line in the concise notation may be a task of great concentration and skill whereas, in the low-density code, one spends more time flipping pages.

We have all experienced or heard of differences in programmer productivity in ratios as high as 1:10. The ratio between the productivity of a programmer who can only think and program in low-density languages, to that of one who can use a high-density language well suited to the problem space may well be 1:100 on certain problems.

Forth may be slow to gain some acceptance because of the range of densities in which it can be used--all the way from an assembler language for its virtual machine to very high language embeddings. For the experienced user, this flexibility in power is the tool that allows the productivity needed to get the job done. As the power of machines increases, and the demands put on real-time systems rise, other languages have become less well adapted as system solutions. Efforts such as Modula and Ada have yet to prove their value. Forth has an increased opportunity, but it requires a sophisticated and broad-based development as a language.

When we consider the features of Forth that are found in few other languages, it might be expected that they come at a cost. One criticism of these features seems to be that they are integrated

into the language in such a way that a novice can use them to create programs and structures which have a high information content. By high information content, I mean the number and variety of abstract ideas, machine instructions, data structures, and user interactions that can be invoked by a single symbol or string of symbols. In other languages, some of these features either are less general in their application or require a more sophisticated user to obtain access to the facility.

The interactive nature of Forth at run time allows incremental compilation and direct user calling of procedures. This environment has existed in only a few languages like BASIC and APL but is now being found in implementations of LISP, Prolog, and Smalltalk.

The `create...does>` construct allows a form of data structure packaging. It is not unlike those found in Pascal, Ada, or Smalltalk. As far as I can see, there is little criticism of this packaging approach in any of the other languages.

Forth, like most computer languages, is based on subroutine calls. Every time we define a new subroutine we extend the semantic constructs available to the user of a system. Those who view semantic extensibility as a liability as well as an asset must face it in any reasonable language. The Forth user's ability to define immediate words which affect the compiler, and thereby the syntax of the language, is unique, although macros and preprocessors have some similar characteristics, and LISP and Prolog can manipulate program text. Thus, syntactic extensibility is a more controversial feature. However, the change in the compiler usually exercised in Forth is still more controlled than the run-time construction of executable statements sometimes found in LISP and Prolog.

Languages with a Pure Conceptual Model

Computing never stands still. Computer languages evolve, and new languages emerge. If the reasons why a particular language is popular are based on a technology that is obsolete, then the language may also be obsolete. Some languages are ahead of their time and are popular in anticipation of a technology that does not yet exist. It is possible that Forth can be described by any or all of these statements.

Purity of the underlying computational model is appealing in a language in that it gives hope that resulting programs will be understandable and that the implementations can be simple. Unfortunately, the world is not simple, and we view it in a variety of ways in order to understand it. Computing is a young science, and for the best of computing, we frequently look to adaptations of older fields. Linguistics, logic, mathematics, and engineering give us different pictures of the world, and we have responded with different languages in which to describe these world views.

One common theme that runs through these differing approaches is that of hierarchy. The basic relationship of hierarchy to abstraction is central to scientific thought. Hierarchical relationships, however, are multiple and varied, some being conceptual and some operational. Early thinking in computer science followed strictly hierarchical lines resulting in languages such as ALGOL 60. Real-time control was a notable exception because the problem forced accommodation. Early sequential machines failed to be economical controllers. Not until interrupts were introduced did sequential machines become effective at this task. The ALGOL 60 model of nested blocks was not helpful in thinking about concurrency.

The actor model, which led to the language Smalltalk [DUF84,86], is an attempt to balance the conceptual hierarchical view and an independent object- or task-oriented operational approach. It is interesting that it has been difficult to implement the model efficiently because it has failed to restrict the hierarchical issues to those that could be settled at compile time rather than during run time.

Functional languages such as "pure LISP"or Backus' FP have a conceptual model based on mathematical function theory [BEL87]. This approach presents the working programmer with difficulties because the model fails to recognize state information which is independent of the operational hierarchy. Further, the LISP model of memory is basically sequential and fails to take advantage of the direct access provided by most computer memories.

Prolog, which is a logic programming language, shares many things with the LISP model [ODE87a and b]. Its implementation is search oriented while its conceptual basis is nonprocedural. The result is that for procedural tasks not of the nature of its search strategy, it is not efficient.

The point here is not to prove that conceptual models are useless, but rather to show that there are difficulties in choosing a single model in a world where people think in a number of different ways and use machines that are different from all of them. Forth has a pure computational model, the stack machine, but few Forth programmers stay within that model. The availability of a mapping of memory and of access to the compiler allows the programmer to give the illusion of conceptual consistency while doing whatever is necessary to make the system run. Even at that, the use of the Forth computational model, which differs from the host machine and which requires an interpreter, extracts a high price. The concept of a Forth machine may eliminate that price. If Forth machines can be efficient on Forth-like things and also efficient on other languages, then they may have a basis for support in a wide market.

Languages with Extensibility

Forth as a low-level language has several markets in which it has been and continues to be the most convenient solution. As a middle-level language competing with C, Pascal, and FORTRAN it has met with considerable resistance from management and academics. As a very high-level language it is currently overshadowed by both LISP and Prolog, for which reasonable compiled implementations have emerged. None of the popular languages, however, can compete with Forth on all these levels.

The embeddings of higher level languages in Forth, when carried out completely, seem to be transitory. It is a nice exercise to write a LISP, a Smalltalk, or a Prolog in Forth, but in the long run someone (or in fact the Forth implementor) will go to a direct implementation which bypasses much of the Forth ([DUF84,86]; [ODE87a and b]). However, the inclusion of a Prolog, OPS5, or some other rule-based language implementation in a real-time application, which is being done in an overall Forth environment, is unique and exciting and means that complicated user interfaces can be easily programmed ([DRE86]; [HAR85]; [LEW86]; [ODE87b]; [PAL87]; [PAR86]; [RED86]). Including some Smalltalk or Ada-like objects or packages in Forth is feasible and adds to the general weapons available to make data structures virtual; hence, ease changes in various aspects of the system [DUF84].

The development of Forth machines has given added incentive to the development of very high-level languages in Forth because on these machines, a Forth implementation is a direct implementation [ODE87a]. These machines have also created a new market in that they need a middle-level language acceptable to a wide class of users. Embeddings of such languages have been done ([EPS85]; [MOR85]). While a traditional compiled C may make sense on these machines, to make all Forth packages attractive we need to develop middle-level languages that are embedded in Forth, incrementally compiled, and accessible in the usual Forth environment to augment lower-level Forth and very high level embeddings. Techniques to do this and still issue good code are available. A base language, C-like in structure, could be adapted and would add to the marketability of the total Forth system.

The distribution of Forth systems and modules of Forth systems which include or depend on sub-language modules is a complex software engineering problem. New techniques of software generation must be considered [DIX87], and this goal needs to be a major consideration in further Forth standards efforts.

Conclusions

The embedding of languages in Forth is serving several purposes. These embeddings are useful and the best solutions to certain immediate problems. They serve as the mutants in the evolutionary process from which the future generations of Forths will be derived, and they test the Forth machine architecture, suggesting its strengths and its weaknesses.

Finally, the impact of developments in computing fall in scattered and random ways within certain predictable boundaries. The advance of computer hardware and other technological developments are very well charted, and we have every reason to believe that the predictions are, in fact, conservative. The history of computing tells us that software and systems move forward by having a broad base of independent users doing a wide variety of things. The Forth community does just that, and it has in its hands the most flexible tool to be found.

References

[BEL87] Belinfante, Johan G. H. 1987. S/K/ID: Combinators in Forth. *J. Forth Appl. and Res.* 4(4), this issue.

[DIX87] Dixon, R. D., and Hemmendinger, David. 1987. Compiling and analyzing Forth in Prolog. *J. Forth Appl. and Res.* 4(4), this issue.

[DRE86] Dress, W. B. 1986. REAL-OPS: A real-time engineering applications language for writing expert systems. *J. Forth Appl. and Res.* 4(2):113-24.

[DUF84] Duff, Charles B., and Iverson, Norman D. 1984. Forth meets Smalltalk. *J. Forth Appl. and Res.* 2(3):7-26.

[DUF86] Duff, Charles B. 1986. Actor, a threaded object-oriented language. *J. Forth Appl. and Res.* 4(2):155-60.

[EPS85] Epstein, Arnold. 1985. The MAGIC/L programming language. *J. Forth Appl. and Res.* 3(2):9-22.

[HAR85] Harris, Henry M. 1985. Forth as the basis for an integrated operations environment. *J. Forth Appl. and Res.* 3(2):23-36.

[LEW86] Lewis, Steven M. 1986. Tokenized rule based system. *J. Forth Appl. and Res.* 4(1):29-46.

[MAT86] Matheus, Christopher J. 1986. The internals of FORPS: A FORth-based Production System. *J. Forth Appl. and Res.* 4(1):7-28.

[MOR85] Moreton, Pierre. 1985. HFORTH: A high level business language in FORTH. *J. Forth Appl. and Res.* 3(2):37-45.

[ODE87a] Odette, L. L. 1987. Compiling Prolog to Forth. *J. Forth Appl. and Res.* 4(4), this issue.

[ODE87b] Odette, L. L., and Paloski, W. H. 1987. Use of a Forth-based Prolog for real-time expert systems. II. A full Prolog interpreter embedded in Forth. *J. Forth Appl. and Res.* 4(4), this issue.

[PAL87] Paloski, William H., Odette, Louis L., Krever, Alfred J., and West, Allison K. 1987. Use of a Forth-based Prolog for real-time expert systems. I. Spacelab life sciences experiment application. *J. Forth Appl. and Res.* 4(4), this issue.

[PAR86] Park, Jack. 1986. Toward the development of a real-time expert system. *J. Forth Appl. and Res.* 4(2):133-43.

[RED86] Redington, Dana. 1986. A Forth oriented real-time expert system for sleep staging: A FORTES Polysomnographer. *J. Forth Appl. and Res.* 4(1):47-56.