EXTENSIBILITY IN PROGRAMMING LANGUAGE DESIGN

Thomas A. Standish

Technical Report #60 - January 3, 1975

Invited Paper

Prepared for the 1975 National Computer Conference, May 19-22, 1975

Extensibility in Programming Language Design

A b s t r a c t

This paper assesses the value of including extensibility as a feature of a programming language design. Extensibility means permitting language users to define new language features. Starting with a base language and using various definition facilities an extensible language user can create new notations, new data structures, new operations, and, sometimes, new regimes of control. Given enough insight and craftsmanship, the extensible language user can often create language extensions that are well-adapted to given intended application areas and are useful for writing concise, clear algorithms that are free from contamination with low-level detail. However, experience has revealed that this approach is not as promising as was first hoped. The paper explores the various types of extensibility that have been introduced in the extensible language movement, and it attempts to assess the practical limitations on the use of extensibility. It concludes that extensibility has many more dimensions than was first expected and that extensibility is not the revolutionary idea that some once hoped it would be.

Extensibility in programming language design

by THOMAS A. STANDISH
University of California at Irvine
Irvine, California

INTRODUCTION

What is extensibility? What is it good for, if anything? Is it worth bothering about when designing a programming language?

Simply put, extensibility permits programming language users to define new languages features. Starting with a *base language* and using various *definition facilities*,[1] an extensible language user can create new notations, new data structures, new operations, and, sometimes, new regimes of control. To a certain extent, extensibility permits a user to modify the features of a language to suit his changing needs and purposes. Given enough insight and craftsmanship, the user can create language extensions which are well-adapted to given intended application areas, and are useful for writing concise, clear algorithms free from contamination with low-level detail. However, experience has revealed that this approach is not as promising as was first hoped because of certain practical limitations I shall try to expose in this paper.

WHAT WERE THE EARLY ASPIRATIONS?

One of the earliest expressions of an extensible language philosophy was given by Brooker and Morris in 1960 in their classical paper on the Compiler-Compiler:[2]

> *The system is extendable and allows the user to define the meaning of new formats in terms of existing formats as well as in terms of basic assembly instructions (whose meaning is built in).*
>
> *It is unlikely that every machine user will want to write his own autocode: what is more likely is that he may wish to extend one of the standard languages to include statements suited to his own problem area.*

McIlroy seems to have had much the same idea in mind in 1960 when he explained his aim in introducing assembly language macros:[3]

> *It is our aim to show a limited set of functions readily implemented for a wide variety of programming systems which constitute a powerful tool for extending source languages conveniently and at will.*

In May of 1969, at the Extensible Languages Symposium, Christensen's chairman's introduction characterized the objectives of extensibility as follows:[4]

> *The ultimate and idealized objective of extensible languages is simple and attractive. A single universal programming system is postulated which is included in the software support for every general purpose computer. This programming system is not limited to one particular programming language such as PL/I. Rather, it includes a* base language *and a* meta-language. *A program in this system consists of, first, statements in the meta-language which expand, contract, or otherwise modify the definition of the base language to produce a* derived language, *and, second, statements in the derived language which constitute the executable part of the program.*
>
> *Thus the system includes facilities to define and then to program in a limitless variety of programming languages—languages which are used for business, scientific, or systems applications, languages which may be simple or complex.*

It seems fair to state that some of us were gripped by a curious euphoria in those days. A number of us believed that users would be able to extend the base of an extensible language rapidly and cheaply to encompass the data, operations, notation, and control natural to many diverse application areas. In short, we believed we could make it possible for unsophisticated users to manufacture personalized languages of reasonable efficiency and substantial utility with great ease just by applying some easily learned extension techniques. These beliefs were probably a bit over-ambitious.

EXTENSION TECHNIQUES—A BRIEF CATALOGUE

There are apparently quite a few dimensions to extensibility—more than we at first suspected. Let me give brief examples to illustrate the diversity.

Paraphrase

Paraphrase is a form of definition in which we define something new by showing how to exchange it for something whose meaning is known (or will become known after giving

further definitions). This is commonly used in natural language to define the meaning of new words (e.g., thaumatology = the study or lore of miracles). Paraphrase is used widely as an underlying technique in extensible languages, and it takes many forms. As examples, consider the following:

(a) *macros* in assembly language. [3] E.g.,

```
MACDEF  SUM A, B, C
    LOAD  A
    ADD  B
    STORE  C
ENDDEF
```

(b) *procedure definitions* in algebraic languages, [4] E.g.,

integer procedure Factorial (N); *integer* N;
Factorial := *if* N = 0 *then* 1 *else* N*Factorial (N-1)

(c) *syntax macros* [9] or grammatical extensions at the expression and statement level (as in reference 2), E.g.,

smacro
while ⟨b:boolean expression⟩ *do* ⟨s:compound statement⟩≡
{create new label (Ll) in[
Ll: *if* b *then begin* s; *go to* Ll *end*]}

(d) *data definitions.* [6,7] Here the idea is to be able to define new sorts of composite structured data starting with atomic data (such as *integers*, *characters*, and *reals*) or previously defined composite data. E.g.,

Let a *rational* be a structure which has
a *numerator* which is an *integer* and has
a *denominator* which is an *integer*
Let a *Matrix* be a *row*(*1*:*N*) with *vector* elements

(e) *operator definitions,* [8,9] E.g.,

Let Max(A,B) = (*if* A > B *then* A *else* B)
Let ++ be a *left associative, binary* operator with
meaning = Max and *precedence* > "+"

(f) *control structure extensions,* [10,11] Here one can use process definition facilities to confer attributes on processes at the time of their definition. This is useful for defining processes that act as co-routines, that backtrack, that execute concurrently, which monitor for the occurrence of certain events and seize control when these events happen, and so forth.

Orthophrase

Orthophrase is a means of extension wherein we add "orthogonal" features to a language. An orthogonal feature is one which lies outside the space of features expressible by paraphrase. Its defining expression cannot be composed in the language. For example, suppose we are given a language *L* in which definition facilities (a)-(f) above are available but which lacks I/O device control primitives. We might not then be able to extend *L* by writing expressions in *L* itself to specify programs for reading disk files or printing on a line printer. Adding a file system, a real-time clock, a string valued function giving today's date, or adding call-by-reference parameter passing are four additional examples of the sort of features that cannot usually be defined by paraphrase if some basis for defining them is not already in the language. To add an orthogonal feature, one must normally perform surgery on the underlying guts of a language processor and patch it in. (The adjective "orthogonal" seems to have been borrowed by analogy to vector spaces, wherein a vector orthogonal to each of the vectors in a given basis set is one which can't be expressed as a linear combination of the basis set elements and which lies outside the space "spanned" by the basis elements).

Thus, orthophrase seems to be defined *relative* to a given set of paraphrastic definition facilities. It is not an absolute notion. Paraphrase facilities and an associated base language have a "span", so to speak, which consists of all possible paraphrastic extensions of the base. Any feature lying outside this "span" must be added by orthophrase, but a feature one must add by orthophrase in one language may be reachable by paraphrase in some stronger extensible language.

Metaphrase

While paraphrase and orthophrase add new capabilities, they do not change what is there before. *Metaphrase* consists of altering the interpretation rules of a language so that it processes old expressions in new ways. Examples of metaphrase are changing scoping policies for lifetimes or bindings of variables, changing the meaning of parameter evaluation, and changing the meaning of *assignment* and *go to* statements to allow programs to run backwards (useful perhaps in debugging to locate the source of an error after noticing incorrect program behavior).

WHAT HAVE WE LEARNED ABOUT EXTENSIBILITY?

By my latest count, 27 extensible languages have been proposed by 55 people (see Reference 12 for examples). Some of these proposed languages were implemented and, in some cases, [13,14] considerable experience has accumulated. What did we learn?

As you might expect, different categories of extensions can require widely different amounts of skill and labor. Some are hard and some are easy. Which kinds have which properties?

Let's first discuss extensions that can be made with modest amounts of labor by unsophisticated users. For example, in a well-designed extensible language, [15] it is straight-forward to extend the real and integer arithmetic available in the base language to include rational, complex, or multi-precision arithmetic or to add matrix, vector, or formula manipulation. While these feats are easy, there is more in them than at first meets the eye.

For example, how do we arrange to share the syntax of arithmetic expressions among all these domains (so that

A+B*C could be used to specify operations on integers, matrices or complex numbers where appropriate)? How can we define arithmetic concisely on the huge space of mixed types (such as "a rational" + "a complex" or "a real" * "a matrix")? How do we augment the lexical analyzer to recognize and translate new expression forms for constants (such as "3+4i" for a complex constant)? Can we seal off the behaviors of underlying representations so users transact only with data types in the extension (such as preventing the user from tinkering with lower triangular arrays used as a substrate to represent symmetric matrices in the surface of the language)?

Other examples of extensions that can be done with a modest amount of labor include adding strings, lists, balanced binary trees and file records together with their appropriate operations, behaviors and notations. The extensible language experience [14] has revealed that not only must we be able to add typical combining, growth, and decay operators (such as concatenation, insertion, and deletion) and to express them using new or shared notations, we must also be able incrementally to extend common background language functions such as printing, assignment, and selection to handle special behaviors required of each new type (e.g., reduction of rational numbers to "least common terms" each time they are assigned to variables, or printing balanced binary trees in two dimensions if they will fit on a printed page, etc.)

In each of these extensions the "style" of the base language tends to be preserved unaltered. For example, none of these extensions affect conventions for the presence or absence of block structure, the presence or absence of declarations, or for the form of conditional and iterative expressions or for parameter evaluation.

Examples of hard extensions are: trying to add block structure to a language which doesn't have it (as in attempting to extend BASIC or FORTRAN to become ALGOL 60), adding declarations if they aren't there beforehand, adding backtracking or concurrent processing to the control structure if they aren't there already, or adding a real-time clock or a file system if they weren't there previously.

In oversimplified terms, the easy extensions seem to be those that use paraphrase to add new features, whereas the hard extensions seem to be those that use orthophrase or metaphrase to modify what was there before.

More precisely, in the case of paraphrase, the space of definitions users can give has been provided for explicitly in the extensible language design. Such extensions often specify independent additions to the base language and leave all base language features constant.

In the cases of orthophrase and metaphrase, one normally has to modify a description of the language processor. These descriptions are usually complex and considerable knowledge and sophistication are required of the user desirous of making alterations. It is often hard to add something new in an non-injurious way, so it blends smoothly and doesn't upset a collection of mutually dependent behaviors.

Here the extensible language experience reinforces the familiar notion that complex systems are resistant to change. The more intricate they are, the less easy it is to find out how to alter them significantly. Not only does this seem characteristic of the use of orthophrase and metaphrase, it also seems characteristic of cascades of extensions constructed by paraphrase—the more the intracacy of a set of extensions the more the difficulty of further extension.

These considerations seem to point to the conclusion that each extensible language is surrounded by an envelope of possible extensions reachable by modest amounts of labor by unsophisticated users. These easily reached extensions tend characteristically to be those which add one layer of new data and new operations and which perhaps make minor additions to notation while leaving the conventions of the base language invariant. Beyond this envelope, lie hard extensions requiring surgery on the language processor, major deformations of its conventions or style, or careful consideration of how to modify collections of mutually dependent extensions previously constructed. These seem generally to require a level of knowledge and sophistication beyond that of the casual user but perhaps attainable by the interested and committed professional. The unsophisticated user could no more likely add a file I/O facility to an extensible language not having one previously than he could add a date window to his wristwatch—both cases require a high level of investment in understanding complex descriptions and learning to use intricate tools. For all practical purposes, then, extensible languages seem only mildly extensible by unsophisticated users.

A basic difficulty with the philosophy of extensible languages derives from the fact that it is basically a "do-it-yourself-kit" philosophy. As a user, you may not want to build your own programming language any more than you may want to build your own car. Building your own car requires large investments of labor, knowledge, tools and time, and most home-builts are not very elegant. Unless you are a hobbiest, you are often willing to sacrifice control over form to obtain the benefits of prefabricated labor, especially when there is lots of choice in the marketplace. This is why most prefer to buy rather than build their cars.

So it is with extensible languages. It takes a huge dose of labor to extend a simple base to the level of capability represented by a high content language. Are users the right people to make these extensions or can skilled professional craftsman do a better job? Starting with a simple base and powerful extension facilities one often leaves the labor of building advanced features to those most likely to do an unattractive and unskilled job of it—the users.

A final difficulty revealed by the extensible language experience is that extending a simple base results often in long, thin extension cascades that are often ugly and inefficient. Look at how LISP [16] gets extended. One starts with a diamond (EVALQUOTE) and progressively corrupts it by adding warts (the PROG feature, SETQ, FUNARG, etc.). After you are finished, the extended LISP is far from simple and harmonious and you might have obtained better results by designing to meet the overall constraints all at once.

WHAT THEN IS EXTENSIBILITY GOOD FOR?

It should be obvious that the extensible language movement has succeeded only partially in meeting the objectives

stated by the early explorers and that it did not create a programming revolution. It probably did succeed in clarifying for us the amounts of labor and knowledge required to produce several sorts of language variation, and in revealing what sorts of extensions are reasonable to expect of unsophisticated users.

Irons [17] probably put the matter in reasonable perspective when he observed that extensibility bears the same relation to high level languages as macros do to assembly languages. Maybe you use macros, say, 10 percent of the time or less, but they can be very convenient when you need them. Not only can you use them to suppress low level detail and promote program conciseness and clarity—you can often use them to mask irritating features of a language someone else has supplied. Extensibility seems worthwhile for analagous reasons. If implementation resources and execution space permit, it is probably better to have extensibility than to do without it.

REFERENCES

1. Christensen, C. and C. J. Shaw, *eds.*, Proceedings of the Extensible Languages Symposium, *SIGPLAN Notices*, August 1969.
2. Brooker, R. A. and D. Morris, "A General Translation Program for Phrase Structure Languages," *JACM*, January 1962, pp. 1-10.
3. McIlroy, M. D., "Macro Instruction Extensions of Compiler Languages," *CACM*, April 1960, pp. 214-220.
4. Naur, P. *et al.*, "Revised Report on the Algorithmic Language Algol 60," *CACM*, January 1963, pp. 1-17.
5. Leavenworth, B. M., "Syntax Macros and Extended Translation," *CACM*, November 1966, pp. 790-793.
6. van Wijngaarden, A. *et al.*, "Report on the Algorithmic Language Algol 68," *Num. Math.*, *14*, 1969, pp. 29-218.
7. Landin, P. J., "The Mechanical Evaluation of Expressions," *Computer Journal*, January 1964, pp. 308-320.
8. Taft, E. A. and T. A. Standish, *PPL User's Manual*, Tech. Rept., Center for Research in Computing Technology, Harvard University, September 1970.
9. Galler, B. A. and A. J. Perlis, "A Proposal for Definitions in Algol," *CACM*, April 1967, pp. 204-219.
10. Fisher, D. A., *Control Structures for Programming Languages*, Ph.D. Thesis, Carnegie-Mellon University, Department of Computer Science, May 1970.
11. Prenner, C. J., *Multi-Path Control Structures for Programming Languages*, Ph.D. Thesis, Center for Research in Computing Technology, Harvard University, June 1972.
12. Schuman, S., *ed.*, Proceedings of the International Symposium on Extensible Languages, *SIGPLAN Notices*, December 1971.
13. Irons, E. T., "Experience with an Extensible Language," *CACM*, January 1970, pp. 31-40.
14. Cheatham, T. E. and J. A. Townley, *Some Applications of ECL*, Center for Research in Computing Technology, Harvard University, 1973.
15. Wegbreit, B., *et al.*, *ECL Programmer's Manual*, Harvard Univ., 1973.
16. McCarthy, J., *et al.*, "Lisp 1.5 Programmer's Manual," *MIT Press*, Cambridge, 1962.
17. Irons, E. T., personnal communication, February 22, 1971.