Three Uncommon Lisps

Julian Padget
**School of Mathematical Sciences,
University of Bath, Bath, Avon, UK**

Abstract

This paper surveys three Lisp systems which do not enjoy widespread popularity. Nevertheless, each has features which make them particularly suitable for certain applications and all exhibit care in certain aspects of their design which are worthy of greater recognition. The three Lisps are Portable Standard Lisp (nee Standard Lisp), Lisp/VM (nee Lisp/370) and Cambridge Lisp. In fact, the last inherits ideas from both Standard Lisp and from Lisp370 and it is interesting to compare the compromises made in Cambridge Lisp versus those in PSL. each of which is a second generation Lisp from a similar background.

1. Introduction

The obvious way to structure this paper, following from the abstract, would be to give descriptions of each system in turn. Such a travelogue style would, as well as being tedious to the reader, be relatively unenlightening, since the useful information is to be derived from how these designs have interacted and formed a group within the larger community of Lisp systems. In order to make comparisons it is still necessary to provide some background on each system and to this end the next section gives some thumbnail sketches. This is followed by three sections orthogonal to the systems to themselves, but issues that arise in all systems: semantics, portability and usability.

2. Biographies

2.1. Standard Lisp

Standard Lisp has had three stages in its life; it was not really a single implementation or perhaps even a dialect, rather it was a fairly small set of functions, the behaviours of which were first defined by Hearn [1969] and revised and extended in Marti *et al.* [1979], finally Portable Standard Lisp grew out of the 1979 report. As long as the set of operations defined in the Standard Lisp report existed in a Lisp system then that system provided a Standard Lisp compatibility mode and thus provided the minimal environment for the Reduce algebra system [Bradford *et al.*, 1986]. Reduce is written adhering to the constraints of the Standard Lisp report, which makes it a remarkably easy program to port despite its size and sophistication. To make Reduce run in Common Lisp it was simply a matter of building a package (call it Standard Lisp) with the necessary definitions.

The widespread availability of the Reduce algebra system was the main driving force behind Standard Lisp, so the underlying goal was portability, and thus the development of Portable Standard Lisp [Griss *et al.*, 1982] [PASS, 1987] was a natural consequence. Between the first Standard Lisp document and the second, it was realised that making Standard Lisp a compatibility definition was too inflexible and that the document constituted a good basis for writing new Lisp systems. Subsequently various implementations of Standard Lisp were built (SLISP/360 which was based on Stanford Lisp for the IBM/360 and Standard Lisp for PDP-10, PDP-20 and various Burroughs machines) and still run in many parts of the world. SLISP/360, for instance, is widely used to support Reduce in the eastern bloc.

What irked about all these implementations of Standard Lisp was the amount of effort involved in developing each system when there should be so much information in common. This realisation spurred the development of Portable Standard Lisp as a system intended to be used for experiments in Lisp systems design and as a system in which the majority of the code could be reused and retargetted for different machines, architectures and operating systems. The most effort in the development of PSL has been placed on factoring out system dependencies and in building a retargettable compiler (first described by Griss and Hearn [1981] and a later (but less successful) version described in Kessler *et al.* [1986]). The purpose of emphasising that these areas are where most effort has been expended is to aid in highlighting the strengths and deficiencies of PSL against those of the other Lisp systems described in this paper. Other areas, such as semantics and usability have suffered because they were not primary concerns.

2.2. Lisp/VM

Lisp/VM is the antithesis of Portable Standard Lisp in that great care has been paid to the definition and the consistency of the semantics and a corollary effect has been the provision of very detailed debugging information and a powerful unusual debugging environment. Lisp/VM can trace its roots back to the earliest Lisp systems on the IBM 703 and IBM 704, but work really started in 1974 and resulted in the implementation of Lisp/370 which is described by Blair [1976].

The early work on Scheme [Sussman & Steele, 1975] was also a major influence on the design of Lisp/370 and thus Lisp/VM is one of the longest-lived Lisps which has always had only one value for each name and which uses lexical scoping by default. Lisp/VM also provides dynamically scoped variables and a full funarg mechanism for both lexical and dynamic closures by means of a *pasta* stack mechanism based on the model described by Bobrow and Wegbreit [1972] and extended by Steele [1977].

Another notable aspect of the design of Lisp/VM was the use of a formal model to describe the actions of the interpreter. The interpreter is described by a much extended version of Landin's SECD machine [Landin, 1967] and the implementation mirrors this faithfully by comprising solely of routines which act on the four components of that machine. The compiler for Lisp/VM attempts to mimic the action of the interpreter as closely as possible so that interpreted and compiled code do behave in the same way. It is a remarkable comment on Lisp systems in general that such a property is the exception rather than the norm. Note that it was still necessary to qualify the accuracy of the compiler with *as closely as possible*; this is because macros make it feasible to defeat the system implementors' attempts at interpreted/compiled consistency.

Lisp/VM also takes great pains to provide both security and consistency by using the multiple environment facility of the branching stack. For example, the compiler resides in an environment disjoint from the user's environment so that neither may interfere inadvertently or maliciously with one another. Similarly macros are expanded in yet another disjoint environment and this mediates against any tendancy to write macros which depend on the calling environment because the macro's calling environment is not there. More detailed descriptions of the Lisp/VM system can be found in [White, 1978] and [Alberga *et al.*, 1986].

Drawbacks to the design of Lisp/VM lie in its dependence on the IBM/370 architecture and the large body of the system that is written in assembler. The dependence on the 370 architecture is due mostly to the time at which the system was designed and prevailing views on address space limitations. That a large part of the system is written in assembler stems from the fact that the the system had to be written from scratch and that there was little or no experience of writing Lisps in high-level languages then. So, although Lisp/VM scores well in the areas of semantics and usability it loses out on portability and adaptability.

2.3. Cambridge Lisp

Cambridge Lisp was developed in the period just after Lisp/370 and during the second phase of Standard Lisp, but before Portable Standard Lisp. The authors of Cambridge Lisp were individually involved in Standard Lisp and Lisp/370 and these experiences are reflected in the structure of Cambridge Lisp. As with Standard Lisp, Cambridge Lisp was borne out of a need to support algebra research but rather than following the Standard Lisp report word for word, there was a strong influence from the heavier emphasis placed on semantics by Lisp/370.

Consequently, Cambridge Lisp is also single valued. However, the interpreter is not founded on a formal model and default scoping is dynamic, which must be seen as a retorgrade step after the richness of Lisp/370. On the other hand, the principle of providing very detailed information after an error was promulgated from Lisp/370, albeit in a less usable manner. Experience of writing several (Standard) Lisp systems led to the implementation of a large part of Cambridge Lisp in the high-level language BCPL (also used for Alto Lisp). This body of code provides a complete interpreter and a fair proportion of the system; the compiler and various other packages are written in Lisp and loaded into the core BCPL system.

Because of the choice of a particular high-level language, rather than Lisp itself as the implementation language for a Lisp system, Cambridge Lisp is rigidly tied to the calling and other conventions of BCPL, so although in the early stages this decision was extremely beneficial, in the longer term it is perhaps questionable. The use of BCPL has also helped to make it somewhat easier to port Cambridge Lisp to other machines, although the process is more painful than for Portable Standard Lisp because the first task is to port BCPL, and then to retarget the Lisp compiler. Cambridge Lisp was first implemented on an IBM/370 and several design decisions in Cambridge Lisp reflect this physical architecture, which can make for problems when moving the system to a machine which is not fundamentally IBM/370-like. Cambridge Lisp was not designed with portability in mind, but the (serendipitous) choice of a high-level language for the core system has made this possible at some cost. In many respects Cambridge Lisp is a compromise between the leanness of Standard Lisp/Portable Standard Lisp (in which semantics are ignored for the sake of efficiency) and the comprehensiveness of Lisp/VM (in which correctness has the upper hand, almost regardless of cost). More detailed explanations of Cambridge Lisp can be found in [Fitch & Norman, 1976] and [Fitch & Norman, 1977].

3. Semantics

The relationship between Lisp implementations and the mathematical foundations of Lisp in lambda-calculus has been somewhat loose, although lately, Lisp implementations can be seen to be converging with their logical roots. It is acknowledged that the first implementations of Lisp enshrined a misunderstanding of lambda-calculus, albeit a useful misunderstanding and one that may well have contributed to Lisp's popularity as a prototyping language and thus to its longevity. The principal issue under semantics is the meaning of names, which can be divided into two sub-issues: the scope of a binding and the environment used in resolving a reference. The preceding is an attempt to state in unemotive terms the question of lexical and dynamic scope and the question of function cells and value cells.

3.1. Scoping

The matter of scope and extent of the bindings of names is not just one of adherence to the true meaning of lambda calculus because it also has ramifications for the efficiency of the mechanisation of lambda calculus [Sussman & Steele, 1975]. Whilst lexical scoping permits dead-reckoning of the position of the allocation of the binding, dynamic scoping requires more complex machinery for maintenance of and access to bindings. Although the majority of Lisp systems adopt dynamic scoping by default in the interpreter, lexical scoping is preferred for compiled code because it is more efficient.

However, the semantics are not the same and code may behave differently after compilation, requiring further debugging, often in a more inimical environment than that provided by the interpreter.

This *discrepancy* exists in PSL and Cambridge Lisp but does not in Lisp/VM. As described earlier, the Lisp/VM interpreter is based on an SECD machine and compiled code is required to have an effect equivalent to the effect of the actions of the SECD machine given the same program (that is, compilation is a semantics preserving program transformation). Thus, if at any time, a compiled program is found to behave differently from its interpreted counterpart, then it the compiler is changed to correspond, rather the difference being declared a feature. There is a price to pay, in that sometimes some quite complex support mechanisms are required to make compiled code behave *properly* and those mechanisms engender costs in both time and space.

There is still room for deviant behaviours because Lisp permits macro operators, which are expanded when an expression is compiled. There is no way that such variance can be prevented without circumscribing the current nature of macros since the treatment of macros under compilation is forcing an early binding of something that depends on late binding. It is another issue as to whether using macros in such a way that they *do* depend on late binding is advisable.

3.2. Name Resolution

The second topic mentioned in the opening paragraph of this section has been the subject of much controversy. Early Lisps were ambivalent on the issue of whether the value of a name in operator position was different from the value of the same name in an operand position. That they should be the same is not explicitly stated in the theory of lambda calculus, but it is clear that (x x) means x[y:=x], or apply the value of x to itself. Some Lisps, (Scheme, Lisp/VM, Cambridge Lisp, T) have sought to maintain this consistency of resolution whilst other Lisps have preferred to maintain the ambiguity

(MacLisp, InterLisp, Franz Lisp, Common Lisp) on the grounds of efficiency and difficulties in macro expansion. The first objection is refuted both in the experience of the implementors of the, so called, single value Lisp systems and in recent experiments by Gabriel and Fahlman [Gabriel, 1987]. The second objection concerns the liklihood of inadvertent free variable capture in macro expansion and the roles that those names play in a computation. The argument is that since a macro can only capture free variables, it can only capture non-function bindings in a, so called, two value Lisp system, whilst there is a greater chance of capturing function bindings as well in a single value Lisp. This reasoning lacks rigour.

The interaction of macros with the free variables is still an unreasonably clouded issue. Clearly the naive substitution operation is open to confusion. The technique adopted in Lisp/VM is the most sophisticated the author has encountered, although experience has revealed deficiencies too [Alberga *et al*, 1986].

3.3. Closures

The early 1970's saw a preoccupation with an obscure problem called the *funarg* problem. The difficulty was how to pass functions as arguments and return them as results (a direct link to the lambda calculus) and maintain the *correct* evaluation environment for that function. It could be said that the problem was one of Lisp's own making since the management of the evaluation environment would not have been so difficult had it not been for dynamic scoping and the need to extend the calling environment on function application. The consequence was that for functional values to work correctly, the environment extant at the time of the creation of the functional value had to be captured and passed around with the functional value. Thus, when the object was used later the bindings of free variables of the function resolved to those existing at definition time. The stack model proposed by Bobrow and Wegbreit [1972] addressed all of these problems and was subsequently adopted in the implementation of InterLisp and extended for Lisp/VM.

In the middle of that decade, Scheme showed how effective lexical closures could be. In addition, lexical closures are much simpler objects to build since the closure of an expression is apparent statically and can be computed by inspection, whereas the dynamic closure, as described above, can only exist at the instant it is needed.

Neither Standard Lisp nor Cambridge Lisp has concerned itself directly with the the funarg problem, although both have suffered experiments to provide such a facility. Lisp/VM because of its use of the Bobrow/Wegbreit stack model has always provided a funarg mechanism, but curiously up until recently it has only had the facility for dynamic closures; now lexical closures are provided too [Mikelsons, 1987].

The funarg problem is fundamentally related to the structure used to store dynamic bindings. The classical storage model is the association list (so called, deep binding), which is good at context switching and bad at binding access. Its dual - a fact that only becomes apparent after looking at Baker's description [1978] - is the rerooting association list (so called, shallow binding), which is good at binding access and bad at context switching. Because Standard Lisp and Cambridge Lisp do not support funargs, there is no need to context switch and so both use a simplified form of shallow binding. Lisp/VM is providing context switching facilities and so it uses a rather complex variant of deep binding designed to lower the cost of variable access significantly.

4. Portability

A Lisp system has a more complex run-time system than many other languages and so porting it to a new operating environment is potentially a harder task than for many other languages. It seems that three phases can be identified in the writing of systems: assembler based, high-level language based and meta-circular based. The three systems under consideration offer an example from each of these phases, which is a direct consequence of the times at which they were written. Lisp/VM has a large proportion written in assembler, with the rest in Lisp, whilst Cambridge Lisp has a large proportion written in a high-level language with the rest in Lisp and Portable Standard Lisp has a very small number of functions written in C and everything else in Lisp. In fact, Standard Lisp offers examples of each situation (PDP-10 Standard Lisp was written in MACRO-10, Burroughs 1700 Standard Lisp was written in MBALM and Portable Standard Lisp in Lisp (as described above)).

Each phase above increasingly frees the system (but not the implementation) from a specific machine. Thus because Lisp/VM has a fair proportion of code in assembler, the task of changing it to take advantage of the XA addressing scheme on IBM/370 is not minor and retargetting Lisp/VM for a completely different architecture is almost unthinkable. Similarly, it is not straightforward to take Cambridge Lisp from one machine to another: first one must port BCPL and second one must port the Lisp and, in addition, because of its IBM ancestry, changing Cambridge Lisp not to use a high-byte tagging scheme would not be trivial. That said it is not a job for the inexperienced to retarget Portable Standard Lisp and modifying it to use low end tags does require some care. Nevertheless, PSL was built to be retargettable from the outset and the latest incarnation (version 3.4) is quite manageable, in that accumulated experience has identified and factored out almost every implementation specific feature.

5. Usability

The usability of a system must, to a certain extent, be subjective, although provision of particular features does allow for simple comparisons. A rough guide to the usability of each of the systems in this paper can be had from looking at where the priorities lay in the development of each system.

PSL was concerned with efficiency and portability; in consequence little effort is made to preserve information for debugging purposes and although the interpreter will check types, in many cases, compiled code will not. Thus, at best, one may be able to get a (partial) list of the pending function calls on the stack; at worst, one may simply get a protection violation or an invalid opcode and whatever ensues...

Cambridge Lisp takes much more care to leave a trail of information for the debugger to pick over, although it presents it simply as a stream of function calls and arguments. Naturally generating this information does impose some performance penalty. Cambridge Lisp is more careful about type checking in compiled code and coupled with the debugging information available does provide a rather more secure environment for programming than PSL.

Lisp/VM is in a different league. At a minor level, there is very comprehensive information available about the state of the computation when an error occurred. It is perhaps paradoxical that the information is more helpful when working on compiled code than when working on interpreted code! This is a slightly unfortunate feature of the use of an SECD machine model for the interpreter, since the stack only contains information about the state of the SECD machine and understanding the nature of the error requires some understanding of the rules in the SECD machine. The more obvious debugging support in Lisp/VM is provided by the HEVAL stepper/debugger [Mikelsons, 1985] within the LispEdit interactive editing environment. HEVAL takes advantage of the branching stack facility and multiple environments in Lisp/VM to run the debugging control on one branch with the program that is the subject of the debugging operation on another branch. HEVAL coupled with LispEdit and the sophisticated indexed file mechanism (version management, automatic recompilation) make for a much more supportive program development environment than either of PSL or Cambridge Lisp.

6. Conclusions

It is natural to consider that the latest systems embody the latest and best ideas. Although that can be the case, and certainly PSL offers some well-engineered ideas on portability and system organisation, some of the ideas that are only beginning to be accepted now had their genesis over 10 years ago. None of the systems presented here are ideal: Portable Standard Lisp and Lisp/VM in some senses represent opposite ends of a spectrum which puts portability and efficiency at one pole and semantics and usability at the other. Cambridge Lisp could be seen to represent a compromise position but by virtue of the choice to use BCPL for a large part of the implementation and the loss of Lisp/VM's lexical scoping several opportunities were lost.

All these systems have points to recommend them and the wider Lisp community would benefit from recognising the thought and care into the aspects of Lisp system design on which each system has focussed.

7. References

[Alberga *et al.*, 1986] Alberga, C.N., Bosman-Clark, C., Mikelsons, M., Van Deusen, M.S. & Padget, J.A., *Experience with an Uncommon LISP*, Proceedings of 1986 ACM Symposium on Lisp and Functional Programming, ACM, New York.

[Baker, 1978] Baker, H.G., *Shallow Binding in LISP 1.5*, CACM, July 1978, Vol. 21, No. 7, pp280-294.

[Blair, 1976] Blair, F.W., *The Definition of LISP 1.8+0.3i*, IBM Internal Report.

[Bobrow & Wegbreit, 1972] Bobrow, D.G. & Wegbreit, B., *A Stack Model and Implementation of Multiple Environments*, CACM, October 1973, Vol. 16, No. 10, pp591-603.

[Bradford *et al.*, 1986] Bradford, R.J., Hearn, A.C., Padget, J.A. & Schrufer, E., *Enlarging the Reduce Domain of Computation*, Proceedings of SYMSAC 86, ACM, New York.

[Fitch & Norman, 1976] Fitch, J.P. & Norman, A.C., *Implementing LISP in a High-level Language*, Software Practice and Experience 1977, Vol 7, pp713-725, published by Wiley and Sons.

[Fitch & Norman, 1977] Fitch, J.P & Norman, A.C. *A Note on Compacting Garbage Collection*, The Computer Journal.

[Gabriel, 1987] Gabriel R.P., private communication.

[Griss & Hearn, 1981] Griss, M.L., A.C. Hearn, *A Portable LISP Compiler*, Software Practice and Experience 11, 541-605, 1981.

[Griss *et al., 1982] Griss, M.L., Benson, E. & Maguire, G.Q., PSL: A Portable LISP System*, Proceedings of 1982 ACM Symposium on LISP and Functional Programming, ACM, New York, 1982.

[Hearn, 1969] Hearn, A.C., *Standard Lisp* SIGPLAN Notices, ACM, New York, 4, No. 9 (Sept. 69). Reprinted in SIGSAM Bulletin, 28-49, ACM, New York, 13, 1969.

[Kessler *et al.*, 1986] Kessler, R.R., Peterson, J., Carr, H., Duggan, G., Galway, W.F., Knell, J. & Krohnfeldt, J., *EPIC - A Retargettable, Highly Optimising Lisp Compiler*, Proceedings of SIGPLAN 86: Symposium on Compiler Construction, ACM, New York, 1986.

[Landin, 1967] Landin, P.J., *The Mechanical Evaluation of Expressions*, The Computer Journal, Vol 6, 1964.

[Marti *et al.*, 1979] Marti, J.B., Hearn, A.C., Griss, M.L. & Griss, C., *Standard Lisp Report*, ACM-SIGPLAN Notices, Vol 14, No 10, 1979.

[Mikelsons, 1978] Mikelsons M., private communication.

[Steele, 1977] Steele, G.L.S., *Macaroni is Better than Spaghetti*, ACM-SIGPLAN Notices, April 1977.

[Sussman & Steele, 1975] Sussman, G.J. & Steele, G.L.S, *SCHEME: an Interpreter for Extended Lambda Calculus*, Memo #349, Artificial Intelligence Laboratory, MIT, December 1975.

[PASS, 1987] Portable AI Systems Support report, *PSL3.4 Manual*, Computer Science Department, University of Utah, 1987.

[White, 1978] White, J.L., *Lisp/370: A Short Technical Description of the Implementation*, ACM-SIGSAM Bulletin, November 1978.