Abstract

It is usually agreed that programming languages for implementing (other) programming languages, or "implementation languages", should be simple low-level languages which are close to the machine code and to the operating system. In this paper I argue that a very high level implementation language is a good idea, of particular importance for Knowledge-based systems, and that LISP (as a language and as a system) is very well suited to be a very high level implementation language. I also discuss the significance of special-purpose programming languages, and the requirements that they have for a very high level implementation language.

1 The niche for a Very High Level Implementation Language

Programming languages become successful by establishing and dominating a particular language niche, such as number-crunching for FORTRAN or data-crunching for COBOL. Indeed it may be argued that new programming languages can only establish themselves if they fill a new niche, and that it is extremely difficult to displace an established language in an existing niche.

LISP is usually understood to occupy the niche of "a programming language for artificial intelligence", or possibly for formula manipulation. It was also a favorite implementation language for the first generation of commerical expert systems, but has gradually lost ground to other languages such as C. In this paper I will argue (1) the general significance of "Very High Level Implementation Languages", or VHLIL for short, from a software engineering perspective; (2) that the software technology of Knowledge-Based Systems depends on the availability of good VHLIL, and (3) that it is appropriate to view LISP as the foremost VHLIL language.

Since VHLIL is not generally recognized as a language type, the topic of the present paper does not focus only on Knowledge-Base Systems as such. I also want to argue the case for Very High Level Implementation Languages as such, and to argue that LISP is well suited to play a rôle there. The topic is large, and in order to keep the paper within bounds I will base it on references to earlier work in my own research community (i.e. my present and previous affiliations), where Lisp has mostly been used as a VHLIL. Since the references are to specific cases which exemplify the principles proposed here, I will not make any attempt to be fair by referring to examples from elsewhere. The reader can hopefully add similar examples from his or her own experience.

The utility of VHLIL is derived from the utility of *special purpose programming languages*. By a special purpose programming language (SPPL) I mean a programming language which is designed for being able to characterize a limited application area, or a specialized function in software. The development and use of SPPL has been a standard methodology in A.I. research since many years: ATN parser systems, frame data bases, rule-based systems such as EMYCIN, and reason maintenance systems are only some of the examples that come to mind. This tradition was continued for application domains when Expert systems and Knowledge-based systems evolved.

However there are many other examples of SPPL:s in the practical world besides in Knowledge-based systems. Many "fourth-generation" software tools, such as spreadsheet systems and database query languages, implement and support specialized programming languages. Forms management and other dialogue management systems offer another example of SPPL:s for a specific function; languages for programming real-time and process control systems represent an example of SPPL:s for a specific application area.

Researchers in programming languages and software engineering tend to treat special purpose languages condescendingly, however. One reason is that from the point of view of professional language design, the embedded languages in for example spreadsheet systems offer little of interest. The languages in existing products often fail to satisfy (conventional) criteria for "good language design", and the task of designing such a language appears to be so simple that it does not offer a challenge for the professional language designer.

Maybe one would expect research in software engineering to have assimilated the idea of special purpose languages, in view of their considerable practical importance. This does not seem to have happened, and the reason is possibly that in its attempt to establish software development as a branch of engineering, they have identified factors which are important in other kinds of engineering (particularly in electrical engineering), but missed factors which are indigenuous to software, such as the possibility to build a system which can analyze and compile its own programs; an approach that has no counterpart in hardware.

One of the very few trend-setting texts which emphasize the use of special-purpose languages is the book by G. Sussman and H. Abelson, *Structure and Interpretation of Computer Programs*, and it is one of the important messages in that book.

To summarize, we have made the following observations:

- by tradition in A.I. and KBS, LISP is often used as a VHLIL for implementing special-purpose programming languages

- SPPL are important in practical software engineering
- SPPL have not been "discovered" by the research communities in programming languages and software engineering
- consequently, little research has been explicitly directed towards the development of actual VHLIL or to design principles for such languages.

As integration between knowledge-based systems and other aspects of computing increases, there is a danger that the special character and special strength of KBS is lost under the strong constraints imposed by conventional operating systems, conventional data base systems, and conventional programming languages. If we can identify and argue for the special significance of SPPL and VHLIL, not only from the KBS point of view but from a general software engineering point of view, then there will be a better chance for the special character of KBS to survive the integration process.

2 Criteria for good special-purpose languages and their implementations

Before proceeding to the rôle of LISP in the context of VHLIL, let us first discuss the characteristics of good SPPL:s and their implementation languages.

Probably the most important feature in an SPPL is *conciseness*. The thesis work of Tore Risch[**?**, **?**] illustrates this point. The task that Risch undertook was to write an optimizing translator from a fairly conventional query language, to a corresponding, efficient COBOL program. Optimization in this context means, for queries that refer to several relations or files, to select the order in which the different relations are accessed, based on information about the "fan-in" and "fan-out" of each field in each relation. Risch compared the output of his compiler with the COBOL programs that were manually written by the commercial programmers, and found a remarkably high agreement. The query language was however considerably more concise: a query of a few lines expanded into almost as many pages of COBOL code.

A popular explanation for this factor of size is that it confirms what you already knew about the verbosity of COBOL. However the difference is so large that there must also be some other explanation. In my opinion one should understand it in terms of expressiveness: the ideal niche for a special-purpose programming languages is an area which is *semantically narrow*, meaning that only a limited set of programs and functionalities belong to it, and at the same time it is *commercially broad*, meaning that there is a large demand for programs that fall within the niche. A special purpose programming language for a semantically narrow niche only needs to express enough information for identifying one member from the rest of the niche. It can therefore be much more concise than a language which identifies the intended program in the class of all possible computer programs, as a general purpose programming language must do.

Interfacability is another, very obvious requirement on an SPPL. It follows from the definition: since a special-purpose language can not satisfy all needs, it must usually be combined with other special-purpose tools that solve the other parts of the whole given problem. Also since requirements and technologies change, an SPPL must be designed and implemented with an *open system* philosophy, so that it can easily be interfaced to new products that appear in the future.

Interfacability has implications for how the SPPL is implemented, but it also has consequences for the language design: a *clear semantics* and *clear language definition* are important for making sure that interfaces to other systems can be realized and maintained. This adds to all the other, well known reasons for providing precise and clear language definitions.

Many SPPL:s are in principle *programmable editors*, in a broad sense of the word "editor". In other words they are primarily defined as a "system" i.e. a program which enables the user to operate on a particular kind of data structure, such as the text buffer in a text editor, or the spreadsheet matrix in a spreadsheet system. The editor is made programmable so that sequences of user commands can be turned into "subroutines", and so that certain operations (such as updating the screen) are performed automatically but in a way customized by the user. In such cases the language aspect of the SPPL is secondary, and one should really see a *generic software system*, henceforth written as GSS, where the "language" is a secondary design which is only needed to support the customization facility.

Several examples of generic software systems come to mind easily: not only text editors such as EMACS, and spreadsheet systems such as VISICALC, but also forms management systems, hypertext systems, and more specialized "editors" such as electronic mail systems and electronic calendars. Noninteractive text formatting systems, such as LATEX, are also generic software systems although not editors.

In all these generic editors the primary data structure that the editor operates on, is fairly simple. In addition there is a whole range of "editors" which perform complex and high-level operations on a high-level and application-oriented data structure, for example knowledge acquisition systems and computer-aided design systems. Many such systems can also be configured using an auxiliary programming language.

When an SPPL is really the language aspect of a generic software system, then the semantics definition should start with a definition of system's primary data structure in high level terms. The second step is to define the elementary operation of the software system, including the interactive command language used by the user in case of an editor-type system. Only then is it meaningful to define the syntax and the semantics of the SPPL as a language for defining more complex operations on the primary data structure. In some earlier papers[**?**, **?**] I have proposed methods for defining and compiling the formal semantics of editor-type generic software systems.

3 Programming environments for special purpose programming languages

In early history, the programming language was the primary design issue. The task in systems building was to implement an efficient compiler for a given programming language and a given target computer; a task which was also amenable to theoretical analysis for those so inclined, but which did not seem to require any new concepts. The alternative approach of emphasizing the *programming environment* developed primarily in the LISP world[**?**, **?**], and started from the idea to organize a number of tools around a common, computer-oriented representation of the program as a data structure. Translating a piece of program to machine language was just one of the functions of the programming environment. During the 1980's the concept of programming environments has been adopted by other programming language cultures, and has been generalized to other kinds of environments, for example for software development.

The common thread of this development has been to downplay the significance of the programming language, and to emphasize the importance of the system for administrating pieces of programs and performing various functions on them. These principles are naturally applicable to special purpose programming languages as well as to general purpose ones, GPPL:s. They also combine nicely with the observation that was made above, that some SPPL:s are really just the customization language of a generic software system. Since such a system in itself is typically a collection of tools for operating on the primary data structure, the "programming environment" for the SPPL is then merely defined as being those of the tools that support the customization language.

In fact, for several reasons it is more convenient to develop a programming environment for an SPPL than for a GPPL. The limited domain and limited expressiveness of the SPPL is one obvious reason. If the SPPL is well designed and has a clear semantics, is not very complex, and only addresses a limited range of functionalities, then one should be able to support it more effectively than a general purpose language. This goes both for the compilation service, where the characteristics of the SPPL's niche can be utilized, as in Risch's system, and for the user dialogue which could be set up in a domain-specific and therefore user-oriented fashion.

The limiting factor is of course *cost*. The development costs for programming environments of modern general-purpose programming languages, such as ADA, appear to be astronomous. By its very nature, a special purpose language and its user community can not carry such expenditures.

There is an obvious and fascinating solution to this dilemma: let us drop the traditional notion that an implementation language (for implementing programming languages and their programming environments) should be as close to machine language as possible, presumably for performance reasons, and consider instead the possibility of *very high level implementation languages* which are designed to be suitable for implementing SPPL:s and generic software systems. The next section discusses the requirements on such a VHLIL.

4 Requirements on Very High Level Implementation Languages

The discussion of SPPL (special purpose programming languages) and GSS (generic software systems) above has a number of specific implications for the design of a VHLIL. First, clearly the VHLIL should be able to host several SPPL:s in a common environment, in order to provide as much support as possible for the bridgings between multiple SPPL:s. Therefore in particular the VHLIL must itself be a general purpose programming language.

Furthermore since many SPPL:s are attached to generic software systems, the VHLIL must be suitable for implementing such GSS. In particular this means that the VHLIL must have full access to dialogue screens and other user dialogue interfaces, file systems, data bases, computer network and other communication interfaces, and other systems programming services.

Almost all programming languages are designed so that programs are complex formulas, and this goes for SPPL:s as well. Language facilities for formula manipulation, such as the availability of recursive data structures (list structures) and recursive program execution, are therefore elementary needs. This is particularly important as the programming environment proceeds from only offering a compilation or interpretation service, to providing a host of other operations on programs.

A very important factor in any programming environment is the possibility to move easily between program execution and program editing. It is unthinkable that the user of a programming environment for one or more SPPL should have to leave the environment in order to test-run his or her programs, and the VHLIL must therefore be able to interpret SPPL programs. The natural way to arrange this is to make the VHLIL itself interpretive, and to arrange that SPPL programs can be compiled to VHLIL programs (possibly in a subset of the VHLIL), and that the VHLIL programs can in their turn be compiled to a machine language level.

In LISP this property is obtained automatically as a consequence of the non-distinction between programs and data. It is so obvious in the LISP community that I wouldn't have to mention it here, except in order to point out how VHLIL naturally require a facility which is characteristic of LISP but unavailable in most other programming cultures.

In addition to these basic needs for the VHLIL, there is the obvious need for facilities which increase the reusability of the various parts of the SPPL programming environment, and thereby reduce the software costs. Standardized facilities for documentation (for example a hypercard-type facility), for user interaction (such as a standardized window system) and so on are important in that respect.

Another powerful tool for programming environments is *partial evaluators*. The first time I thought of SPPL as an independent software technique, was when we were using ATN grammars for syntax analysis, and predicate logic for deduction in a natural language project. We had written interpreters for both the ATN language and for Horn clause logic, and wanted to compile both to LISP code. The idea we came up with was to write a universal tool which takes an interpreter, and a program which it can interpret, and which constructs from them a corresponding "compiled" program in the language in which the interpreter is written. That universal tool, the partial evaluator or partial interpreter, was first used for compiling a subset of predicate logic[**?**, **?**, **?**], and was extended by Haraldsson et al. to other uses[**?**, **?**, **?**] and by Emanuelsson to partial evaluation of an interpreter for pattern matching[**?**]. The next step was taken by Komorowski[**?**, **?**] who transferred the partial interpretation idea from LISP to PROLOG, and generalized it to the notion of partial deduction[**?**].

Partial evaluation is a powerful technique, but its successful usage depends on a number of preconditions: it belongs naturally in an interpretive, symbol manipulating environment, and it can only work if the SPPL at hand has been defined by an interpreter which is reasonably clean, simple, and well organized. What was said above about the need for precise and simple syntax and semantics for SPPL, is therefore also a precondition for the use of partial evaluation techniques.

5 Systems Development Environments

Since many special purpose languages are best seen as the language aspect of a generic software system, it is natural to ask whether that holds for *general* purpose languages as well. Would it make sense to view the *systems development environment* as the primary technical device, and the programming language as the secondary instrument for tailoring the environment to particular applications or application classes?

From the viewpoint of conventional programming languages this would be a profound paradigm shift, and it is well known how long it takes for those to happen. From the viewpoint of the LISP culture the answer to the question is "yes, of course", since already in the late 1960's LISP changed from a programming language to a systems development environment, with list structures (including programs) as the primary data structure. By systems development environment I then mean a fully general GSS, which is more or less universally applicable.

In the context of the present paper we must ask what would be the best implementation technology for SPPL:s, and for the GSS:s that often carry them. Again the answer to the question must be "yes, of course"; a software development environment would be very suitable for implementing SPPL:s, and for two reasons. On one hand, many of the functions which are needed in a programming environment for an SPPL require manipulation of the object program, for various kinds of program analysis and program transformation, including partial evaluation. A generic software system which is particularly capable of handling programs is then an adequate tool.

Also, when it comes to implementing generic software systems, notice that the very character of a GSS is that it starts as a relatively general tool, which is specialized to each particular application area by adding more information to it. It would therefore be a very natural strategy to begin with a super-generic software tool, i.e. the software development environment, which can at first be specialized in various ways to create the various GSS, which are then in turn further specialized for each particular usage. The various techniques which were mentioned in the previous section, including partial evaluation, tie in immediately into that notion. The same goes for the standardized facilities for documentation, user interaction, etc: they can now be viewed as organic parts of a software development environment, continuously available as the environment is configured in various directions.

Finally the notion of a very configurable software development environment can contribute to interfacability between multiple SPPL:s which may need to be combined in the same application. If all the SPPL:s are based on the same software development environment then interaction between them would at least be facilitated.

To summarize, the duality between special purpose programming languages and generic software system, is matched by a duality between VHLIL:s and software development environments. It is common to refer to the language although the corresponding system is actually more significant. We continue the tradition, so that the term VHLIL in the following sections actually refers both to the language and the system.

In an earlier paper[?] I have described and discussed our experience with using LISP as a systems development environment for office information systems.

6 The role of LISP as a Very High Level Implementation Language

The requirements on VHLIL which have been described above, are almost exactly an abstract description of LISP, but they have been independently derived from the basic characteristics and basic needs of SPPL and GSS. In other words, if LISP did not already exist, it would have to be invented since no other[1] existing language is adequate as a VHLIL, and VHLIL are a useful and powerful concept.

There are already a number of examples of how LISP is being used as a VHLIL and as an embedded language in widespread software products: expert systems tools such as EPITOOL and KEE, text editors such as GNU EMACS, computer-aided design systems such as AUTOCAD, data base systems such as MIMER, and so forth. However the total "market" is several magnitudes larger than the present usage, and it is therefore important to discuss what have been and what are the factors that limit its more widespread usage.

Four factors stand out as the most significant ones: buyer and user acceptance, performance, interfacability, and standardization. The standardization, which is needed for very obvious reasons, has obtained a solution through the establishment of the COMMONLISP standard. Interfacability is needed because parts of a generic software system, or otherwise of the runtime system for an SPPL, needs to be written in other languages than LISP, such as C for example. The arrival of LISP systems which interface well with

[1] except of course those that are very similar with LISP or contain an embedded LISP system within them

code modules written in other languages is therefore very welcome although long overdue.

But again it is important to remember that systems, not languages are the primary issue. It is not enough to interface to other programming languages; there must also be interfaces to standard text editors and formatters, standard data base systems, and standard user interface management systems. Also in designing such an interface, the rapport between the data structures in the respective systems is at least as big a problem as the compatibility of the procedure invocation or function-call mechanisms.

We all know that LISP has a bad reputation with respect to performance, and that the reputation is too pessimistic and not entirely deserved. Although much has been done to improve the situation, more can and should be done. In many cases it seems natural to only use LISP for higher-level parts of the system that is being developed, and to use other languages for "inner loops" or performance sensitive parts of the system. In such cases, improved interfacability is essential for dealing with the performance issue. In other cases the special properties of LISP are uniformly valuable throughout the application, and then LISP co-processors or even Lisp machines are the natural way to go.

User and buyer acceptance, finally, is in a certain sense *the* crucial hurdle, but it also depends on the other issues: if good answers can be found to the problems of performance, interfacability, and standardization, then a good software product which happens to be wholly or partially implemented in LISP *should* not have any user acceptance problems.

But still it might: in order to gain acceptance a product must have not only technical strength, but also credibility. It is important for the market acceptance of LISP that we can *explain its raison d'être* in clear and convincing terms. The general views of LISP as a language for list processing, for A.I., or for formula manipulation are then simply not strong enough: not enough people identify with these niches (when was the last time you had a list processing application?), and also these views fail to explain the philosophical differences between LISP and conventional third generation languages. Why shouldn't a programming language for A.I. or Knowledge-based systems enjoy the blessings of strong data types, block structure, and syntactic sugar?

My proposal here is that it would be better to advance LISP as a Very High Level Implementation Language, since the need for SPPL, GSS, and VHLIL are readily received by practitioners, and since the distinguishing characteristics of LISP make perfect sense in that context.

This proposal also bears on the present trend to re-implement expert systems shells from LISP to a conventional language such as C. The reasons for the change are quoted in terms of performance (in time and in memory space), interfacability, and user acceptance. It seems to me that the benefits of such reimplementations are often overstressed, and the disadvantages are underestimated when this trend is described in the computing press. But again, the unique contributions that LISP can provide for an advanced expert systems shell are best formulated if the shell is seen as a generic software system which needs an auxiliary special-purpose language, and for whose implementation one needs a good VHLIL.

To summarize the situation then, there has been significant progress in some of the areas which are relevant for increasing the use of LISP in practical computing, but a lot remains to be done. None of the problems appears unsurmountable in principle. The software developers which bet on LISP as a viable implementation instrument, have in my opinion a great opportunity for the future, but need to put additional efforts into LISP's infrastructure. Also we must all contribute to the analysis of, and the public debate about the deeper characteristics of LISP visavis other programming languages.

7 LISP in the Systems Software Borderland

My argument up to now has dealt with LISP as a Very High Level Implementation Language for Special Purpose Programming Languages in general, subsuming Knowledge-based systems. Let me conclude with a few words about the possible rôle of LISP in the forthcoming restructuring of the Systems Software Borderland. This term was introduced in a recent paper[**?**] along with an argument about the sharp distinction that contemporary software engineering makes between the operating system on one hand, and the programming language(s) and its run-time system on the other.

In this context I view the operating system as that software system which is in charge of the primary dialogue with the user, and which is able to invoke other software systems supporting general-purpose or special-purpose languages. By this definition I de-emphasize other standard definitions of an operating system, such as the administrator of shared resources.

The rôles for the operating system and for the programming system are well defined by the conventional wisdom. It defines, for example, the character of the data structures that the operating system maintains and the programming system uses, namely the usual structure of directories containing files which are characterized by (mnemonic) character strings.

Two things are interesting about this interface between the operating system and the programming systems: it is tailored to the realities of procedural or "third generation" programming languages, and it is very stable. The basic design choices are the same in operating system that are now on the rise, such as UNIX or (?) OS-2, as they were in OS-360, TOPS-10, and even in their now forgotten predecessors. The reason is very simple: every new programming language and programming system must be able to thrive under existing operating systems, and every new operating system must accomodate existing programming systems.

Unfortunately the character of this established interface is now completely inadequate, since the requirements on the operating system must be seen as the requirements made by the common generic software systems, and not as the requirements made by the common programming languages. To mention just one example, consider how the realities of programming would change if the operating system would offer a hypertext or "hypercard" data base instead of the conventional file directory, to be used as a common resource for all generic software systems. Such a change would provide strong additional support, both for the organization of application data and for the organization of the program and its documentation.

Thus the borderline area between the realm of the operating system (OS) and the programming system (PS) is today a border zone that is very difficult to transgress. In principle one could obtain a much more interesting, and I believe a much better software technology if it were possible to draw the line differently. In fact one would then also realign the border between data base systems on one hand, and the OS and PS on the other. The term *systems software borderland* was proposed in [**?**] for this challenging problem area.

For the design of advanced Knowledge-based systems this borderline is a significant difficulty. For traditional "crunching" types of computing the problem is marginal, but the non-uniform data and the close integration of programs and data that is characteristic of a knowledge base, does not fit comfortably with the traditional borderline.

The LISP programming technology is likewise hurt by the current stalemate in the systems software borderland, since the services offered by LISP do not fit on only one side of the curtain. This I believe was the most significant reason for the development and attraction of special LISP machines, and a much more important one than the hardware issue. The LISP machines of course dealt with the problem by going for LISP a hundred per cent: when the computer uses LISP as the only language, even on the user entry level that is usually considered to be the turf of the operating system, then the operating system is assimilated into the programming system, and the misplaced divider has been removed.

Unfortunately the result of the commercial experiment with LISP machines was negative. One may speculate whether this outcome was necessary, and whether for example a LISP based operating system on conventional hardware would have been more successful. Certainly in the short run now the battle is over, and it is hard to see any alternative to UNIX as a host for LISP systems.

In a somewhat longer perspective, it is however important to remember that the intrinsic anomalies in the systems software borderland have not been resolved, only suppressed. The deadlock that characterizes the situation only means that the revolution will be so much more violent when the present structure collapses, as it eventually must, and a new structure establishes itself. It is my very strong belief that the future good design for basic systems software will be a generic, but highly general software system which supports a reportoire of several languages, ranging from general purpose to special purpose and from high level to low level. Such a computing environment will be able to accomodate knowledge processing as well as routine data processing. When it is designed, many of the typical characteristics of LISP will be retained – or reinvented.