L*: An Interactive, Symbolic Implementation System

A. Newell, D. McCracken, and G. Robertson

Department of Computer Science
Carnegie-Mellon University

October 3, 1977

L* is a system for building software systems. The design of L* stresses features not found in many other implementation languages or systems: it is interactive rather than compiler based, it has a symbol manipulation language embedded in it, it is highly extensible, and it allows for total accessibility to both the underlying hardware and the L* system itself. This paper describes the design philosophy of L*, the mechanisms used to implement it, the experience we have had using it, and some unresolved issues it still presents.

This work was supported by the Defense Advanced Research Projects Agency under contract no. F44620-73-C-0074 and monitored by the Air Force Office of Scientific Research.

1. Introduction

L* is a system for building software systems. It is a tool for the professional programmer, and was originally intended for use in constructing artificial intelligence systems. Its most important use, however, has been in providing the basic software support for experimental computer systems. Under development at CMU since 1969, operational versions of L* have existed since 1970 and have been in experimental use by a small community. However, only a short description of the system and the design philosophy that underlies it has been published (Newell, Freeman, McCracken, and Robertson 1971). The system, its philosophy, and our experience with it have now reached a sufficiently mature state so that a general exposition of it seems useful.

L*'s roots lie in the series of IPLs, the original list processing languages (Newell and Shaw 1957; Newell, Tonge, Feigenbaum, Green, and Mealy 1964). As experience mounted with IPL-V and with LISP about the nature of system building in artificial intelligence, it seemed appropriate to make a fresh start with the emphasis on system implementation rather than on the language aspects. L6 (Knowlton 1966) had shown that efficient low level systems could be built using list structures as a data type. An early attempt to understand the lessons of L6 resulted in a similar macro-based system on the IBM 360 called *1 (Newell, Earley, and Haney 1967). An attempt to understand the nature of a flexible dynamic user interface resulted in a system call BIP (Basic Interface Package, Newell and Freeman 1968), embedded within IPL-V. All these systems can be taken as the direct precursors to L*.

Familiarity with two basic, albeit informal, notions is assumed throughout this paper.

> (1) A <u>software system</u> is an integrated collection of programs and data which provides the diverse functions necessary to an operating environment: communication with users, resource management, debugging aids, behavior monitoring aids, archiving, communication with other software systems, as well as the main problem solving functions for the task (though no particular subset seems to be essential). Typical examples are operating systems, large AI programs, and airline reservation systems.
>
> (2) An <u>implementation system</u> is a specialized set of software tools used to create software systems. Typical examples are BCPL (Richards 1969), BLISS (Wulf, Russel, and Habermann 1971), ECL (Wegbreit 1971), and XPOP (Halpern 1964). The concept of an implementation system has arisen more or less concurrently with an awareness of the severity of software production problems and with the discipline of software engineering devoted to coping with these problems.

To understand the relationship between L* and other implementation systems, it is useful to look at the basic forms that implementation systems have taken. Roughly speaking there have been three paths of development:

> Macrosystems: The assembly language has always been the court of last resort for creating any programming structure. The addition of macro facilities has been the main vehicle for adding facility to assemblers. This has led to the development of systems that take the macro facility as the central device in an implementation system. XPOP is a good example.
>
> High level language systems: The desire to use high level languages for system implementation has existed for a long time. These languages make system implementation and maintenance easier by making system structure more apparent. Until recently, their use has been limited because of the relative inefficiency of code produced by their compilers. This has changed, with several languages being used effectively, notably BLISS11 (Wulf, Johnsson, Weinstock, Hobbs, and Geshchke 1975) and BCPL.
>
> List processing systems: List processing systems offer interactive and symbol manipulation capabilities not generally found in either macrosystems or traditional high level languages. Their use has been primarily in building large artificial intelligence systems, which have many of the same properties as general software systems although they tend to be experimental in nature. The most commonly used list processing language is LISP (McCarthy, Abrahams, Edwards, Hart, and Levin 1962).

Implementation systems always involve a language of some sort and this often serves as a shorthand for denoting the system. But an implementation system is always much more; it is the total set of facilities that are provided to create software systems. These include all the usual functions that typically show up in a software system itself (e.g., an operating environment and debugging facilities).

We will assume general familiarity with these notions. Further elaboration can be found in Freeman (1975).

In Section 2 we will present the design philosophy of L*, to make clear where it stands in the space of implementation systems. In Section 3 we will describe the structure of the system that makes possible the realization of these design principles. Section 4 discusses the results of some experience with L*. Section 5 concludes with a general discussion of the open issues and some comparisons with other specific implementation systems.

2. Design Philosophy

In this section, we will describe the design characteristics of L*, and compare them with design characteristics of other implementation systems. The design philosophy of L* can be described in terms of a number of dimensions (or issues) on which implementation systems must take a stand. For these issue-dimensions there rarely exists a complete characterization of the alternatives, but only a few points that have been adopted by various systems. To help place L*, we will also indicate the position along these dimensions of typical high level language (HLL) systems and of LISP. Detailed comparison of L* with specific systems (e.g., BLISS or LISP) will be postponed to the end of the paper, since our purpose in this section is to describe L*.

Figure 1 lists the issue-dimensions vertically, with a column for each of the three systems. The entries are explained in the subsection on each particular dimension.

	HLL	LISP	L*
Symbols	Identifiers	Symbols	Symbols
Interaction	Batch	Interactive	Interactive
Production Mode	External	Internal	Internal
Flexibility			
Data-types	Some	None	Total
Control	Little	None	Total
Syntax	Little	None	Total
Contraction	None	None	Little
Accessibility	Partial	Partial	Total
Addresses	No	Yes	Yes
Efficiency	Uniform	Selective	Selective
Language Form			
Algebraic	Yes	No	No
Operand syntax	Infix,function	Function	Postfix
Variables	Yes	Yes	No
Integration	No	Complete	Complete
User Community	Universal	Dialect	Personal
Maintainability	Centralized	Dispersed	Local(self)
Design Strategy	Structured	Iterative	Iterative

Figure 1. Design Philosophy Characteristics

Symbols. What capabilities exist for symbolizing and representing within the implementation system entities in the object system, and for manipulating these representations? L* abides by the following principle:

> *Universal symbol system* : There should exist a single homogeneous system of symbols which can be used to represent any aspect of the object system.

This requirement is a stringent one that, in the current art, almost forces an implementation system to contain a symbol manipulation language. Thus, LISP also satisfies this principle. But BLISS does not. Instead, it has limited (though powerful) symbolic capabilities fixed in advance by the design of the system (e.g., for instructions, for blocks of memory space, for addresses and for integers). The requirement for a more general symbolic capability has been recognized, for instance, in Alphard (Wulf 1974), a language scheme under development, which has as one goal the ability to represent arbitrary levels of abstraction.

Interaction. What type of interaction occurs between the object system designer and the evolving object system? Alternatively, what is the interaction rate between designer and operational object system? L* abides by the following principle:

> *Full Interaction* : The designer should operate interactively with the implementation system in all aspects of object system creation.

Most implementation systems, either high level or macro systems, are compilers. The mode of operation with them is essentially batch: code, compile, run, debug and repeat. The loop is quite long with substantial coding often taking place between compiling steps. Some other languages, especially LISP, operate as full interactive languages where the loop is very short, and many incremental changes are closely interwoven with small running steps.

Production Mode. How should the implementation system construct the object system? Should the object system be grown from within the implementation system or deposited (as an object file is deposited by a compiler)? Should the object system be produced as one entity or as a series of modules? What kind of run time support is required if the object system is deposited? L* abides by the following principle:

> *Growth of object systems* : The object system should be created within the implementation system by adapting and adding to the existing facilities.

There are two basic possibilities: deposit the object system, or grow it. It has been practically taken for granted that the implementation system -- the assembler, compiler or macrosystem -- should produce the object system as a body of code independent of itself (i.e., as a module). Thus, HLL systems all deposit their object systems. Only with fully interactive systems has the alternative of growing the object system emerged. Thus LISP, as well as L*, grows its systems. The tradeoff is clear.

On the one hand, the object system may bear little relation to the implementation system; there is simply no reason why it should be mixed. On the other hand, large numbers of mechanisms can be imported from the implementation system and adapted; furthermore, a functioning system can exist at all times. With the choice of growing the object system, the problem of excess mechanism is a real one in many operating environments, so that contraction of the system is an important function.

Flexibility (Extensibility). What aspects of the system should be flexible (i.e., should be capable of extension or contraction)? The issue applies both to the implementation system itself and to the object systems that are to be constructed (though different positions may be taken for each). L* abides by the following principle:

> *Total flexibility* : All aspects of a software system should be subject to modification and extension.

The principle may be stated in another way:

> *No designer's prerogative* : The system designer should avoid design choices that cannot be later modified by a user of the system.

Perhaps the issue should have been called extensibility, since much of the relevant work has occurred under that label. But, following a suggestion of Krutar (1976), we prefer the term flexibility since we need to be as concerned with contraction and modification as with extension. Thus, when we refer to flexing a particular aspect of a system, we refer to the act of making it different in response to some demand.

Programming languages are in essence devices for flexing a computer. In so doing they set up a number of conventions which force the object system to be structured or to be specified in fixed ways. Attempts to relieve these rigidities can be classified under a number of headings: syntax, data-types, control, and name conventions. In Figure 1 we separated out these categories, since the progress in obtaining flexibility has moved through them. (Some categories, such as flexibility of the lexicon and flexibility of the procedures, are common to all programming systems and need not be listed.) We have added contraction as an additional category, simply as a reminder that almost no systems permit easy contraction, as opposed to extension.

L* takes it as central that flexibility should be present in every aspect of a system. This applies equally to both implementation and object system since in L* they are one and the same.

Accessibility. What parts of the programming system and the underlying machine are accessible from within the system for purposes of modification and exploitation? L* abides by the following principle:

> *Total accessibility* : All aspects of the system and the machine should be available for manipulation.

Despite the apparent desirability of such notions as total accessibility, there are alternative views that have equal plausibility. The main one is that a software system (especially an operating system) takes a machine and converts it to another more suitable machine, which is then what the user sees. The user is not supposed to have access to the underlying machine. This view, which is essentially that adopted by systems such as LISP, as well as by most operating systems, produces a sharp distinction between users of a system and system designer-maintainers. Only the latter deal with the guts of a system, which is a different world in terms of its conventions and flexibilities from that which the user experiences.

This alternative view produces two distinct issues of accessibility: (1) total accessibility of the underlying machine and (2) accessibility within the shell provided by the underlying system. Thus, LISP has complete accessibility within the system, but not to the basic machine. L*, on the other hand, attempts to provide complete accessibility to the underlying machine, while at the same time providing a world of the same interactive convenience as LISP.

Efficiency. What is the efficiency of the system with respect to the various vital resources: processor cycles, memory space, and i/o channels? Efficiency issues arise independently for the object system and for the implementation system, and with respect to the latter for initial construction and for modification. L* abides by the following principle:

> *Selective optimization* : Efficiency is to be achieved by the detection of the critical constraints in a running version and their selective removal.

The programming languages initially used in L* are interpretive, thus trading time efficiency for flexibility. This contrasts with compiler-based implementation systems, which endeavor to produce relatively efficient code. In the limit, as with BLISS11, very efficient code is produced at once for an entire object system. Since efficient systems are ultimately required, L* has to obtain the efficiency somehow. It attempts to do this by selective compilation, reorganization, data compaction, and microcoding.

Language Form. What linguistic forms does the designer use to communicate with the implementation system? This pertains only to the implementation system; presumably it is possible to construct object systems with any desired linguistic characteristics. L* abides by the following principle:

> *Dynamic interface* : The linguistic interface with the user should be dynamically adaptable.

There are two normal forms for implementation languages. The first is that of a higher level language, namely, an Algol-like language with expressions, procedures, functions, and infix operators. The second grows out of the form of assembly language; namely, a sequence of operator-argument forms that retains some sequential correspondence to the internal memory space (this carries over to most of the macro languages). LISP represents yet another path, retaining the expression form combined with a uniform prefix notation. As we will see, the form of L*'s language (which is postfix with no syntactic structure at all) grows out of its interpretation principle, which is determined on independent grounds.

Integration. How is the total set of facilities used by the designer organized? L* abides by the following principle:

> *Complete integration* : All of the facilities to be used in constructing a system are to be available as subparts of a single uniform world.

The situation normally faced by the designer of a system is that he has a set of distinct facilities -- such as languages, editors, debuggers, timing packages, and cross-reference programs. Each of these is created with a separate set of linguistic and interactive conventions. There is integration only at the level of the underlying operating system, seen by the user-designer as a single uniform command language. On the other hand, general interactive systems from JOSS (Shaw 1965) onward have adopted the other position by integrating all the facilities with the main language system. LISP, APL and L* belong in this latter category, along with a less well-known system called LCC (Perlis, Mitchell, and VanZoeren 1968). Though at heart a JOSS-like system, LCC has many of the features of a standard HLL, and thus represents a rather unique marriage of HLL constructs with the philosophy of complete integration.

User Community. What is the community over which the object system is to be fixed? L* abides by the following principle:

> *Personalization* : The system is to be adapted to the particular circumstances of machine, system builder, and task.

This issue poses a genuine tradeoff. Fixing the properties of a programming system (either an implementation or an object system) increases the communicability and portability of whatever programs are created within it. On the other hand, features of the particular computer or preferences and insights of particular designers and users cannot be fully exploited. The need for complete adaptation is greater when the available resources must be exploited to the limit, less when tasks do not press the art or the resources. Adaptation, and thus idiosyncracy, is more acceptable when the systems or tasks are unique in other ways, so that there is little to be gained from standardization in any event.

Maintainability. How is a system to be maintained, meaning both the removal of bugs and the gradual evolution towards increased capability? This applies both to the implementation system and to the object systems produced. L* abides by the following principle:

> *Local maintenance*: Maintenance should be totally within the purview of the users of the system (and hence their responsibility).

Design Strategy. How shall the designer proceed in creating a new object system? L* abides by the following principle:

> *Iterative design*: A system should be created by a series of successive approximations in the form of operational systems.

This issue of design strategy is usually referred to as "design philosophy", but we need to distinguish it from the design philosophy of L*, which we are in the midst of describing. Iterative design is to be contrasted with a top-down, structured programming design philosophy, in which a high premium is put on producing careful specifications, and even in proving that the algorithm being programmed is correct. It must be noted that none of the systems in Figure 1 actually dictate a design philosophy along the dimension of concern here. They only predispose towards one or the other. Implementation systems such as BLISS are consonant with the structured approach.

There is a subsidiary principle that we use with L* as a guide in designing object systems:

> *No excess generality*: No parameterizations should be created without concrete evidence that variation will actually be exercised.

Excess generality exists in almost all large systems and often in immense quantities. Such generality always costs time and memory and is a prime contributor to what one might call "system bloat". The above principle can be adhered to, of course, only if it is easy to introduce new generality whenever it becomes appropriate.

3. Mechanisms

Having in the prior section laid out a set of design principles that characterize L*, in this section we describe the main structural mechanisms that permit their realization. Most of the principles permeate the structure of the entire system. Drawing the mapping explicitly between principles and mechanisms leads to much redundant exposition and will not be attempted. The way these mechanisms dovetail with the principles is quite apparent at a surface level. The deeper evaluation of what unanticipated consequences the mechanisms bring with them, and whether the principles themselves yield good system implementations cannot be seen from a description of mechanisms alone. Some of these issues will be addressed in Section 4 on experience with L*.

3.1. Facilities

We describe first the scheme whereby an L* system is organized. This is not a critical mechanism, but will permit the description of the mechanisms to fall into place. We will return to the overall organization later.

L* is organized as a collection of facilities. A facility is an increment of code and data that provides a collection of interrelated functional capabilities to the system. Since the basic style is that of growing a system, a facility is not a self-contained module, but makes use of facilities existing at the time of its addition to the system. Thus, there is a graph of dependencies among facilities, since any facility requires that certain others already exist in the system for it to operate successfully.

The facility as an organizational unit is responsive to the principle of no designer's prerogative. The concept of module (Parnas 1972) implies in existing machine architectures that a base be provided to support the module structure. This designer-posited base is not itself a module and cannot be modified without destroying the system. No such base exists for L*; we envision L* systems being regrown from scratch with arbitrary modification and redesign.

3.2. Symbol System

The notion of a symbol system is widespread, although seldom formalized. It consists of a set of symbols and a set of data-types (symbolic structures) in which tokens of these symbols can occur. Besides the usual operations on the data-types (which create, manipulate and modify them), the essential operation is that of an association between symbol tokens and entities called their referents.[1] The associative

[1] Within the symbolic system these are always to data structures which represent in some general way the entities actually referred to.

relation is one of access: given the symbol token, access is had to its referent (representation). Sometimes there is a single such association (often called assignment), but systems can permit many such associations.

The decision to admit a uniform class of symbol so that any sort of entity could be referred to, coupled with the requirement for total accessibility, led to the following:

> Symbols in L* are identified with addresses: All symbols are addresses and all addresses are symbols.

This has far-reaching consequences. On the positive side, within the symbol structures that are basic to L*, any address can occur without causing some operation (such as printing, erasing or searching) to misbehave. One can build structures that refer to objects such as operating stacks, basic machine code, and even the registers of the underlying machine.[2] Furthermore, simplicity will be fostered, since the symbolic structures in the system will be as simply constructed as possible.

On the negative side, there is a limit to the number of symbols in the system, namely the size of the address space. But much more important, the mapping of a symbol to its referent is fixed by the hardware (i.e., a symbol refers to a fixed location in the address space, whether physical or virtual). Thus the freedom to assign symbols to referents, and especially to reassign them, becomes restricted. This is a genuine restriction, and one we will return to at several points; it has been accepted as the price for the benefits above.

Notice that symbols are internal to the computer. They can just as easily be created internally as by a user externally; in fact many are defined a priori.[3] For the user to work with any of these symbols he must attach an external name (e.g., in ASCII). We will treat this at length below, but it is important to note the difference between names and symbols: names are external character strings associated with particular internal symbols.

> A Symbol Facility provides the basic capabilities for creating and erasing symbols, and for doing the primitive operations that can be defined for symbols independent of what they refer to. These operations are tests of equality and inequality, and incrementing and decrementing.[4]

---

[2] If they are within the user's address space, as they are on the PDP10.

[3] E.g., in the PDP10 the registers and the so-called job data area (where the monitor stores user job-dependent information) are both within the user's address space.

[4] The latter exist because symbols are addresses, and would not be meaningful if symbols had been defined as an abstract set.

A complete capability for symbol manipulation requires also a flexible data-type for symbolic expressions. Not all data-types in a system will allow symbolic expressions, but there must be one such, and it will play a fundamental role in the system. It will be the medium used for all representational tasks which cannot be accomplished by other more specialized means. In L* this basic data-type is the linked list of symbols.

> A <u>List Facility</u> provides the processes proper to manipulating lists: getting the next cell, getting the symbol in a cell, inserting, deleting, copying, and erasing.

A single designative relation (here, between addresses and the addressed memory locations) is sufficient for all purposes, but quite cumbersome. L* thus provides a general mechanism for attribute-value associations. Given any two symbols, say X and Y, it is possible to create an association (along some attribute symbol, say A) from X to Y. Then given X and A one can directly retrieve the symbol Y. There can be as many associations (and as many different attribute symbols) as desired. This is an example of the positive benefit of choosing symbols to be addresses. Associations are permitted on any symbols, hence on any addresses.

Associations are realized by a hashing scheme. A noteworthy feature is that each attribute has its own hashing table, thus allowing the sizes of these tables to be independently determined and dynamically adjusted. This allows control over the space-time tradeoff. For example, if access is rare for some particular attribute, its hash table can be made small -- resulting in slower access but reducing the waste of empty table slots.

> An <u>Association System Facility</u> provides the capabilities for creating association symbols, creating, retrieving and deleting associations, and otherwise manipulating the association structures.

3.3. Universal Type System

All the data within L* is of some <u>type</u> and there is a symbol that designates each data-type (called the type symbol). Given a symbol, the type of the data structure it designates can be determined. Similarly, when a symbol is defined, its data-type must be specified (though this can be done by the system rather than the user). New data-types can be created at will. Figure 2 lists the data-types that are defined in the initial system, with the external names of their type symbols. This initial set is neither a complete set nor a minimal set; rather, it is what is necessary and sufficient for the set of initial facilities.

T/T	Type type
T/M	Type machine (code)
T/W	Type word (also integer)
T/L	Type list
T/P	Type program list
T/J	Type stack
T/K	Type character
T/KS	Type character string
T/A	Type attribute (hash table)
T/AL	Type attribute list (conflict list)

Figure 2. Initial data-types

Data-types serve three important functions. First, they permit type dependent processes. A print routine can print any input structure appropriately by first accessing its type, thus relieving the user of always knowing what type of structure is being printed. Second, space can be managed by data-type (and the initial system does so). Thus there may be several data-types which are identical in structure but are distinguished in order to manage the space they occupy (e.g., several areas all with lists, but of separate types, **T/L1**, **T/L2**, ..., to keep them segregated). This management can be seen as just more type-dependent processing (copying and erasing), and the implementation does in fact operate that way, but space management by data-type is still worthy of special note. Third, the interpretation of programs is totally type-dependent. This fact has very widespread ramifications for the basic programming language used within L*, which we treat in the next section.

Four requirements on the type system have emerged from the discussion so far:

(1) Every symbol, hence every address, must have a type (thus a process must exist which, for every symbol, delivers its type).

(2) Types must be dynamically creatable.

(3) Types are to be used in the interpretation of the programming language (thus the process for finding types must be very fast).

(4) Types of symbols must be dynamically changeable.

We have not indicated the relative frequencies of executing type dependent processes (such as print, erase, copy), of new type creation, and of changing the type of an existing symbol. It is clear, however, that all these will be very much less frequent than type interpretation, and thus they do not dictate the design of the type system.

This is an exceedingly hard set of requirements to meet on a predetermined architecture (e.g., PDP10, PDP11, IBM 360). For instance, storing a type symbol for each symbol (i.e., address) in the address space takes on the order of half the memory. Further, making the type access fast requires a simple algorithm. Since the type must be found given only the symbol (address), the type must be either a simple function of the address or else stored at a place accessed via the address. Extracting the type from the address makes type creation difficult, and changing types nearly impossible. Although the type may be considered as just an association from the symbol, this is not a possible implementation since all symbols in the association structure must themselves have types; nor is it clear that a hash table scheme is fast enough for requirement (2).

In the current version we compromise requirement (4) on changing types, but not the other three. We assign types in contiguous blocks (128 locations on the PDP10) by using a type table with one entry for each block (2048 entries on the PDP10). Access to this table can be made directly from the symbol (address) in two PDP10 instructions, which becomes the basic type access time. Changing types is effectively stymied because the types of a whole block of symbols are tied together and cannot be changed independently.

Actually, we have created a facility for dynamic types by exception, in which a block can be declared dynamic to allow each symbol in the block its own type. However, experience has shown that such facilities are not much used, probably because of their high cost rather than a lack of desirability.[5]

> A <u>Type</u> <u>Facility</u> provides processes for getting the type and testing types of symbols. It contains subfacilities for creating new types and for creating dynamic types.

3.4. PL*: The Programming Language

Often the preferred strategy for creating a complex program is to create a special problem-oriented language whose structure reflects the unique assumptions about the task. To maximize the number of application areas, L* anticipates the existence of many programming languages within it. The initial L* system, however, contains just two languages: a form of machine language (called ML*) and a general list processing language for manipulating the symbolic expressions (PL*). PL* is, in some sense, <u>the</u> L* language, but we emphasize that other languages and systems grown within L* are not necessarily built on top of PL*. Often, they begin that way and are converted to ML* under the press of selective optimization. ML* and PL* (and other languages like them) are to be distinguished from the language through which the user

[5] The cost is high: extra space for each symbol whose type is an exception to the block's type, and extra time for accessing the type of every symbol in the block.

at a terminal interacts with the system. This latter is called EL*, and will be discussed later.

PL* was designed to be the simplest possible list processing language. The basic data-type is the list. Normally, in the design of a programming language, the language itself is a unique and complex data-type, radically different from the types of the data structures on which it operates. List processing languages, on the other hand, have been able to use a single data-type for both program and data, thus providing a unification not possible in standard languages. This unification has quite real effects when it comes to programs that create programs.

Coalescing of data and program is not achieved simply by deciding to do so. The most fundamental property of a programming language is that it determines what data and operations are to be brought together, and when (and if) they will be executed. All programming languages must therefore have some way of distinguishing operands from operators (or functions). In a standard programming language this distinction has no counterpart in the data structures. Thus, the program data-type, in terms of its most basic requirements, threatens to be unmappable in any natural way into the other data-types. A list, for instance, is a homogeneous sequence of symbols without anything to distinguish operators from operands.

The solution adopted by LISP is to employ one of the natural features of the list (that it has a first symbol) to make the distinction between operator and operand: the first symbol in a list is to be the operator and all others the operands. This makes the program data-type different, but easily assimilable into the general list data-type. Indeed, it fits with a common encoding of data in which the first symbol on the list is taken as a "tag" or data identifier, with the remaining symbols in the list fitting the conventions determined by the tag.

The solution adopted by L* is to retain the homogeneous character of the list, so that the interpretation of every symbol is to be the same. Then the distinction between operand and operator cannot be given by the structure of the program (the syntax); it must be given by the nature of the symbols themselves (the semantics). This distinction is taken to reside in the type of the symbol. Thus for each type there is an interpreter, which is to be executed whenever a symbol of its type is encountered. We can express this in a principle:

> <u>PL*</u> <u>interpretation</u> <u>by</u> <u>type</u>: A list of symbols (S1 S2 ...) is to be interpreted by successively interpreting each of its symbols, S1, S2, and so on. A symbol Si is to be interpreted by executing the interpreter associated with the type of Si.

With this interpretation rule there is a distinct data-type for PL* (called T/P, for type program list), but it is structurally identical to the list data-type (T/L). The difference in the two rests in their associated interpreters; the T/P interpreter treats the list as a program, while the T/L interpreter treats the list as data. There will, of course, be interpreters associated with each of the types of Figure 2, and indeed with all types that are created.

The interpretation principle does not completely determine the character of the language; it depends intimately on the details of the individual interpreters.[6] Given that an interpreter has access to the data structures of the operating environment and to the program structure itself, there is wide freedom to specify further the character of the language through the actions of the interpreters.

The interpreter set for PL* abides in so far as possible by the following principle:

> Context independence: The interpretation of a symbol in a program list does not depend on the part of the program list not yet interpreted.

This principle, in conjunction with the one above, almost completely determines the character of the language. It has three general effects: (1) there are no incomplete expressions; (2) symbols establish a state within which the following symbols can be interpreted; and (3) operator-like symbols cannon be taken as operand-like symbols (for if they had to be interpreted in isolation, their interpretation would have been operator-like). These imply the following structural features of the PL* language:

> Post-fix: Operand-like symbols must come before operator-like symbols.
>
> Stack communication: Operand-like symbols must have somewhere to wait until the operator-like symbols come along, and they must do this without knowledge about the operator-like symbol.
>
> Goto-less control structure: Goto operators take operator-like symbols as operands, which violates the rule.
>
> Condition signal: A test must occur before conditional action based on it is possible; hence the effect of the test must be statisized.
>
> Explicit quote: There must be some way of obtaining an operator-like symbol as operand. Thus, some violation of the context independence principle must occur. A quote operator localizes this as much as possible.[7]

The fundamental reason for adopting the context independence principle is to make PL* simple to understand in terms of its underlying mechanism. There can be no complex actions that cannot be resolved into a sequence of simple ones. Equally, in

[6] The interpretation principle does, however, essentially determine the interpreter for T/P.

[7] I.e., the quote operator is a symbol that is interpreted prior to its operand and which acquires its operand without interpretation.

terms of interaction, the interpretation of the language can be broken at any point and additional processes inserted or executed.[8]

In terms of language design, we have long been interested in understanding the extent to which a strong context independence assumption is compatible with a language whose surface appearance is still very much that of a higher level language.

> The PL* Facility provides the operating environment for PL* along with the control processes to be used in PL* programs.
>
> There is also an Interpreter Facility which provides the set of interpreters used in PL*.

Some examples of PL* coding will tie down these various design decisions and also reveal the surface form of the language. Figure 3 shows a list named **L1** defined to have three elements **A**, **B** and **C**. In terms of the underlying linked list representation, there are three memory cells **L1**, **L2** and **L3**, each of which holds a symbol (A, B or C respectively) and a link to the next cell. We consider a program, called TBL, which tests if the symbol B is in the list. This program has a single input (the list to be searched), but is specific to the symbol B. The various component processes are listed in the figure. Vertically below the program list we have shown graphically the data stack (called Z); we have written a * at the bottom to indicate an indefinite number of other symbols that will not enter into the processing of TBL. Directly above the stack we indicate the condition signal (+ if true or succeed, - if false or fail).

[8] The explicit quote is an exception.

TBL -- Test if symbol B is in list

TBL: ((P S B =S .+ N F .R+) .- U)

Example list input on stack: L1: (A B C)

Internally: L1:[A | L2] L2:[B | L3] L3:[C | NIL]

Processes used in TBL (all T/M -- machine code routines)

P	Push top symbol on data stack (Z)
S	Get first symbol in list
=S	Test if two symbols are identical
.+	Exit if signal +
N	Get next location in list (tail)
F	Find list (set + if continues, else - and pop)
.R+	Repeat if signal +, else no-op
.-	Exit if signal -
U	Pop data stack (Z)

```
TBL: (    (    P    S    B    =S   .+   N    F    .R+  )    .-   U    )

       +    +    +    +    +    -    -    -    +
       L1   L1   L1   A    B    L1   L1   L2   L2
       *    *    L1   L1   A    *    *    *    *
                 *    *    L1
                           *

            +    +    +    +    +                        +    +    +
            L2   L2   B    B    L2                       L2   L2   *
            *    L2   L2   B    *                        *    *
                 *    *    L2
                           *
```

Figure 3. Example of a simple PL* routine

Z originally holds **L1**, the operand for TBL. TBL is a T/P list of three elements, the first being of T/P itself and given in extension as the sequence (P ... **.**R+). Interpretation of this list leads to interpreting each of its elements. The first symbol is P. This is a T/M symbol, which is to say machine code, and the T/M interpreter simply makes a subroutine call to the machine code routine. The effect of this is to push Z, resulting in two instances of **L1**. The interpretation proceeds sequentially. S gets the symbol in the cell **L1** (i.e., A); B is a data symbol and its interpreter pushes it onto the stack; =S is the test for symbol equality, which sets the condition signal - since A and B are not the same (note that processes in general consume their inputs); **.**+ is a conditional exit, which is a no-op here since the signal is -; N gets the next cell after **L1** (i.e., **L2**); F sets the signal + to indicate the list actually continues (N might have produced the end of a list); **.**R+ is a conditional repeat, which moves control back to the front of the list if the signal is +.

The second loop through the sublist continues as before (the second line of trace in Figure 3). This time =S gets a positive result, since it has found the symbol, so that the exit is taken. Hence, the next symbol interpreted is **.**- (a no-op here since the signal is +). The final symbol is U, which pops the stack removing the temporary working symbol **L2** (the moving pointer into the list).

The post-fix character of PL* is evident. Processes simply operate on the operands that have been developed in the stack. To use TBL on a different list, say **L7**, one would write:

... **L7** TBL ...

L7, being a data symbol, would be pushed on the Z stack and then TBL, being T/P, would be executed on it, just as above. The goto-less character is evident in the control operators, **.**+, **.**- and **.**R+. Note in particular that looping is handled in a way that is symbolically not much different from giving a superordinate command (e.g., (REPEAT ...)), but conforms to the requirement that it be a context independent action. It is apparent from the example that the language is a mixture of high level and low level constructs. For instance, stack management is explicitly the responsibility of the program.

Some variations on this simple routine will convey some additional aspects of the language. One would like to write TBL simply as:

TBL: (P S B =S **.**+ N F **.**R+)

The additional **.**- and U are required for cleaning up the stack, which is done automatically by F at the end of the list, but not by **.**+ on the positive exit. The following additional control primitive is defined in PL* (and is indeed required for completeness):

.-H -- Exit on - , else remove the next higher level

A primitive such as .-H is required because it must be possible to control for a given program level the continuation of levels above.[9] Using .-H one can recode TBL as:

```
TBL: (P S B =S (.-H U) N F .R+)
```

Thus we have localized the code for exiting and cleaning up the stack. If we wish we can introduce a new control routine:

```
.U+: (.-H U)   -- Exit if +, popping stack
```

With this we can rewrite TBL once more as:

```
TBL: (P S B =S .U+ N F .R+)
```

Actually, the routine is incorrect in its handling of empty lists, since it goes through the motions of testing the "first symbol" of the empty list before quitting.[10] Another variation can be written that handles this correctly by moving F to the start:

```
TBL: (F .- P S B =S .U+ N .R)
```

The language also admits recursion, so that yet another alternative form for TBL is:

```
TBL: (F .- P S B =S .U+ N TBL)
```

TBL has been written with a single argument, the list. It is more appropriately written with two arguments. Let us then define another program:

```
TSL   -- Test if symbol (0) is on list (1)
```

where (0) (1) ... designate position on the Z stack. Then we obtain:

```
TSL: (Z0 I (F .- P S Z0 S =S .U+ N .R) Z0 D)
```

where I inserts (1) into list (0), and D deletes the top of list (0).

We have used a cell Z0 (T/L) to hold the symbol to be tested. Thus we must insert the symbol from the Z stack onto Z0 at the beginning, and again delete it from Z0 at the end. To access the symbol from Z0 for the test, we input Z0 to the stack and

[9] Besides .-H, there also exist .+H, and .H (unconditional removal of the next higher level); similarly there exist .R, .R+ and .R-, ., .+ and .- .

[10] The routine works correctly on empty lists because S on an empty list merely delivers the symbol NIL as output.

then execute S on it. Otherwise TSL is just the same as TBL. We can collapse these operations on Z0 by defining some additional routines:

```
IZ0: (Z0 I)    SZ0: (Z0 S)    DZ0: (Z0 D)

TSL: (IZ0 (F .- P S SZ0 =S .U+ N .R) DZ0)
```

If we wanted to generalize TSL further to take as input a generalized test, rather than just symbol equality, we might define:

```
TXL   -- Test if there is a symbol on list (1) satisfying test (0)

      TXL: (IZ0 (F .- P S SZ0 .X .U+ N .R) DZ0)
```

where .X executes (0).

One of TXL's inputs is now a process; to use TXL we must use the quote process. We can illustrate this by reconstructing TBL from TXL:

```
TBL: (.Q (B =S) TXL)
```

The quote, .Q, is a T/M routine that puts the next symbol in the program list (here the subprogram (B =S)) into the stack and advances interpretation past it. Thus TXL will be the next symbol interpreted after the .Q.

The examples above illustrate the simplicity of the PL* language. To summarize, PL* is interpreted by type, maintains context independence (no syntactic structure), is post-fix, and uses a stack for operand communication. Let us now examine the other initial programming language embedded in L*, the ML* machine language.

3.5. ML*, the Machine Language, and Stacks

3.5.1 ML*: Machine Language

The use of machine language must remain integral to L*, since it is the means through which the machine is ultimately controlled. One shields the implementer-user from access to machine language only by committing a major act of designer's prerogative -- of deciding that the forms of access determined by the original software designers (here the L* designers) constitute the only means by which the machine will be utilized. But time and space efficiencies are of the essence -- that computational completeness remains available to the user does not suffice. This is especially true for an implementation system, whose users will create still undetermined object systems.

Access to the machine language does not imply that an initial component of L* must be an assembler of some form. In fact, L* adopts a specific principle:

> <u>Machine access</u>: Access to the basic machine is to be obtained via symbolic structures created within the system itself.

Thus, though there are assemblers and compilers in L*, they are not available as primitive facilities, but are constructed by means of PL* programs and data structures. What must be guaranteed (though it is not difficult) is the possibility of obtaining full control of the machine ultimately. This occurs by having the word be a basic data-type (T/W) with primitive operations that include the standard arithmetic, boolean, and shifting operations. Given that words with arbitrary bit content can be easily fashioned, it is straightforward to construct, within PL*, simple assemblers, macro assemblers and compilers.

This principle, with the implied delay in obtaining facilities for assembling and compiling, rests solidly on the design goals of L*. To make such facilities part of the initial system poses an almost impossible tradeoff between initial simplicity and ultimate facility and flexibility. Assemblers, as a genre, are deficient in the facilities they provide (compared to, say, LISP or PL/1) precisely because they are "initial" systems. Such systems are not only lean, they are inflexible. HLLs solve this problem by creating a large initial system (the HLL itself, e.g., PL/1). This at least obtains facility, though it doesn't obtain flexibility.

Basically, the computer itself dictates the machine language. However, using machine language within a system requires various conventions that constitute, in essence, a particular sublanguage. Thus, ML* is the machine language plus a set of conventions:

> The operating environment consists of three stacks: (1) a control stack, holding the current instruction; (2) an operand stack; and (3) a test condition stack.
>
> All language systems will use a common operating environment, if possible. In particular, ML*, PL*, EL* use the same operating environment.

The three-stack operating environment is dictated by the requirements of common machine language and PL* use, these being the initial language systems. However, it is also an appropriate environment for realizing a wide variety of higher level languages (like LISP or Algol). The major restraints on the machine language programming are: (1) All argument-passing must use the operand stack (specifically, registers may not be used); (2) Signal communication must use the test condition stack (e.g., no skip returns may be used to return a signal); (3) Working register usage is limited to those not required to provide the three-stack operating environment.

The communality of use over language systems is primarily an efficiency issue. Without this, passage of control across a language boundary also requires shuffling of data, in addition to the transfers required by the basic mechanics of the calling sequence. The costs involved are substantial. In the initial PDP10 systems through L*(G) we explored a number of variations on mechanisms that kept the operating environments separate, and the tradeoffs are quite clearly in favor of communality on standard computer architectures.

An important consequence of these conventions is that there is only a single routine for a single function. For example, consider the function of inserting a symbol in a list; this is needed in both language environments, PL* and ML* (and others as well perhaps). There is a process named I (itself written in ML*) which is to be used within both ML* and PL* programs to carry out the insertion function. Thus, there does not have to be any duplication of function across the two language environments. This is in fact an extremely strong contributor to simplicity in the L* structure. As for executing PL* programs from within ML*, designer's prerogative by the hardware architects prohibited this from happening as it should. The PL* call must be surrounded with a small machine language cliche.[11] Except for this, the situation is symmetrical. Most interpreted language systems are hierarchical, with the interpreted code lying "above" a machine code base. The L* language structure is not hierarchical. Many PL* routines are called from within ML*.

3.5.2 Stacks

The abstract data-type called a stack is an extremely useful data structure in software systems, wherever there is interruption and return in the use of resources (viewed quite abstractly). Phrase-structured languages, interrupt service, subroutine hierarchies, and variable-binding hierarchies are only a few examples. Thus an implementation system needs a stack data-type, both for its internal use and to employ in object systems. There are literally dozens of stacks in use in a typical L* system, at all levels of system organization.

Stacks can be implemented in many ways. The most familiar is a sequential stack occupying a continuous interval of the address space, in which push and pop are accomplished by incrementing and decrementing addresses. But in a system which already has list processing, a quite natural choice is to map stacks onto lists -- a subset of the list primitives are isomorphic to the standard stack operations. Using the notation in PL*:

[11] Which puts the PL* symbol on the Z stack and then calls the PL* interpreter.

	List	Stack
S	Get symbol in cell	Read top symbol
R	Replace symbol in cell	Replace top symbol
I	Insert symbol at front	Push new symbol on top
IC	Insert copy at front	Push (double top symbol)
D	Delete	Pop

General advantages of list stacks include the immunity to overflow (since they are not contiguous blocks) and the availability of the more powerful list operations when needed (insertion, deletion and reading other than the top entry). When used for the operating environment of PL*, list stacks allow easy exploration and modification of that environment (e.g., in the coding of new control operations which directly manipulate operating stacks). In the PL* environment stacks hold symbols (i.e., addresses, which occupy half-words on the PDP10); hence space and time costs are about equal between list stacks and sequential stacks (usually considered the most efficient implementation). All of our initial versions of L* (through L*(G)) used list stacks[12].

The machine language operating environment must be realized with sequential stacks in current standard architecture. Designer's prerogative by the hardware architects has been exercised in the subroutine call and return functions to make all other choices for stack implementation prohibitive. Thus, a choice of list stacks for PL* (producing homogeneity there) produces a split between the operating environments of PL* and ML*, with the negative consequences discussed above.

The current L* systems have adopted the other choice. Stacks are realized as a distinct data-type, T/J[13]. Then the PL* operating environment is identified with the ML* environment, as above, with the consequent simplicity and speed increase.

Stacks are now a general data-type providing functions which partially duplicate list functions. Corresponding to the processes on lists (S N R I D F ...) there are processes on stacks (SJ NJ RJ IJ DJ FJ ...). Stacks are realized with a pointer structure that keeps lower and upper stack bounds plus the actual pointer. The stack memory area is a separate block, which can be relocated in memory to provide expanded or contracted memory space. An important advantage in having stacks is their preferred use in object systems which do not wish to import list processing.

By providing an efficient stack data-type and a machine language (ML*) integrated with the rest of the L* system, we have provided mechanisms for achieving total accessibility to the underlying machine and for aiding selective optimization. ML* and PL* are the two initial programming languages provided in L*, but neither has any

[12] Note that the examples in this paper use list stacks (e.g., for ZO and ZQ); they could have used sequential stacks.

[13] J for nothing, but think of the stem of the J as the stack and the cup as the overflow test.

privileged status. Other programming languages may be added at the same "level" as ML* or PL*. Now, let us describe the language used to communicate between the user at a terminal and the L* system, EL*.

3.6. EL*: The External Language

L* treats PL* as an internal language -- a set of data structures in memory that can control processing. It would treat similarly any number of other languages, such as LISP. The human user, of course, resides outside the computer system at a terminal, and he communicates with L* through some other language, or at least through some notation for the internal languages. We call this external language EL*.

EL* must meet several requirements that separate it sharply from the set of internal languages:

> <u>Names vs. symbols</u>: EL* is written as a sequence of characters (assuming text, not graphic, input devices). Hence correspondences must be made between sequences of characters and internal symbols. We use the term <u>name</u> for a character sequence that maps into an internal symbol.
>
> <u>External fidelity</u>: It should be possible to make the external language isomorphic to any given internal system (this follows from the requirement for simplicity).
>
> <u>Total accessibility</u>: All internal symbols and structure must be representable within EL* (i.e., ML*, PL*, and all languages to be subsequently created).
>
> <u>Growth into object systems</u>: It must be possible to transform EL* into a problem-oriented language for an object system, with an operating environment sealed off from the total L* environment. (The full range of standard notational and linguistic devices must be easily created within EL*.)
>
> <u>Dynamic modification and simplicity</u>: EL* must be capable of being modified interactively by someone working within EL*. (This also implies a simple mapping between EL* and internal structures.)

The requirements for fidelity and total accessibility for all internal languages imply that EL* cannot simply be another language, analogous to PL*, with a particular data-type and set of interpreters. Thus EL*, though a language functionally, must be conceptually orthogonal to the other languages in the system.

We first discuss EL* considered as a sequence of symbols, assuming the mapping from character strings to names to symbols has already taken place. This is the level of syntax, but also includes the higher levels of semantics and action. Then we will return to consider the lexical processing that produces the symbols.

3.6.1 Syntax, Semantics, and Action

Standard syntax schemes imply the existence of a grammar and a parser between the user (i.e., the creator of EL* surface structure) and the corresponding internal data structure. Given the requirements for syntactic power, such a scheme would seem to interpose a veil of complexity at odds with the desired fidelity and simplicity of the total system.

The central feature of a solution to these multiple requirements lies in taking a "process-view" of syntactic interpretation (Newell and Freeman 1968). Namely, the interpretation of an EL* sequence of names is to be carried out as a sequential process. Some names will correspond to processes whose immediate execution will carry out the analysis of the input stream, to convert it into a sequence of internal actions or structures. Other names will correspond to lexical items and will become the internal symbolic data.

Let us capture this in a definition:

> Sequential Process Grammar: A linear sequence of symbols, each of which is either active or passive; active symbols are interpreted immediately in an operating environment that includes access to the language stream. The active symbols are called syntax actions.

For EL*, the stream of characters must be converted into a sequence of names, and these names must be mapped into their corresponding internal symbols. Then interpretation implies, as usual, execution of the associated interpreter according to type. Of necessity, then, the operating environment will be the same operating environment used by these interpreters.

It remains to show the extent to which such a scheme can realize appropriate surface structure. Though we do not know of any other programming language that takes exactly this course, the scheme is closely related to the original formula translation schemes introduced long ago by Samelson and Bauer (1960), and to the pushdown schemes of Floyd and Evans (Evans 1964). EL*, however, does not form a closed system, but operates within the environment of internal computation.

EL* is to be used for realizing context dependent surface syntax of all kinds; in doing so there is no reason to adhere to the context independence principle adopted for PL*. On the other hand, a sequential process grammar, with its execution of independent syntax actions, lends itself to a strict adherence to the principle. This is important in realizing an external isomorph of PL*, but is also useful more widely in interactive programming.

Some simple examples will make the scheme concrete. Figure 4 lays out a scheme to define a simple list, (A B C). Each of the five character strings, "(", "A", and so on, is a name. There is no distinction between identifiers and syntactic marks. More accurately, the distinction is encoded into whether a symbol is active or passive (as indicated in the figure by a̲ and p̲). However, as we shall see, any symbol can be active or passive and this state can be changed dynamically.

The first symbol (is active; being a PL* program it is executed. The result is to put a symbol into the stack to act as a floor. This symbol should be forever unnamed, so it is simply held in cell called **FLR**. We call the symbol address **97**, just to call it something. The next name **A** maps into a passive symbol; it is pushed onto the stack, which is the fate of all passive symbols. Thus, the symbols corresponding to the names B and C also end up in the stack. Finally) corresponds to an active symbol which also designates a PL* program. It creates a new symbol (via the CR process) and goes into a loop transferring symbols from the stack onto the list, while looking for the floor in the stack as a signal that it has finished. The symbol for the list is held in cell Z0 and all the inserts are made to the front (by I), so that the order is the same as that initially written. The symbol for the list (called here **632**) is output to the stack.

```
Typed at the terminal:    (A B C)


       (     A     B     C     )

       a     p     p     p     a                a = active, p = passive


   *     97    A     B     C     632
         *     97    A     B     *
               *     97    A
                     *     97
                           *

        "( : (FLR S)

        ") : (T/L CR IZ0 (P FLR S =S .U+ SZ0 I .R) SZ0 DZ0)
```

Figure 4. Defining list structures

The scheme of Figure 4 suffices for any list structure. Tracing through the example below will show how each list is created as a symbol and becomes part of its embedding list:

(A (B ((C) D (E F) G H)))

To add the syntax for assigning the symbol for a list (**:**) we can proceed as shown in Figure 5. The **:** is a binary operator and cannot complete its own operation until both of its operands are complete. In particular, **:** must delay until its right operand is complete, which will not happen until after it has itself been interpreted. A natural device for this is to create a delayed process that fires when the operand is complete. In the present context, only **)** will know this, so it must be given the responsibility for executing the delayed process. As Figure 5 shows, **:** leaves **L1** in the stack and puts a process on another stack, **ZQ**. **)** is modified to execute what is on the **ZQ** stack after building the list. It does this by a routine **.XQ**, which we discuss below. The delayed action exchanges the first cell of the list to be that associated with the specified first symbol, **L1**, rather than the created symbol used by **)** (using **RW**). The stack manipulations shown (V is reverse, **P1** is push second element) are to preserve **632** (the un-named list structure) until the end so it can be erased (erasing is done explicitly in the basic L* system).

```
    L1   :    (    A    B    C    )

    p    a    a    p    p    p    a


*    L1   L1   97   A    B    C    *
     *    *    L1   97   A    B
                *   L1   97   A
                    *    L1   97
                         *    L1
                              *


        ": : (.Q (V P1 RW E) IQ)

        "( : (FLR S)

        ") : (T/L CR IZ0 (P FLR S =S .U+ SZ0 I .R) SZ0 DZ0 .XQ)

        .XQ : (SQ .X DQ)

        SQ : (ZQ S)    IQ : (ZQ I)    DQ : (ZQ D)
```

Figure 5. Defining a named list

Our purpose of presenting this much detail is to show how syntax can be built up with little effort. In the **ZQ** stack and **.XQ** we have essentially all that is needed to add the full complement of binary operations (e.g., relations and arithmetic). In the actual system, matters are somewhat more complicated, because one must handle multiple types and redefinition of structures.

L* is a fully interactive language, so that there is the possibility of arbitrary immediate execution as well as delayed execution. The analysis of the EL* input stream already implies immediate execution of active symbols; to obtain immediate execution of passive symbols there exists a process called **!** (itself active, of course). The process **!** is actually identical to the execute process **.X** (except that **.X** is not active); **!** interprets (executes) the symbol on the top of the Z stack. Hence, in the following typed in sequence, the first two names put **L1** and **TSL** respectively on the stack and **!** executes TSL on input **L1**, just as we saw in Figure 3.

... **L1 TSL !**

This example should help emphasize that not all (or even most) PL* programs are active. Most are passive and are treated in EL* simply as additional data to be put away in newly constructed programs.

EL* uses the same processes as are used in PL* (or whatever language is being worked with). It achieves the desired isomorphism provided that the external syntactic structure can be realized by means of syntax actions that build isomorphic structures, which in general is possible. Conversely all of the processes that are available in EL* must also be available internally. This can be appreciated, for example, in copying a list. Suppose we wanted to copy a known list, **L1**:

```
        CL1  -- Copy list L1
```

Then an obvious way to do this, in analogy to the way lists are built externally, would be (schematically):

```
            CL1: ( <contents of L1> )
```

Since (and) are simply symbols that designate PL* programs, we can use these internally as well as externally. All we need to do is treat (and) passively. An actual routine along these lines is:

```
            CL1: ( '( L1 '←L ') )
```

The quote ' is an active symbol that passivates the following EL* symbol. Thus the outer ( ) define the list **CL1**, just as always. The inner '(... ') are the same two processes, but they exist internally as part of the process **CL1**. The process **←L** dumps the contents of a list into the stack; since it happens to be active as well as (and), it must also be passivated with '.[14]

We can provide one last illustration by writing the code for ←L, as it might have been defined:

```
            "←L : (F ,- P S V N ,R)        ←L ACT!
```

The character ← happens to already exist as a name, so it is necessary to use " before the name ←L to prevent it from being mis-recognized as ← followed by **L**.[15] Then follows the PL* program, made up of the usual primitives. This defines ←L, which is now simply a routine, like **TSL**. To make it active we input it, input the activating process ACT and do an immediate execution with !. From that point on ←L is active.

[14] It is of course possible to passivate ' itself by writing ' '.

[15] The operation of " will be explained more fully in the following section.

3.6.2 Lexical Recognition System

We have assumed above that EL* can be represented as a sequence of symbols. In one respect this already differs from standard practice, in that all syntactic marks (e.g., (or :) correspond to symbols in the same way as what are usually termed "identifiers". We now need to show how the character stream is segmented and converted into a sequence of names, which then become associated with internal symbols.

Two requirements are particularly pressing for a lexical recognition system:

> Flexibility: It must be able to encode a great variety of segmentation and classification rules, to support the growth of object systems from within. (In particular, it must handle punctuation and syntactic marks in a way homogeneous with other lexical items.)
>
> Efficiency: It must be highly efficient -- time efficient since it sits in the basic interactive loop of the system, and space efficient since users can be expected to use many names. (Systems with thousands of names must be practicable.)

The central recognition system is composed of two-way symbol tables, called dictionaries (i.e., name-symbol and symbol-name correspondences). A dictionary scheme requires further specification. (1) What class of names will be admitted to a dictionary? First, call a lexical recognition point a point in the character stream such that the prior sequence of characters has been completely segmented and recognized and the following sequence is unanalyzed. (2) What dictionary applies at a point (if several exist)? (3) What candidate names get submitted to the dictionary for recognition at a point? (4) Given that more than one dictionary entry is satisfied at a point, how is the ambiguity resolved? Standard practice is to do rule-based segmentation of the character stream to determine a single candidate, such that ambiguity cannot arise, admitting to the dictionaries only such names as are consistent with the segmentation rules. Multiple dictionaries in the form of nested block structure are also standard.

The flexibility requirements imply a different approach. EL* adopts the following principle:

> Longest-recognizable: That name will be recognized which is the longest name that matches, starting at a recognition point (from within the applicable dictionaries).

Some examples will clarify the principles (where we always take the recognition point to be at the far left). Under Text are example inputs, each resulting in the recognition under Recognize when the names under Entries already exist in the dictionary:

Input Text	Dictionary Entries		Recognize
.+H	.+	.+H	.+H
L\ABC	ABC	L\	L\
(B	(B	(	(B
<=	<=	<	<=

The principle of the longest-recognizable is a semantic segmentation scheme which couples segmentation to the contents of the applicable names. It permits one to have overlapping notations, not restricted by a set of artificial segmentation rules. The phrase "artificial" is used advisedly, since it appears to informal observation that humans operate perceptually closer to the longest-recognizable principle than to any other segmentation scheme in common use.

A typical example of the use of this principle within L* can be seen in the problem of building lists of different types. Parentheses, (...), will build a list, but do not specify its type (e.g., T/L, T/P). This is done by default, properly enough. However, a notation is required to declare the type specifically:

(...)	Build a list of default type
L\(...)	Build a list of T/L
P\(...)	Build a list of T/P

The longest-recognizable principle recognizes L\(as a whole in preference to segmenting before the (. Given that we had defined a new list type, say T/Q, we would of course face the problem of getting Q\(into the dictionary in the first place. This is accomplished by the double quote:

```
"Q\( : ( T/Q '\( )
```

Double-quote is a symbol just like any other; it exists in the dictionary and is active. It signals the recognition system to recognize all characters up to the next space as a new name. Notice in this that the general routine \(also exists which will take any type symbol as input and start a list structure of that type; it is normally active, so it had to be passivated to get it incorporated into the routine. In fact, we could have not bothered to define Q\(and instead simply written:

```
T/Q\( ... )
```

The longest-recognizable principle would have segmented T/Q (which is in the dictionary), and then \(.[16]

The question of putting new names into the dictionary requires additional design decisions. In general, EL* avoids declarations (a design decision generally favored by interactive languages and not by batch languages). In cases of ambiguity, as above, a declaration (via ") is unavoidable. But generally if a symbol is mentioned and not recognized, then it is an implicit declaration of a new entity.

Implicit declaration poses a problem for a semantic segmentation scheme. At what point is a newly input name complete? To identify termination points for recognizable input names, the character set is divided into several classes: name characters, digit characters, boundary characters, and participating boundary characters. The name characters generally include all alphabetic characters and some of the special characters. Boundary characters act as rigid boundaries for recognition, and include characters like space, tab, carriage-return and line feed. Names do not contain rigid boundary characters. Participating boundary characters act as conditional boundaries for name recognition and can be part of names. They generally include most of the special characters. For example, if :, (, and) are participating boundary characters and space is a rigid boundary, then:

```
ABC:(AB CD)
```

would be parsed so that recognition would be attempted on ABC:(AB, ABC:(, ABC:, and finally ABC. If none of these possible names is found in any of the relevant dictionaries, then the last will be implicitly declared. The class assignments for characters is under control of the programmer to allow maximum flexibility.

The final decision at this level in the recognition system is what sort of name contexts are available. EL* permits an indefinite set of dictionaries. An ordered list of these is available at a given recognition point. Normally this is block structured, but can as easily provide sealed off lower contexts which do not have access to higher ones. This latter scheme serves the growth of object systems which should exist in an isolated world as far as the user is concerned.

3.6.3 The Dictionary Mechanisms

The description so far does not provide the fundamental mechanisms out of which the dictionary shall be built. Principles, such as longest-recognizable, are of course only a particular (useful) design choice. The fundamental scheme must permit the creation of name segmentation and dictionary schemes of arbitrary variety.

[16] This assumes that T/Q\ and T/Q\(are not defined (i.e., segmentation depends on the semantics, which is to say on the actual set of names being used).

The problem is analogous to that faced by EL* for the higher level syntax, and the solution adopted is fundamentally the same -- namely, to take a "process view" of recognition:

> Sequential Process Recognizer: A linear sequence of characters, each of which designates a symbol, which is interpreted immediately in an operating environment that includes access to the character stream. (The symbols are called character actions.)

The function of accessing a dictionary is distributed to the collective actions of the characters starting at the recognition point. The interpretation of syntax actions occurs immediately as they become recognized. From one point of view there is only the interpretation of a sequence of character actions. In general, there cannot be separate levels of processing that create first a representation of a character sequence, then a representation of a name sequence, then a representation an EL* symbol sequence, then an internal parsed structure, which is then executed. Subsequent processing at all levels (including that of characters) can depend on the processing at all higher levels.

It remains to be shown how to realize various dictionary schemes by such a mechanism. Such detail is beyond this paper. We have constructed by these means multiple linear table dictionaries, discrimination net dictionaries, hashing table dictionaries, and others -- representing a wide variation of space and time tradeoffs.

There is nothing inherent in the Sequential Process Recognizer that restricts it to dictionary lookups. For instance, it is used for conversion of digit sequences to numbers. More generally, a user interface where all the syntax is conveyed by punctuation marks could be realized by character actions rather than by syntax actions. Indeed the same actions could be used, simply associating them to the characters rather than to the names.[17] EL* renounces this possibility in order to provide the ability for syntax to reside in arbitrary character strings (e.g., <=).

3.7. The L* Kernel

L* is grown from a kernel. That is, there is a small body of code and data which constitutes a system that is sufficient to run. From that point on, all of the additional facilities are added to the system using its own mechanisms.

The kernel of L* is usually measured by the amount of machine language it requires, for this somehow measures how much basic mechanism has been defined. The size of the kernel, measured this way, is given for several implementations in Figure

[17] In fact, various command actions (e.g., in the editor) are associated with standard control characters such as carriage-return, line-feed, or altmode.

6. It is about 1600 words on the PDP10 and 2500 words on the PDP11 standalone (either a simple PDP11 or C.mmp, a multiprocessor (Wulf and Bell 1972)). The types which are represented in Figure 6 account for most of the space in each system. The ML* language is composed of machine code, or type T/M. The PL* language is composed of interpreted programs of type T/P. The T/W data include buffers, tables, stack work space, and single word constants and variables. The T/L, T/A, and T/AL data are symbolic lists and associations. The T/KS data are character strings for the names in the system. The L*A (A-level) system is the complete basic L* system that most users use. Note that the 8080 version is described in terms of 16 bit words instead of 8 bit bytes.

The sizes for the systems are fairly constant (in terms of bits) except for two cases: L*(I) is broken into two segments, with the high segment (the numbers in parentheses) only loaded on demand; and L*C.(D), which contains facilities for multiprocessing and for dealing with overlayed pages in its address space. It is interesting to note that the existence of an underlying operating system does not greatly affect how much machine code is required in the kernel.

Machine	PDP10	PDP10	PDP11	C.mmp	ALTO	8080
Word size (bits)	36	36	16	16	16	16
Operating Sys	TOPS-10	TENEX	None	Hydra	None	None
L* Version	L*(I)	L*(I)X	L*11(H)	L*C.(D)	L*ALTO	L*8080
Kernel						
Machine code	1.6	1.6	1.8	5.7	2.5	2.5
T/P	6.6(1.6)	7.8	4.0	6.7	6.0	5.0
T/W data	6.1(1.2)	7.2	5.0	10.0	5.8	4.2
T/L	4.1(0.4)	4.5	1.0	1.1	0.7	0.6
T/A, T/AL	1.0	0.9	0.3	0.6	0.8	1.1
T/KS	1.5(0.4)	1.9	-	4.8	2.3	2.3
L*A (A-level)	25(+10)	32	13	32	18	16

Figure 6. Sizes of various L* versions in thousands of words

The machine code in the kernel alone does not produce a self-sufficient system. Some PL* programs and data are required as well to make a minimal self-sufficient system. In the A-level system, the PL* programs account for about five times the size of the machine code, and the data accounts for about nine times. The large proportion of data is largely due to the space needed for names (character strings) and associations between names and symbols.

There are important reasons for L* to have a small kernel. First, it limits the amount of basic system that must be understood by a user (i.e., a system implementer). Thus, it contributes substantially to the total accessibility of the system. Size alone does not determine how accessible the kernel is -- if it were a single entity, then even 1600 words could be formidable. In fact, the kernel consists of a highly rationalized set of routines, all built around the same operating environment. Thus in a typical kernel there are about 130 routines, most of which are small, independent language operations (e.g., S, N, .+). There are about 15 routines larger than 20 instructions, and the largest is about 80 instructions. The hierarchy among these routines is shallow, with calls that nest deeper than two being rare.

The fact that there is only a single routine for accomplishing any function, as described in Section 3.5 on ML*, is an important contributor to the simplicity of the kernel. Although there are routines that the user does not normally look at (e.g., those in the space management or interface facilities), there is no special subsystem of internal housekeeping routines. There is nothing hidden under the floor -- and indeed no floor at all.

A second reason for having a small kernel is that L* is constructed to be repeatedly grown into an object system. Such a growth process is not just augmentation, but can involve modification and replacement of existing facilities. (For example, a new dictionary scheme, or a type system with added features). Regrowth is often the strategy of choice in such cases, backing down to a minimal system and putting the new version together from scratch. The smaller the kernel, the easier such a process will be. Indeed if the kernel is small enough and simple enough (as we believe the L* kernels are), even revisions of the kernel are possible without too much difficulty.

Third, with a small and simple kernel it is much easier to produce a completely debugged basic system, such that most errors encountered can be assumed to reside in newly-added code. This has indeed proved to be case -- L* itself is highly stable.

Fourth, portability from one environment to another and from one machine to another is made much easier. We did not have portability explicitly among the major design goals of L*, but it is clearly important. And indeed, we have brought up versions of L* on several different computer systems.

Additional benefits derive from the small kernel, but they affect mostly the basic system designers themselves. Thus, it has been possible to carry out a substantial number of iterations of the basic L* system. Each of the iterations has experimented with radically different solutions to the various basic system problems. This would surely not have been possible if the system itself were as large, say, as a compiler for a higher-level language.

3.8. Complete Facilities

In Section 3.1 we said that L* was organized by facility. Throughout the section we have noted that the various mechanisms we have discussed were included in such and such a facility.

The design goal of producing a complete operating environment is to be realized by providing a complete set of facilities. In creating an object system the user should find available as facilities all the software tools of whatever kind that he needs. Since all facilities exist within the same system, and since this also includes the object system as well, several consequences follow:

> All software tools will be evoked and used within the same set of conventions.
>
> All software tools can be modified, examined and debugged (for even much-used systems experience an occasional bug) in a common way.
>
> To the extent that new tools are required, they can be added within the same framework and in real time.

To make this concrete, Figure 7 lists all of the facilities in the so called A-level system. This is a stage of growth (starting from the kernel) where enough facilities have been added to provide what any beginning implementer wants. It is the version normally evoked at the monitor level on the PDP10 -- what you find in the standard documentation.

System.support facilities

- Interpreter
  - Interpreter.step
  - Fast.interpreter
- Type (T/T)
  - Dynamic.type
  - New.type
- Symbol
- Space.management
  - Block.space.management
- External.interface
- Extended.system
  - System.initialization
  - Save
  - High.segment

Data.structure facilities

- List (T/L, T/P)
  - List.structure
- Pair.list (T/AL)
- Block
- Stack (T/J)
- Character.string (T/KS)
  - String.conversion
  - Byte.string
- Word (T/W)
- Association.system
  - Association.list (T/AL)
  - Attribute (T/A)

Language.environment facilities

- ML* (T/M)
  - System.macro
- PL* (T/P)
  - Operating.state
  - Control
    - Iterative.control

Language.environment (cont.)

- EL*
  - Recognition
    - Character.action (T/K)
    - Executive
    - Name.context
    - Local.name
    - Fast.name
    - Type.recognition
    - Undefined.symbol
    - Name.assignment
  - Print
    - Print.machine.code

Utility facilities

- Debugging
  - Error.detection.and.recovery
  - Undefined.T/P
  - Tracing
    - PL*.step
    - PL*.breakpoint
    - ML*.breakpoint
    - General.breakpoint
    - Symbol.monitor
- File
  - File.read
  - File.write
  - File.ppns
- List.edit
- Assembly
  - Machine.opcodes
  - Machine.instruction.assembly
    - Macro.assembly
  - General.word.assembly
  - Machine.opcodes.complete
- Translation
  - Translation.update
- Space.accounting
- Time.accounting

Figure 7. Facilities of the A-level L*(I) system

We do not enumerate the contents of these facilities because they are the facilities to be expected in any total programming environment: editors, debuggers, compilers, assemblers, accounting systems. There is nothing sacred about the exact set. It reflects a uniprocessor with a non-L* operating system -- other facilities show up in stand-alone versions of L*. Likewise various multiprocessing and overlay management facilities show up in L*s that inhabit C.mmp. There are some notable omissions from the set of facilities in Figure 7, reflecting priorities and existing alternative systems in our CMU environment. For instance, no documentation facility appears, nor does a text-editor facility (which does exist in other versions), a structured programming facility, or an optimization subfacility of the translation (compiler) facility. These are not missing as matters of principle; all should be there, and will be eventually.

As noted earlier, facilities are not self-contained modules. An important reason for this is the very large amount of function that is represented in Figure 7. Each facility should add only a minimal amount of coding to accomplish the incremental function. In practice, this means there are strong dependencies between facilities: facility Y requires the existence of facilities X1, X2, ... in order to operate. Normally, this takes the form simply of the growth order from the kernel -- assuming, of course, that any facility can use any pre-existing facility.

An important characteristic of the facility organization is its avoidance of designer's prerogative. There is no artificial boundary in the system between what the L* designers provided, and what is provided by users (or even by a set of advanced system designer-users who might provide more tools, or by object-system implementers down the line). Although the kernel consists of a particular set of facilities, and others get added to produce the A-level system, there is no way of distinguishing such facilities from others added later except in terms of the substantive dependencies. The object-system implementer can regrow the system with alternative facilities, or replace a basic facility that underlies much else (say the Space Management facility).

An important extension of this characteristic occurs because the object systems are grown within the implementation system. Thus, it is not necessary to distinguish implementation tools from software that is seen as part of the object system (say for monitoring of the running object system). Because all of the tools reside in one environment, augmentations of the implementation system and augmentations of the object system merge to become a single activity.

4. Experience

Any proposed implementation system, certainly including L*, requires evaluation of how well it meets its design goals, and indeed of how well these design goals attain the ultimate goal of producing good software systems. Evaluation of such complex systems is not easily accomplished, a fact generally acknowledged, and we have no special miracles to make it easy. Furthermore, systems such as L* are not static but continue to adapt. Thus L* provides a moving target that tends to overcome deficiencies revealed in early evaluations. This is especially true of a language with the flexibilities of L*.

An immense number of aspects need to be assessed. How fast can software be produced with L*? How efficient are the systems so produced, in space and in time? How maintainable and modifiable? How portable? How long does it take to bring up a new L* system on a new machine? What is the performance of L* with variation in size and complexity of object system? With variations in the experience and quality of programmers? How long does it take to learn L*? As laid out in Figure 1, L* embodies many specific elements of design philosophy: their specific contribution to these evaluative dimensions must be assayed. The mechanisms in L* that realize these elements must be analyzed in their own right, since failures in L* performance may be due to imperfect mechanisms rather than inappropriate philosophy. Mechanisms will also exist in L* that do not seem to serve any stated design philosophy, and these need to be identified and their contribution (positive or negative) determined.

Recitation of this litany is not meant to overwhelm, or to let ourselves off the hook. We do believe that answers to such questions should be actively sought, both for L* and for other implementation systems. (And we believe that the lack of such data on existing implementation systems approaches the scandalous.) We provide here the few facts we currently have; the issues discussed in the next section give some indication of what our future data-gathering will focus on.

A few of the facts required for an assessment can come from measurement of static systems -- of the amounts of code and data that make up L* or L*-produced object systems. Most, however, must come from data on performance. We have endeavored to obtain some data by means of what we call software experiments (Robertson, Newell, and McCracken 1974). A software experiment involves the recording of at least certain minimal data on a actual software-producing event. That minimum includes objective times on the total effort involved and the amounts devoted to various activities, with objective measurements on the amount and type of outputs produced (usually code and data). It includes some minimal description of the programming talent involved and the computing environment within which the event occurred.

The scientific yield of such software experiments is crude indeed. Viewed from the hilltop of good experimentation, such experiments are wildly out of control. But the numbers are not thereby devoid of significance. They are infinitely superior to having no objective numbers at all, however much they need to be qualified by subsequent analysis.

4.1. General Use

L* has been in use for several years, but by a very small user community. One style of use, the original one envisioned, has been to construct one-man experimental AI programs (Freeman 1970; Newell 1972; Moore 1971). These systems range anywhere from 50K 36-bit words to the maximum capacity of the target machine, are run largely in interpretive mode, are modified repeatedly, and become highly personalized. Most such systems, of which there are probably a few hundred per year produced in the U.S., are currently produced in LISP. No hard facts are available on any of these systems. A comparison of the functional design features of LISP and L* in Figure 1 suggests that they would not differ significantly in their suitability for this task; casual observation supports this.

L* has been used for a variety of interactive systems. For instance, it has been used for a sequence of production systems called PSG (Newell 1972; Newell and McDermott 1975) and OPS (Forgy and McDermott 1976), which can be viewed as programming languages, from the viewpoint of L* application. Some of these applications have a substantial component of low-level system programming. For example, ZOG (Newell, Simon, Hayes, and Gregg 1972), a system to aquaint users naive to the PDP10 with a collection of large AI programs and to guide them in their use of these programs, interposed itself between the users and the PDP10 operating system, handling many of the command level functions for the users. A second generation ZOG (Robertson, Newell, and Ramakrishna 1977) is exploring man-machine communication issues, and is also being implemented in L*.

L* has begun to be used as an implementation system on experimental computer systems. A good example is its use on an experimental version of Hearsay-II, a speech understanding system, brought up on C.mmp. Hearsay-II is coded in SAIL on the PDP10. SAIL was not available on C.mmp, though BLISS11 was (and was the basic implementation system). The selection of L* (over BLISS11 or bringing up SAIL) rested strongly on the claim of providing a complete operating environment. C.mmp, being a one-of-a-kind experimental system just beginning its operational life, offered a lean software environment. This was a matter of great concern to the Hearsay developers. That L* would provide essentially all the tools and facilities of a complete software environment as soon as it became operational on C.mmp made it a quite attractive alternative.

Two other trials of this same type are currently in progress. L* has been made available as the system on a standalone minicomputer (with graphics) to support an experimental psychological laboratory, to be used for stimulus display and experimental control. Again, one reason for its attractiveness is the total environment it presents. L* is also to be used on a microcomputer based system (a network of Intel 8080's). Again, a strong component of the appeal is the need to obtain a friendly user software environment on a system that is experimental and one-of-a-kind, and which poses stiff barriers to obtaining that environment in the usual way through the accretion of many individual programs. Even the fact that C.mmp uses PDP11 processors, hence has access to existing PDP11 programs, only helps a little, since the imposed multiprocessor structure makes the importation of such programs non-trivial. Both these trial cases are still in an early stage.

4.2. Software Experiments

A brief description of some of the software experiments we have performed with L* will give some further indication of its use. Comments on some of the data from these experiments appears after the descriptions.

WILE. The basis of the first experiment was an experimental programming language designed by Wile (1974). The language was exploring some novel control structures, but was not implemented. Taking familiarity with Wile's study as a starting condition, we implemented it within L*. This involved design, coding , debugging and testing. We kept a record of progress along the way, the amounts of code produced, and difficulties encountered -- from bugs to design errors. The total effort was done by three expert programmers (the present authors) in a single 17 hour session, with another dozen hours of follow-up maintenance. The surface structure of the language was essentially identical to the notation used in the original study. The language was fully interactive with displays of partial computations, so that convenient exploration was possible. We sought to make the point that it was possible to create experimental languages in reasonably short order.

APR74. A version of L* (L*C.(A)) was brought up on an early version of C.mmp to provide a demonstration of real-time speech signal acquisition, segmentation, labelling and display for an IEEE Speech Recognition Conference (CMU Speech Group 1974). The L* software experiment involved creating the L* system on C.mmp, providing the operating system and multiprocessing features necessary for the standalone application, and integrating the pieces into a running system. The speech programs were coded in BLISS11, so this involved embedding BLISS11 into L*. C.mmp was at a very early stage of development at the time, so the environment was extremely raw in terms of reliability and software facility. The entire experiment took 30 calendar days against the hard deadline of the conference, and produced a running system that was demonstrable but imperfect (it ran fully the next day). The point of the L* experiment was that total system facility was made available on a raw machine, with extreme flexibility to meet the unexpected demands of such a complex system programming situation.

SOS. The standard line editor for the PDP10 is called SOS. An L* software experiment was performed to produce a version for C.mmp. This was a slightly limited version (no justification or contextual searching commands), but was to be totally to specifications, since it was intended for production use by programmers who use SOS daily on the PDP10. The experiment took 30 man days.

L*ALTO and L*8080. Two software experiments were performed to test the transportability of L*. The first was the construction of an L* for the ALTO, a 16 bit minicomputer. This took 70 man days to complete (for one man). The task involved some graphics support and microcode support not found in other L* systems. The second was the construction of an L* for the Intel 8080, an 8 bit microcomputer. L*8080 is much closer to the other L* systems, and took 28 man days (for one man).

Most of our experimental data concerns programmer productivity when working with L*. As noted, this is not the only important quantitative measure of the worth of an implementation system, but it is a critical one.

Figure 8 gives several productivity measurements for L*, with a few measurements from the literature (Wolverton 1974) for calibration. For the software experiments mentioned above, the numbers are accurate, since we have precise code counts and numbers of hours worked. For the other L* situations, the code counts are accurate, but the time estimates are somewhat less exact. For the measurements from the literature, the information does not seem very reliable.

The measure used is number of debugged instructions per man-day, where a day is taken as an 8 hour period (working round the clock produces 3 man-days per calendar day, as in the WILE experiment). The instruction count is taken on the final system at the end of the period. The number of instructions is measured by the code and associated data in the computer, it is not measured in the source language. Counting associated data (not input data) properly handles some program/data tradeoffs, but care must be taken with programs that use large simple tables. This puts all languages on one common footing, but leaves open the relation of source language to ultimate machine code size. It is well known that poor compilers produce a larger code+data size than do optimizing compilers, thus making the poor ones appear more productive. This has to be handled through additional measurements of these ratios. With L*, each symbol resides in one cell, so that in effect one can count instructions in the memory by counting symbols (code+data) in the listing.

System	Language	Size	Instructions/man/day
L*(I)	Assembly + L*	30K	375
SOS-C.mmp	L*	5K	310
Industry average - very simple task			167
L*ALTO	Assembly + L* + BCPL	9K	136
L*8080	Assembly + L*	3K	107
WILE	L*	0.8K	96
APR74	L* + Bliss11 + Assembly	3.7K	51
Industry average - simple task			50
L*	Assembly + L*	50K	30
SL*	L*	2K	25
Hearsay-C.	L*	5K	21
Industry average - moderately complex			16.7
Industry average - complex			8.3

Figure 8. Productivity data for L*

There are several important points to notice about the L* data. The programmers were all experienced programmers, but not all were experienced in the use of L* (particularly in the case of SL* and Hearsay-C.). These figures are all based only on new code generated (e.g., L*8080 is an 18K system, but only 3K of it had to be written from scratch).

With all the caveats stated, the productivity numbers are very high. Some things can be said about specific numbers. The SOS numbers are possibly high because of the peculiar decomposition of an editor (i.e., a large number of commands, each with an isolated bit of code). The relatively low figures for Hearsay-C. almost surely express programmer differences, plus losses inherent in the transfer of the incomplete system from one programmer to another. This data suggests that programming with L* increases programmer productivity. More software experiments are needed to further substantiate that claim.

5. Issues

Not all that can be said about L* is positive. We have faced up to many problems throughout the long series of design iterations; but there still remain serious unsolved problems at the frontiers, some of which have surfaced only with extended experience. We will now discuss several of the negative issues, and at the same time take the opportunity to draw some contrasts between L* and two specific alternative systems: BLISS and LISP.

5.1. Sufficient efficiency

Almost without exception, efficiency (both of time and space) is a prime requirement for object systems. Even one-of-a-kind experimental systems, which will never reach wide-spread usage, cannot ignore efficiency altogether. Thus, it is undeniable that L* must permit flexibility to be cashed in for efficiency whenever it becomes appropriate, and to whatever degree is necessary. If an L* object system cannot match closely the performance of an equivalent BLISS system[18], then all the advantages of L* may largely go by the boards. On the other hand, very high efficiency is not always necessary, especially during the period a system is undergoing development and experimentation. Thus implementation systems which are able to delay the optimization until it becomes crucial (i.e., systems such as LISP, ECL and L*) can enjoy the benefits of flexibility when they are most needed.

A Time.accounting Facility exists in L* to help find the bottlenecks in a running system. Once they have been located, it is usually possible to obtain some initial speedup merely by minor reorganization and recoding. There are several documented cases where large factors (e.g., ten to twenty) were obtained in subcomponents of a system in this manner.[19]

A Translation Facility in L* compiles PL* programs into ML* code, which essentially eliminates the type-access and PL* interpreter cycle for each symbol in the program.[20] A Cycle.accounting Facility enables one to monitor a running system to obtain a list of every PL* routine that was called, ranked by the total number of interpreter cycles spent just inside each. A typical result might show twenty routines above the 1% rank (i.e., each of the twenty claimed more than 1% of the total interpreter cycles) and these routines would then be an appropriate choice for

[18] With BLISS the efficiency is premeditated, and thus virtually ensured.

[19] Most often the recoding takes the form of specializing the use of an operation which was overly general for the particular case.

[20] To maintain some flexibility, the PL* version of a compiled program is saved so that it may later be edited and recompiled if necessary.

compilation. The cycle-counting and compilation process can then be repeated, but with significantly diminished gain since the distribution flattens out quickly. Since an uncompiled L* system typically spends half to two-thirds of its time inside the PL* interpreter, speedup factors of up to two or three can be realized by compilation.

A typical next phase in the optimization process is the hand-coding in ML* of certain critical routines. (Again, the Time.accounting facility is a valuable aid to intuition for identifying the critical routines). A completely integrated Assembly facility within L* allows the ML* code to be loaded and interfaced with other ML* and PL* routines very conveniently. Hand-coding can achieve significant improvements by removing the overhead of calls to small subroutines (both in terms of control and argument-passing on the Z stack), and through more extensive use of the machine registers for temporary storage. Speedup factors of two to three can be expected for typical routines.

One final optimization step is microcoding, although it is possible on only two existing L* systems, and has actually been accomplished on only one of them (L*ALTO). The microcoded version of L*ALTO used 900 words of microcode to recode 51 L* kernel routines (e.g., the PL* interpreter, type access, stack operations), and achieved a factor of 3.2 speedup over the non-microcoded version.

Although the approach to selective optimization just outlined seems right to us, the mechanisms to support that approach have never been fully developed. Also, there are areas where the approach breaks down; most notably, when dealing with the tradeoff between generality and efficiency. The most often cited case is the file reading mechanism in L*, which is about six times slower than the corresponding mechanism in LISP. About half of that difference is due to code which is interpreted in L* versus hand written machine code in LISP. That part of the difference can be approached with selective optimization. However, the other half of the difference is due to the extensible nature of the L* mechanism, and cannot be easily removed by selective optimization. Whether or not the selective optimization approach will succeed in L* is still an open question.

5.2. Contraction

Although contraction could have been discussed in the previous subsection (i.e., contraction = selective space efficiency), it is important enough to rate a subsection of its own. In fact, its importance is greatly magnified for L* due to the integration of implementation system and object system. Without a contraction capability, an L* object system would of necessity contain all the implementation tools used to construct it -- a quite unworkable situation.

One common paradigm for contraction is to somehow mark the unwanted structures (e.g., in L* by erasing them -- putting them on the available space list) and then relocate structures within the address space in such a way that a strict partition is created, with all unwanted structures at the high end of the address space. It is then normally a simple matter to eliminate the unwanted structures by chopping off the top of the space. However, this approach has seemed to be barred for L* due to the apparent impossibility of building a foolproof structure relocator. With the great flexibility and accessibility that an L* user exercises, it is all too easy to imagine ways in which a relocating program could be inadvertantly fooled in its memory search for references to a symbol.[21]

It is easy enough in L* to erase structures that are no longer needed, but the resulting available space will almost surely be scattered throughout memory; thus the system will have a greater capacity for growth within its current memory size, but it cannot be truly contracted without relocation.

In L*(I) we have developed a partial solution to the contraction problem by taking advantage of a feature of the DEC TOPS-10 monitor which allows independent control over the two halves of the users 256K address space (called the low and high segments). Within L*(I), available space lists are maintained independently for the two segments so that the user may control the segment in which new structures are to be created. By convention, the high segment is used for utilities (e.g., the editor, debugging tools, compiler) and other facilities that are expected to be used relatively infrequently. This means that the system can normally run with just the low segment, and only load the high segment (which is read into memory from a drum) when access is needed to the facilities there.[22]

This scheme is reasonably effective, allowing the A-level L*(I) system to be contracted from a total size of 35K to a low segment size of 25K with no noticeable degradation in response due to high segment swapping. The main difficulties with the scheme are: (1) the choice of the segment in which to place a structure must be made in advance, and is then for all practical purposes fixed; and (2) knowledge about cross-segment references must be explicitly represented, and thus bugs caused by references to an unloaded high segment are a problem. Given a better underlying operating system, this basic idea could be more effectively exploited.

5.3. Higher order language

Some user feedback has suggested that L* is a low or medium level language. In fact, there are features found in most high level languages which are not found in L*, and vice-versa. In this section, we will examine those differences. It should be pointed out that L* is a high level language in terms of expressive power (i.e., the size of an L* program to perform a particular task is as small as a program written in other high level languages to perform the same task).

---

[21] This same technical difficulty is probably responsible for the fact that no garbage-collection facility has yet been built for L*.

[22] Loading of the high segment is not automatic -- thus the system must anticipate all accesses to the high segment with an explicit load operation.

The most obvious difference between L* and other high level languages is that L* is neither an algebraic nor an expression language. In an algebraic or expression language, what one writes are expressions which evaluate to a localized value (except in the case of side effects, which are rare). These languages are also phrase structured (i.e., a symbol may be replaced by an expression) for operands in general, while most are not phrase structured for operators. PL* does permit phrase structuring of both operands and operators. However, values are not localized and are not the result of evaluation of expressions. Instead, values are generally placed in the central data stack.

A second difference is that L* has no variables. This came about from a conscious choice to always deal with symbols, and not their values. There are mechanisms in L* that allow variables to be easily implemented, but variables are not part of the basic system.

Third, L* does not provide symbol contexts (i.e., there are no local variables, everything is global). L* does provide name contexts which can be used to gain the same kind of protection that local variables offer, but at the name level instead of the symbol level.

Fourth, L* is postfix, while most languages provide infix for arithmetic and boolean expressions. This is not considered a major problem, however, since infix syntax can be easily constructed in L* (a one page program can provide infix and function notation).

And finally, L* uses the machine as the model instead of external expressions. This was necessary in L* to provide the kind of accessibility to the underlying machine that we desired. Because of it, L* may be thought of as a high order machine language, although its expressive power argues that it is more than just that.

5.4. Habitability

One of the negative aspects of our experience with L* has been the slowness with which its usage has spread, even within the local environment. Some of this is no doubt due to our casual attitude toward promotion, but there is nevertheless an accumulating body of experience with new L* users that strongly suggests a habitability problem with L*. While we do not yet have a good understanding of the reasons for this problem, we have formed a few plausible conjectures with the help of user feedback.

The first conjecture is that total accessibility produces apparent complexity from the relative simplicity of the L* system by exposing large masses of detail which are irrelevant to a beginning user. An A-level L* system has around 900 names (which is significantly more than a new LISP or BLISS user sees; e.g., LISP 1.5 presents about 150 names to a new user), with no good way to differentiate the relevant from the (for the time being) irrelevant. One possible simple solution (but one which we have not yet tried) would be to introduce into the system a discrimination of the small subset of essential names from the great masses. For example, two separate name contexts might be used to provide the discrimination. This approach looks promising since users tend to operate most effectively in small worlds that they know and control.

Another conjecture for the cause of the habitability problem is the use of concise names. Short names (e.g., S, N, I, and D for the common list operations) have the advantage of economy of expression, which is especially important for a highly interactive system. However, feedback from new L* users indicates that the short names form a real stumbling block to learning the language. This problem is aggravated by the failure to produce a self-documenting system (a feature explored in L*(H) but rejected because of high space costs). A considerable amount of iteration has gone into the design of the names, which are based on a set of standard abbreviations with fairly consistent conventions. Users report that after they learn this set of conventions, the short names do provide the intended economy of expression.

5.5. Symbols and addresses

As discussed earlier, one of the essential design principles of L* is the universal symbol system. Although that principle has never been in question, the mechanism of equating symbols and addresses to achieve a universal symbol system has. As noted earlier, this mechanism has distinct advantages. It assures that all addresses will be valid in symbolic expressions. It also simplifies the design by equating address arithmetic and symbol arithmetic.

However, equating symbols and addresses has serious disadvantages. First, it limits the number of symbols to the address space of the underlying machine. With most hardware architecture, this poses no problem. But if the address space is smaller than the physical memory and some kind of relocation or overlay mechanism is available to take advantage of that additional memory (as on C.mmp), then the limit becomes real and serious.

Second, binding symbols to addresses makes reassignment of symbols difficult. If all references to a symbol cannot be located, then the symbol cannot be moved; it must remain as a place holder and pointer to the new symbol. This problem, combined with the fact that symbols can and are stored in arbitrary structures, makes general relocation essentially impossible. This in turn makes contraction and automatic garbage collection difficult to implement.

Finally, along the same lines, reassignment of the type of a symbol is more difficult if symbols are equated with addresses. If symbols were separate structures, their type would be part of that structure and therefore easily changed. The only way to achieve the same flexibility with types in the current system is to use half of the memory to hold types for the other half (a scheme that was actually used in an early version of L*).

Creating a new variant of L* in which symbols are not identified with addresses seems clearly indicated. However, it is unclear where the balance of tradeoff will lie when such a system is completed.

6. Conclusions

In this paper, we have discussed the design, implementation, and some experience with L*. The key difference between L* and other implementation systems (like BLISS or ECL) is that L* is an interactive symbol manipulation system (features shared with LISP). It also provides a total operating environment (like LISP) which has all the necessary tools for program development from within (e.g., editors and debuggers). The system also stresses selective rather than global optimization. The key differences between L* and LISP are that L* is highly extensible and allows total accessibility.

The key mechanisms used to implement L* include a universal symbol and type system, a simple interpreted list processing language, a flexible external language, and a kernel approach to building the system.

The experience to date has been with a small user community and has been generally positive. A series of software experiments has been initiated, and the data gathered suggests that programmer productivity is high. Areas of continuing concern include time and space efficiency, the low level nature of parts of the system, and the difficulty new users experience in learning to use the system. More software experiments are needed to analyze and correct these problems. The only design principle that has been brought into serious question by the experience has been total accessibility. Although total accessibility has been used to advantage in a number of software projects, it seems to be a real contributor to the habitability problem. Several mechanisms used to implement the L* philosophy have been brought into question by our experience. Perhaps the most critical is the realization of a universal symbol system by equating symbols and addresses. On the whole, however, the experience has indicated that the basic design philosophy is an interesting and viable alternative to that of more traditional implementation systems.

References

CMU Computer Science Speech Group, "Working Papers in Speech Recognition", Carnegie-Mellon University Technical Report, 1974.

A. Evans, "An ALGOL 60 Compiler", in R. Goodman (Ed.), Annual Review of Automatic Programming, v4, Pergamon, 1964, pp. 87-124.

C. Forgy and J. McDermott, "OPS Reference Manual", Carnegie-Mellon University Technical Report, 1976.

P. Freeman, "Sourcebook for OSD - An Operating System Designer", Ph.D. thesis, Carnegie-Mellon University, 1970.

P. Freeman, "Software Systems Principles", Science Research Associates, Chicago, 1975.

M. Halpern, "XPOP: A Meta-Language without Metaphysics", Proc. FJCC, vol. 26, part 1, 1964, pp. 57-68.

K. Knowlton, "A Programmer's Description of L6", Comm. ACM, August 1966.

R. Krutar, "Flexors (Modification Mechanisms)", Ph.D. thesis, Computer Science Dept., Carnegie-Mellon Univ., 1976.

J. McCarthy, P. Abrahams, D. Edwards, T. Hart, and M. Levin, "LISP 1.5 Programmer's Manual", MIT Press, Cambridge, 1962.

J. Moore, "The Design and Evaluation of a Knowledge Net for MERLIN", Ph.D. thesis, Carnegie-Mellon University, 1971.

A. Newell, "A Theoretical Exploration of Mechanisms for Coding the Stimulus", in A. Melton and E. Martin (Eds.), Coding Processes in Human Memory, Winston, Washington D.C., 1972.

A. Newell, J. Earley, and F. Haney, "*1 Manual", Carnegie Institute of Technology Technical Report, 1967.

A. Newell and P. Freeman, "BIP: Basic Interface Package", unpublished working paper, 1968.

A. Newell, P. Freeman, D. McCracken, and G. Robertson, "The Kernel Approach to Building Software Systems", 1970-71 Computer Science Research Review, Carnegie-Mellon University.

A. Newell and J. McDermott, "PSG Manual", Carnegie-Mellon University Technical Report, 1975.

A. Newell and J. Shaw, "Programming the Logic Theory Machine", Proc. Western Joint Computer Conference, IRE (now IEEE), 1957, pp. 230-240.

A. Newell, H. Simon, R. Hayes, and L. Gregg, "Report on a Workshop in New Techniques in Cognitive Research", Carnegie-Mellon University Technical Report, 1972.

A. Newell, F. Tonge, E. Feigenbaum, B. Green, and G. Mealy, "Information Processing Language-V Manual", Prentice Hall, 1964.

D. Parnas, "On the Criteria to be used in Decomposing Systems into Modules", Comm. ACM, vol. 15, no. 12, December 1972.

A. Perlis, J. Mitchell, and H. VanZoeren, "LC^2: A Language for Conversational Computing", Interactive Systems for Experimental Applied Mathematics, Proceedings of the ACM Symposium, M. Klerer and J. Reinfeld (eds.), Academic Press, 1968.

M. Richards, "BCPL", Proc. SJCC, vol. 34, 1969, p. 577.

G. Robertson, A. Newell, and D. McCracken, "On Doing Software Experiments", 1973-74 Computer Science Research Review, Carnegie-Mellon University.

G. Robertson, A. Newell, and K. Ramakrishna, "ZOG: A Man-Machine Communication Philosophy", Carnegie-Mellon University Technical Report, 1977.

K. Samelson and F. Bauer, "Sequential Formula Translation", Comm. ACM, vol. 3, no. 2, 1960, pp. 76-82.

J. Shaw, "JOSS: A Designer's View of an Experimental On-Line Computing System", Proc. SJCC, vol. 26, 1965, p. 455.

B. Wegbreit, "The ECL Programming System", Proc. FJCC, vol. 39, 1971, pp. 253-262.

D. Wile, "A Generative, Nested Sequential Basis for General Purpose Programming Languages", Ph.D. thesis, Carnegie-Mellon University, 1974.

R. Wolverton, "The Cost of Developing Large-Scale Software", IEEE Transactions on Computers, vol. c-23, no. 6, June 1974.

W. Wulf, D. Russel, and A. Habermann, "BLISS: A Language for Systems Programming", Comm. ACM, vol. 14, no. 12, December 1971.

W. Wulf and C.G. Bell, "C.mmp: A Multi-Mini-Processor", Proc. FJCC, 1972, pp. 765-778.

W. Wulf, "ALPHARD: Toward a Language to Support Structured Programs", Carnegie-Mellon University Technical Report, 1974.

W. Wulf, R. Johnsson, C. Weinstock, S. Hobbs, and C. Geshchke, "The Design of an Optimizing Compiler", American Elsevier, NY, 1975.