L* Introductory User's Manual

Preliminary Draft

Brian K. Reid
November 10, 1975

→→→→	This is a PRELIMINARY DRAFT	←←←←
→→→→	It is incomplete and unproofread	←←←←
→→→→	and distributed cautiously	←←←←
→→→→	Please report errors constructively	←←←←

This work was supported in part by the Advanced Research Projects Agency of the Office of the Secretary of Defense under contract F44620-73-C-0074, monitored by the Air Force Office of Scientific Research.

Part I: The Nature of L*

1. Introduction 1

1.1. The history of L* 1
1.2. Further information 2
1.2.1. Reference documentation 2
1.2.2. Exploring on your own 2

2. L* design Philosophy 3

2.1. General design principles 3
2.2. Facility-specific design principles 4

Part II: The Structure of L*

3. A system overview 4

3.1. L* in perspective: a comparison with some other systems 4
3.2. Terminology and notation 5
3.2.1. Notation conventions 5
3.2.1.1. Bracket notation to describe routines 5
3.2.1.2. Parenthesis notation for symbols on Z 6
3.2.2. Words with a rigorous meaning in L* 6
3.2.2.1. Symbols 6
3.2.2.2. Names 6
3.2.2.3. Name Contexts 6
3.2.2.4. NIL 7
3.2.2.5. Level 7
3.2.2.6. Signal 7
3.3. Major system pieces 7
3.3.1. The central stacks Z and ZX 8
3.3.2. The type system and the storage manager 8
3.3.3. Interpreters and program execution 9
3.3.4. User interface, recognition, debugging, etc. 9

4. Structures: L* building blocks 11

4.1. Defining a type and a structure 11
4.2. Types and structures supported by the basic system 11
4.2.1. Programs 11
4.2.2. Lists 12
4.2.3. Stacks 13
4.2.4. Words and blocks 13
4.2.5. Associations and association lists 13
4.2.6. Character strings 14

5. Program execution and environment 16

5.1. The interpreters 16
5.1.1. Finding the interpreter for a type 16
5.2. Error detection and recovery 17
5.2.1. Error checking 17
5.2.2. Error events 17
5.2.3. Continuing from an error 17

6. The user interface 19

6.1. The EXEC 19
6.1.1. Character actions and name assembly 19
6.1.2. Recognition 20
6.1.3. Recognition actions 21
6.1.4. Name contexts and context lists 22
6.2. Building program and data structures 23
6.3. Reading files 23

Part III: Programming in L*

7. Building and printing structures 25

7.1. Building programs 25
7.1.1. Creating program lists 25
7.1.2. Creating machine-code programs 27
7.1.3. Editing list structures 27
7.2. Building non-program structures 28
7.2.1. Data lists 28
7.2.2. Associations and association lists 29
7.2.3. Word structures and blocks 30

7.3. Storage management 3
7.3.1. Word storage management 3
7.3.2. Block storage management 3
7.4. Printing 3
7.4.1. Printing whole structures 3
7.4.2. Printing symbols by type 3
7.4.3. Printing numbers and addresses 3

8. Program control 3

8.1. Conditionals 3
8.1.1. Conditional operators 3
8.1.2. Signals and the signal stack 3
8.2. Iteration 3
8.2.1. Internal iteration 3
8.2.2. External iteration 3
8.3. Special control operators 3
8.3.1. Symbol quote operators 3
8.3.2. Execute control actions 3
8.3.3. Infix operator control 3
8.4. Address space control (overlays) 4

9. Data manipulation Operators 4

9.1. Operations on the central stacks 4
9.1.1. Operations on Z 4
9.1.2. Operations on Z$ 4
9.1.3. Operations on the scratch stacks 4
9.2. List facility 4
9.2.1. Creating, copying, and erasing lists 4
9.2.1.1. Creating lists 4
9.2.1.2. Copying lists 4
9.2.1.3. Erasing lists 4
9.2.2. Examining and modifying list structures 4
9.2.2.1. Examining and searching lists 4
9.2.2.2. Modifying lists 4
9.3. Stack facility 4
9.4. Association and Association List facility 4
9.4.1. Association-list manipulation 5
9.4.2. Association structure manipulation 5
9.5. Word facility 5
9.6. Character-string facility 5
9.7. Block structures 5

10. The Recognition System 56

10.1. The EXEC in detail 56
10.1.1. Character actions and name assembly 56
10.1.2. Recognition actions 56
10.1.3. Modifying the user interface 56
10.2. Names and contexts 56
10.2.1. Creating and using a context 56
10.2.2. Defining and redefining basic-system names 56
10.2.3. Local names 56
10.2.4. Names within blocks 56
10.3. Creation type control 56
10.3.1. Type-control operators 56
10.3.2. The forward-reference problem 56
10.4. Pertinent data structures in the recognition system 57

11. Operating-system interface 57

11.1. Reading and writing files 57
11.1.1. I/O interfaces 57
11.1.2. Character stream I/O 57
11.1.3. Binary word I/O 57
11.2. Time accounting 57
11.3. Sub-jobs 57
11.4. Parallel processing 57
11.5. Odds and ends 57
11.5.1. PDP-10 notes 57
11.5.2. C.mmp notes 58

12. Debugging tools 58

12.1. Breakpoints 58
12.2. Tracing execution 58
12.3. Error detection and recovery 59

Part IV: Practical L*: hints, tools, and techniques

13. Programming Style 59

13.1. Source program formatting 59
13.2. Names 59
13.3. Technique 60

14. Debugging technique 60

14.1. Finding bugs: some hints and pitfalls 60
14.2. Bug fixing and patching 60
14.2.1. How to correct it 60
14.2.2. How to maintain an up-to-date source file 60
14.3. When to give up and recompile 61

15. A program library: existing code 61

15.1. dummy section 62

INDEX 62

Part I: The Nature of L*

1. Introduction

L* (pronounced 'L star') is a complete system for constructing, running, and debugging software. Its emphasis is on rapid programming and flexibility. L* was developed over the past several years by Allen Newell, Don McCracken, George Robertson, and others. The current version of L* on the PDP-10 is called L*(I), and the current version of L* on C.mmp is called L*C.(C).

Other implementations of L* are similar to L*(I) and L*C.(C), but not sufficiently similar that their users may use this document in its entirety. We will not attempt to enumerate all differences between current L* implementations and previous ones, but in places where that information is either critical or interesting, some older implementations will be discussed.

1.1. The history of L*

L* has evolved from attempts to build a successor to the IPL-V system. In 1968, IPL-V was an old language, and there were certain specific features that were not considered satisfactory. Motivation to design and implement a new system was created by CMU's switch to a new machine, a PDP-10, from a 360/67.

By Fall of 1968, the central concepts which begat L* had been worked out by Allen Newell, and documented in private working papers. These plans called for a 'system of list languages' called L*. The name L* comes from the Kleene star: the closure of all list languages derivable from a basis kernel. (Recall L6: Low Level Linked List Language).

The first incomplete specification for an implementation of an L* system was L*(A), completed in October of 1968. By July of 1969 the written specifications for L*(A) were complete, and plans for an implementation were begun by Allen Newell, Peter Freeman, and Don McCracken. With George Robertson joining the L* group, the design went through several more iterations; finally in June of 1970 an implementation was completed of L*(D). This implementation was on the PDP-10.

A new implementation and two more design iterations were soon completed; L*(F) became available in November of 1970. This version, L*(F), was used to build a large user system, MERLIN (Moore and Newell).

Several more implementations evolved on several more machines. Currently, the PDP-10 version is L*(I), and the C.mmp version is L*C.(C). A version for stand-alone PDP-11's, L*11, is similar to L*(H). Based on L*11, a version of L* for the Computer Modules project should be available by the first part of 1976.

1.2. Further information

1.2.1 Reference documentation

The ultimate reference documentation for any L* system is the source file for that system. This source file is called an M-FILE. The M-FILE for L*(I) can be found as LSIM18[A110LI00]. There does not yet (November 10, 1975) exist an M-FILE for L*C.(C), but there is a binder in the C.mmp room which contains an up-to-date listing of the L*C.(C) system.

There is a tutorial script in L*(I), which is designed to serve as an online 'programmed instruction' tutorial. It is fairly slow during typical load conditions on the PDP-10; an interested user might consider trying the script during off-peak hours. To get at the script, type "R LSIA" to the monitor (which will get you into L*), then type "HELP" to L*.

When all else fails, ask George Robertson.

1.2.2 Exploring on your own

Rather than hunting through reference documentation or asking the experts, users are encouraged to explore L* from within. It is a totally accessible system. Every address in the system is accessible; thus every piece of code and every data structure is available and visible There is a print facility which allows one to print anything in the L* system in a reasonably readable format. There is a stepping monitor which will trace the execution of any routine. If you want to know how something works, go look at it. The print facility, for printing out programs, is described in section 7.4. The stepping monitor, which will allow you to watch the execution of any routine, is described in section 12.2. There should be a sufficient number of data-structure names explained in this manual to get you started; they are all alphabetized in the index.

2. L* design Philosophy

L* is a highly principled system. By this we mean that it has evolved from a systematic exploration of the design alternatives available, with decisions being made on the basis of rigorous principles of design. The design process has taken the form of a long series of iterations, taking seven years, which can be thought of as converging on the ultimate target system.

It is expected that the L* user's programming philosophy will be similar to the designers', which helps to blur the distinction between system implementer and system user.

The design principles that shaped L* could be enumerated like so many commandments on a tablet, and be read as a sort of catechism: almost all design principles look sound when stated reverently enough. However, the implementors of L* take their design principles seriously, and consider that updating of the design principles to suit new discoveries or new ideas is just as important as updating the documentation to suit changes to the system.

So let us enumerate the design principles, with a minimal amount of commentary on each.

2.1. General design principles

2.2. Facility-specific design principles

Part II: The Structure of L*

3. A system overview

3.1. L* in perspective: a comparison with some other systems

L* is a programming system; that is, a means for building and executing programs, and an environment in which to debug and modify them. Many "programming systems" exist today, and they tend to have only fleeting similarities. Since L* is an unusual system, it would be worthwhile to try to define it with respect to other available systems.

All programming systems intend to provide a means whereby a programmer can direct the execution of a computer. An assembler allows the programmer to generate machine instructions more or less by hand. An assembler and a loader together form a system for generating and executing machine code, so they may together rightfully be called a programming system. Often, however, there exist libraries of routines which the programmer may add to his program with a link editor, and there may exist core-dump programs, macrogenerators, editors, debuggers, and so forth. Taken in toto, these pieces form a programming system. Further, they form a complete programming system, since there is no programming task which cannot be programmed with this programming system.

APL is also a programming system, but unlike the assembler-linker-loader system described above, it is self-contained: the APL system is a program which, totally within itself, allows a programmer to enter, display, execute, edit, and debug programs. Different implementations of APL handle the details in different ways: some are strictly interpretive, others compile code for a statement at a time, then execute the machine code so generated. Nevertheless, all this activity takes place within APL. APL is a restricted system, in that not all programming tasks are suitable for APL, or even possible within APL. For example, it would not be possible to write a special magnetic-tape copy routine with APL, since the APL system does not provide any primitive functions for the manipulation of magtape, nor does it provide any mechanism for adding new primitive functions.

L* is a self-contained system somewhat like APL, but it is not restricted: a system or program of any sort may be constructed in L*. Not only is it not restricted, it is arbitrarily extensible and modifiable: the L* user may add new pieces to the system, modify or remove existing pieces, or build up from the "basic system" without actually modifying it. L* is thus a complete system: any programming task which is suitable for the computer in question may be programmed in L*. In calling APL a programming system (which it is), we are in a sense obligating ourselves to think of a different term for programming systems in

which other systems may be built. For example, it would be entirely possible to implement APL in L*, but it would not be possible to implement L* in APL. Hence, L* is frequently referred to as a 'system-building system', in order to stress its differences from other programming systems.

3.2. Terminology and notation

Every system has its own terminology which is used to describe and define the system. Terminology unique to L* is used both inside the L* system (in the source file), and in external documentation. The reader should understand the specific meaning of all of the L* 'system terms', and should keep them in mind while reading any L* documentation.

3.2.1 Notation conventions

Throughout L* documentation, and to a large extent throughout existing L* code, various notational shorthand is used to describe the behavior of L* routines.

3.2.1.1 Bracket notation to describe routines

Often a routine will have a descriptor in square brackets, as for example '[11]' or '[20]'. This notation summarizes the inputs and outputs of the routine. It works as follows: The full form of the notation is

```
[AB$CD]E
```

where

A	is the number of inputs from Z
B	is the number of outputs to Z
$	is present if the routine returns a signal
C	is the number of inputs from Z$
D	is the number of outputs to Z$
E	is a single-quote if the routine is active

Very few routines ever change the contents of Z$, so the 'C' and 'D' fields are hardly ever used. If a routine takes a variable number of inputs, or leaves a variable number of outputs, then the letter V is used instead of a number.

Here are some samples of this notation:

P	[12]	Push: 1 input, 2 outputs
U	[10]	Up: 1 input, no outputs
F	[1V$]	Find: 1 input, variable output, sets signal
U$	[00$10]	Up Z$: no Z in or out; pops Z$
?	[10]'	Print: 1 input, no outputs, active
(	[01]'	Left paren: no input, 1 output, active

3.2.1.2 Parenthesis notation for symbols on Z

When a multi-input or multi-output routine is being described, it is clumsy to say 'the symbol on top of Z' or 'the symbol third from the top of Z'. L* documentation uses a notation with a number in parentheses: (0) means the top of Z, (1) means the next-to-the-top of Z, etc.

3.2.2 Words with a rigorous meaning in L*

3.2.2.1 Symbols

A program's physical existence is implemented in terms of words of computer memory. Since 'word' is a fairly ambiguous term, many technical terms have been coined through the years to describe things built out of chunks of computer memory. At various times, the words cell, atom, token, block, word, structure, packet, record, array, and many more, have all been used to describe something built out of contiguous pieces of computer memory. All of these words have a history and certain prior connotations. The L* implementers chose to use some and reject others.

The word Symbol has two meanings in L*. Primarily, a symbol is an address. Thus, when we speak of 'the value of a symbol' or 'printing a symbol', the word means address. Symbol is also the name of one of the fields of a list cell. One will often see the expression 'taking the symbol of' something. This does not mean 'finding the address of'. Taking a symbol means extracting the symbol field of a list cell. The contents of this symbol field is in fact a symbol in the first sense, but then so is the contents of the 'next' field.

3.2.2.2 Names

A name is a character string used to identify a symbol. The name of a symbol is available for labeling output, and the name of a symbol is used by the L* recognition system to recognize references to symbols. Not all symbols have names, though all symbols can be given names.

3.2.2.3 Name Contexts

A Context is a name table. Frequently one hears the expession 'symbol table', but since we have pre-empted the meaning of the word symbol, we cannot unambiguously speak of a symbol table in L*. So we call it a name context. A name context is a table associating a character-string external name with a symbol (i.e. with an address). There can be many name contexts; the L* basic system provides two: BCX, the base context, to hold system names, and UCX, the user context, to hold user names. Additional contexts may be created and used as needed.

3.2.2.4 NIL

NIL is a particular L* symbol of type list. By convention, a list is terminated when its next field is NIL; since NIL is a list, its own 'next' field must also be NIL. NIL is also used to represent the value 'false' in conditionals. NIL has no intrinsic special properties other than in the way it is used by the rest of the system.

3.2.2.5 Level

A level of a list or program is a sublist or subprogram. Consider the list (A B C (D (E) F) G H). It has three levels. The outermost or highest level is (A B C something G H). The next lower level is (D something F). The lowest or innermost level is (E). This is easier to visualize if you think of a 2-dimensional representation of a list:

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

The term level is often seen in documentation of the editor, the interpreter, and the stepping monitor, and in documentation of control actions.

3.2.2.6 Signal

The word signal in L* documentation refers to a true or false value returned by a routine. At any given time, the current signal is the value at the top of the stack Z$. A value of NIL means false; any other value means true. Many control actions test the signal.

3.3. Major system pieces

There are certain major pieces of L* which a reader must understand before L* documentation will make much sense. They include:

> The central stacks
> The type system and associated storage manager
> The interpreter and the program control mechanisms
> The recognition and printing system

Each of these pieces will be described in detail in Part III ('Programming L*'). We will provide a short description here in order that the pieces and the names not be a myster as the documentation unfolds.

3.3.1 The central stacks Z and ZX

The L* main data stack is named Z, and the main control stack is named ZX. Z is used for almost all argument passing, computation, and temporary storage inside routines. Most L* operators work on the top one or two elements of Z. For example, the +W operator, which is the addition operator for Type Word, takes as input the top two elements of the stack, and emits as output their sum, which it pushes onto Z. This behavior is quite similar to that of the '+' key on a postfix calculator (with which we assume all of our readers are familiar).

The control stack ZX is used to record linkage information for routine calls. It is used in much the same way as an ALGOL runtime stack, with one important exception: ALGOL stores local variables and arguments on the runtime stack, which L* does not. See section 5.1 for a more complete discussion of ZX and its contents. L* supports stacks as a data type; Z and ZX are two specific stacks. ZX, however, is somewhat special: it is not implemented via the general L* stack mechanism, rather it is a machine register. For this reason, attempts to print ZX using the normal print routines described in section 7.4 will fail, since ZX has non-standard structure. The L* basic system provides a routine for printing ZX; its name is ?ZX. ?ZX is an active symbol (see 6.1.3), which means that it is executed immediately when it is typed.

3.3.2 The type system and the storage manager

Every address known to L* has a type. The type system is crucial to L*, driving the interpreters, the user creation and print routines, and the storage manager. Storage allocation is organized by type. When a program needs storage, it calls the storage allocator giving a type and a word count. The storage allocator will return (on Z, of course) the address of a block of storage of the requested type.

An L* user does not normally acquire storage by direct calls to the storage allocator. Instead, there exist type creation routines which allocate a chunk of storage of the correct size, and initialize its contents. For example, one seldom wants to create a block of storage of type stack, rather one wants to create a stack. Instead of calling the storage allocator to get some words of type stack, one calls the stack creation routine, which in turn calls the storage allocator to get the storage that it needs. The individual creation routines and their behavior are described in chapter 7.

L* does not use a garbage-collection scheme; storage must be explicitly erased. Although this scheme requires more attention from the programmer than would a garbage collection scheme, it is much simpler, much simpler to implement, and inherently much faster. When storage is erased, it is returned to the available space list for its type.

Storage is organized in units of type blocks of 128 words each. In L*C.(C) storage is further organized in units of pages (Hydra pages) of 4096 words each. Each type block has a type which is determined by a type map, and all storage allocated in that block has the type of the block. Unallocated storage is of type available space.

3.3.3 Interpreters and program execution

L* is an interpreter-based system. Routines are called by the interpreter, and when they finish execution, they return to the interpreter. In this sense, the interpreter can be considered as a dispatcher controlling the sequence of execution of various machine-code routines.

Initially the T/P interpreter is in control. This interpreter executes a program list by taking in turn each symbol of the list and interpreting it by type. There is a separate interpreter for each type. The T/P interpreter takes each symbol of a program list, determines its type, and then executes the interpreter for that type. The interpreter for type machine-code (T/M), for example, merely calls the machine-code routine as a subroutine. The interpreter for a data list places the address of that list on Z.

The relationship of the interpreter to the type system is more fully described in section 5.1. The detailed behavior of the interpreter when applied to a symbol are described in chapter 9 in the section on that type.

3.3.4 User interface, recognition, debugging, etc.

A large chunk of L* consists of routines which talk to the user and make life easier for him. These are routines which create programs, execute them, edit them, create and print data structures, define names, set breakpoints to trap errors, and so forth. These routines all interact with the user, who is in control at all times.

The recognition system stores symbolic names (i.e. character strings) of internal L* structures, and allows the user to communicate with his system in terms of these external names. The user interface executive routine, called EXEC, allows the user to execute any L* routine from the terminal. In particular, it allows him to execute the structure-building operators which create programs, lists, and the like. The language in which the user communicates with the EXEC is called EL*, or the external language.

Since the EXEC can be used to execute any L* routine from the terminal, it can in particular be used to execute structure-building routines. Structure-building routines are those which create programs, build lists, define blocks, etc. The structure-building routines are given suggestive mnemonic names: '(' and ')' are the names of the routines which begin and end construction of a list; '[' and ']' are the names of the routines which begin and end construction of a stack, etc.

A Print Facility allows a user to print any symbol. The editor and stepping monitor and other debugging tools combine to form a comprehensive environment in which to find and fix bugs.

All of these facilities will be discussed in detail in the approprite chapters, consult the index or table of contents for details.

4. Structures: L* building blocks

4.1. Defining a type and a structure

The L* basic system, i.e. the initial system provided to an L* user, supports numerous data structures which have been found to be an appropriately small set of universally useful structures. Conceptually more important than the collection of structures defined in the basic system, is the ability to add programmer-defined structures to the basic system.

L* data structures are closely tied to types. In particular, there is usually a separate L* type for each kind of data structure, and a set of support routines to manipulate it. Normally, when a type is added to the system, several steps are involved:

> Create a type symbol, which is a symbol of type 'type' (T/T). This symbol represents that type.

> For the structure that the new type represents, code a creation routine, an erasure routine, and a print routine. Enter these routines in various type tables under the new type.

> Code service routines as needed to use the new type. If the type is to be executable, code an interpreter for that type and enter it in the interpreter type table.

4.2. Types and structures supported by the basic system

4.2.1 Programs

L*(I) program structures are linked lists, each symbol of the list being a step of the program. An L*(I) program list is structurally identical to a data list (see next section), but because program lists have a different type, they are handled differently in execution.

L*C.(C) program structures are compacted lists, i.e. blocks: a program is a contiguous block of memory with each word treated as a list cell, so that no 'next' pointer is needed, because the next cell can be found by indexing to the next sequential memory address.

The end of an L*(I) program list is indicated by a cell on that program list whose next field is NIL. The symbol part of that cell (called the termination cell)

contains a pointer to the head of the list. This pointer is useful for looping back to re-execute the program list. The end of an L*C.(C) compacted program list is indicated by a cell whose address has the low-order bit set. Since L* symbols are always word addresses, and word addresses on a PDP-11 are always even, no symbol will ever normally have its low-order bit turned on.

L* supports type program, or T/P, for program lists. A separate type, Type machine or T/M, is supported for machine-code programs. Machine-code programs and their relationship to program lists are discussed in section 7.1.2.

L*C.(C) supports a special type called type indirect, or T/I, which is a means for executing a program in a page that overlays the caller. (i.e. a page which sits in the same logical address space as the caller). A type-indirect symbol is a two-word symbol naming a program and an overlay. The interpreter action for T/I is to load the overlay, call the program, then return to the previous overlay. See section 8.4 for a description of the L* overlay mechanism.

4.2.2 Lists

L* lists are non-executed data structures, as opposed (for example) to the program structures defined above. A list is normally of type list, or T/L. A list is a collection of linked cells, each cell having a contents and a pointer to the next cell of the list. The last cell of a list is called the termination cell of that list, and is not normally considered to be part of the 'contents' of the list. A termination cell is one whose 'next cell' pointer points to NIL.

A list cell must then contain (at least) two addresses. In L*(I) on the PDP-10 (a 36-bit machine) a list cell is a single word, with an address in the left half and an address in the right half. In L*C.(C), running on a 16-bit machine, a list cell is a contiguous pair of 16-bit words, each of which holds a symbol.

The two symbols in a list cell are called the symbol field and the next field. The next field is the symbol of the next cell of the list (or NIL if this is a termination cell), and the symbol field is the 'contents' of the list cell, and can be any L* symbol. Note that the contents of the next field is in fact a symbol. (If that sentence did not make sense, then go back and read section 3.2). As is also the case with program lists in L*(I), the symbol field of a termination cell contains the address of the head of the list.

When a cell is added to an L* list, the physical storage for the list cell is acquired from the L* storage allocator, and the new cell is linked into the list. When a cell is deleted from an L* list, it is unlinked from the list and then its storage is returned to available space. There exist numerous routines for examining, modifying, copying, searching, and iterating over lists.

4.2.3 Stacks

A stack, often called a pushdown stack, is a well-known data structure characterized by its last-in, first-out behavior. In L*, a stack is implemented as two pieces: a block of storage to hold the contents of the stack, and a separate 'control block' which holds pointers to the data part of the stack. L* stack structures are normally of type stack, or T/J.

When an L* stack is initially created, one must specify the size of the stack; this will be the size of the data portion. The 'control' portion of the stack has three parts: a 'current position' pointer, a 'stack full' marker, and a 'stack empty' marker. If a program attempts to remove (i.e. "pop") an entry from an empty stack, the stack manipulation routine will trigger an error event named EJUF (see section 5.2.2). If a program attempts to add (i.e. "push") an entry onto a stack which is already full, an overflow event named EJOV will occur. Normally EJOV contains code to create a new larger stack, copy the contents, erase the old stack, and return to the interrupted code, which will then succeed. L* stacks may thus be considered to have infinite size.

The L* kernel supports stacks both for its own use in the basic system, and for user convenience in building on top of the kernel. There are several specific stacks in the basic system which are crucial to the workings of L*. These particular stacks, called Z and ZX, are the main data and control stacks, respectively. They are described in section 3.3.1. The signal stack Z$ is also a stack, but its being a stack is not central to its behavior: L* could be made to run adequately if Z$ were merely a word instead of a stack. For this reason, Z$ is not considered here to be a central stack.

4.2.4 Words and blocks

L* provides support for manipulation of arbitrary words, whose contents and structure are entirely up to the user. There are operators to perform arithmetic, bit manipulation, comparisons, etc., on these 'arbitrary word' symbols. The basic system supports type word, or T/W for these arbitrary words. words. By clustering words into contiguous blocks, it is possible to build array structures and vectors, etc. A block may be built from symbols of any type; blocks are not restricted to T/W. It just seemed convenient to talk about them here since most of the blocks in the L* basic system are T/W. The L* basic system does not support arrays as a data structure.

4.2.5 Associations and association lists

An association, in the most general sense of the term, is a triad of associated values. Customarily they are called the attribute, the symbol, and the value. One speaks of associating a value with a symbol along some attribute.

Many systems support association structures to varying degrees of power and complexity. A totally general association system, such as that available on an associative-memory machine like STARAN, allows arbitrary content addressing: i.e. one may with very little loss of efficiency ask such questions as "along which attributes is the value 'true' associated from something?" Some software implementations of association systems, such as LEAP, allow this sort of access to the association structures, but at a great loss of efficiency.

L* implements associations as a structure of pairs. Each structure is an attribute; an association is stored in the structure as a pair: (symbol, value). In this sort of structure, lookups of the form "what is the value of symbol XYZ along attribute ABC" are very efficient, but lookups of the form "which attributes have . . ." are exceedingly slow, requiring an exhaustive search of all attribute structures in the system.

An association list is a list in which associated pairs of symbols are stored. It is of type association-list (T/AL). Because of different word sizes of the PDP-10 and the PDP-11, L*(I) and L*C.(C) use slightly different fine structure in association lists. L*(I) stores an association list as alternating pair lists, with every other cell containing a symbol and a value. L*C.(C) stores an association list as a list of pairs, with two symbols in every list cell. Although the service routines must be slightly different, the net effect is the same.

An association structure is a collection of association lists grouped into a hash table. The hash table is in fact a block of T/AL symbols. The symbol of the association structure points to the first word of the hash table. This symbol is of type association (T/A). When an entry is made in an association structure, the symbol part of the association is hashed into the table to specify which one of the association lists will receive the pair. Obviously, when there are a large number of pairs stored in an association, a large gain in efficiency will be effected by this technique.

Associations and association lists are executable data types. This means that they have an interpreter action that does something besides push the symbol onto Z. When the interpreter executes a symbol of type association or of type association list, it performs a lookup of the symbol on top of Z. Any reference in a program to an association list not intended to be executed should be quoted with the .Q quote symbol.

4.2.6 Character strings

Character strings may of course be implemented by the user using lists of character symbols (as was the case in earlier versions of L*), but to save space a character string facility was created for L*(I) on the PDP-10 and L*C.(C) on C.mmp. These are very ordinary character strings, with the usual collection of primitive operations to manipulate them. Character strings are of type character-string (T/KS).

L* character strings are stored tightly packed, with a marker indicating the end of the string. On the PDP-10, there are 5 ASCII characters stored per word; the end of the string is signified by a low-order '1' bit. On C.mmp there are 2 ASCII characters stored per word, with a zero character marking the end of the string.

5. Program execution and environment

5.1. The interpreters

The L* interpreter is the sequencer, the controller of execution. When each routine has finished executing, it returns to the interpreter, which then finds the next routine to be executed and processes it. 'The interpreter' is actually a collection of interpreters and a type-driven mechanism for determining which one to invoke.

5.1.1 Finding the interpreter for a type

At the innermost level there is a loop which steps down a program list, taking in turn each symbol and executing it. To 'execute' a symbol means to put it on top of Z, then determine its type, then call the interpreter for that type. Since the interpreters work off Z (as opposed to some internal registers as they did in prior versions of L*), an interpreter may be written and debugged as a normal user program, taking input in the normal way. There is a type table identifying the interpreter for each type. When a type is created, one of the responsibilities of its creator is to ensure that the interpreter type table has the correct entry for the new type. On L*C.(C) the type table is in fact a table, named TTI. On L*(I), the type table is an association structure named ATI. Be it a table or an association, the interpreter type table identifies the interpreter for each type. Even nonexecutable types like T/W (word) or T/L (data list) must have an entry in the interpreter type table. The entry for nonexecutable types is simply a NOP.

The T/P (type program) interpreter has a mechanism built into it allowing an event to be executed at the end of each interpreter cycle. Events are most frequently used in L* as error-handling devices; in this context they are described in section 5.2.2. The interpreter maintains a step event, which can be made to trigger at the end of every interpreter step. The interpreter step event is called ESTEP. If the side signal cell $STEP is TRUE (i.e. if it contains something besides NIL), then the interpreter will execute the topmost symbol of ESTEP at the end of each interpreter step. This mechanism allows all sorts of debugging and diagnostic tools to be written. The stepping monitor (see section 12.2) and the generalized breakpoint facility (see section 12.1) both use the interpreter step event.

5.2. Error detection and recovery

5.2.1 Error checking

A minimal amount of error checking is performed by the L* basic system; these error-checks tend in general to be for very severe errors which could if committed make debugging difficult or impossible. A user may of course modify the system to include error checking wherever so desired.

When an error is detected, an error event is triggered. An error event contains a subroutine which will be executed by the code that detected the error. Often the error event will print a message and then call the EXEC. If you print the call stack ZX inside an error event, you will see that the call history shows the routine that has detected the error called an error routine in the L* kernel, which in turn called the error event routine, which in turn called EXEC.

5.2.2 Error events

An error event is a data list, and each cell of the list contains a reference to a program. When an error is detected, the kernel error-event routine uses the program at the head of the list as the error routine. If a user wishes to take over an error event for his own purposes, he merely inserts his error routine on the front of the list, effectively pushing the previous error routine. To return to using the previous error routine, just delete the head of the list.

The error events used by the L* basic system are enumerated in section 12.3.

5.2.3 Continuing from an error

Since an error event is a subroutine, it will eventually return to its point of call and continue from the point where the error was detected. If there is no call to the EXEC in the body of the error routine, then the interrupted program will resume as soon as the error routine has completed. Normally, however, an error event routine will break to the EXEC to await user action. From the EXEC, the user may examine the call stack to find out how the error occurred, examine various data structures to determine what caused it, etc.

It may very well be that the user would like to return from the EXEC and continue executing the interrupted routine where it left off. For example, let us suppose that the error detected was 'Undefined Program'. The error event for an undefined program is called EUND/P. It will print a message saying that the user tried to execute an undefined program, and the name of this program will be printed. The user may type in a definition of the program and resume execution.

To exit the EXEC and resume execution of its caller, type a ↑Z (control-Z). The character action for ↑Z sets a switch that causes EXEC to return to its caller.

If you type ↑Z to the 'bottom level' EXEC, i.e. the EXEC that was first in control when L* began execution, an interesting sequence of events will take place: The EXEC will return to its caller, which will discover that there is no prior caller on ZX for it to return to, thereby triggering a 'ZX underflow' error, which will ultimately break to the EXEC. You can't get away from it.

6. The user interface

The L* user interface is the mechanism and language by which a user communicates with and controls the L* system in front of him. The extreme flexibility and generality of the L* user interface accounts for a major part of the power of L*. Via this interface, the user at the terminal plays the role normally played by an executive routine, executing L* operations at will, singly or in groups.

The user interface consists of several pieces, each of which will be discussed in turn. As seen by the user, however, there is a single homogeneous interface; its structure is not visible at the surface.

6.1. The EXEC

The routine which controls the user interface is called EXEC. It is conceptually very simple: it reads a line of input from the user interface, and then processes that line from left to right, interpreting the L* routines whose names appear and which have the attribute of being active. The EXEC maintains an association structure named 'AACT', in which the value 'TRUE' is associated from each active symbol.

The process of interpreting a line input by the EXEC consists of three steps, which are executed repeatedly until the input line is exhausted:

> Assemble a name from input characters.
> Find the L* symbol denoted by that name (by searching one or more name contexts)
> Execute the symbol if it is active, or put it on the data stack Z if it is passive.

The process described above is (more or less) what actually happens at the EXEC. It is not an oversimplification for the purposes of description. That's all: it really is that simple.

6.1.1 Character actions and name assembly

An L* name may consist of any combination of characters other than the format-effector characters like blank, tab, line-feed, etc. For example, "(((" is a perfectly valid name. To provide a name assembly scheme powerful enough to be able to recognize names consisting of arbitrary strings of characters, and also to allow arbitrary user modification of the name assembly process, L* employs an unorthodox character recognition and assembly scheme centered around character actions. A character action is a program executed by the EXEC whenever that character is encountered during the scan. These character-action programs are normal everyday L* subroutines, normally program lists, and may be modified or replaced by the user.

For example, the action associated with lower-case letters is to convert the letter to upper case, then execute the upper-case character action. The character action for upper-case letters is to accumulate the character as part of a name. The action associated with digit characters is a fairly elaborate program which checks whether a number or a name is being accumulated, then adds the digit either to the end of a name or to the number.

There are three character actions in the L* basic system which are crucial to the workings of the recognition system. They are the name character action, rigid boundary character action, and conditional boundary character action. Alphabetic characters all invoke the name action: the character is concatenated to the end of the name being accumulated. Rigid boundary characters like space, tab, etc., trigger recognition. Conditional boundary characters, which comprise most of the punctuation characters, are treated as name characters with one important exception: they can participate in a name, but can also trigger recognition when appropriate. The details of these character actions are described in section 10.1.

6.1.2 Recognition

When the EXEC has accumulated a string of characters which could form a name, it then tries to recognize them as a name. This recognition process is slightly more complicated than a simple table-lookup, because of the following rules for recognition:

> The recognizer will always recognize the longest possible name in a string of characters. Thus, if "?", "?Z", and "?ZX" are all defined names, then the string "?ZX" will be recognized rather than "?" followed by "ZX".

> In the event that the longest possible name is not a name already defined, then the recognizer will back up to the most recent conditional boundary character and try again.

> If there is no conditional boundary character in the unrecognized string, then the recognition system will define that string as a name, and create a symbol for it according to the prevailing creation modes (see 10.3).

> The first occurrence of a name is the defining occurrence; its type (and for L*C.(C) its page) are determined by the creation modes in effect at the first occurrence.

The alert reader will notice that under the above scheme, there is no way to define a name which contains a conditional boundary character (other than the trivial case of a 1-character name), since an undefined name containing a conditional boundary character will never be recognized. To solve this problem, there is a special recognition mode which may be set to temporarily force all

characters other than rigid boundary characters to be name characters. This recognition mode is set by the double-quote (") action. Double-quote is an active sybmol which sets this 'all characters are name characters' mode.

It is time for some examples.

```
ABC:DEF     will be recognized as 'ABC', ':', and 'DEF'.
"ABC:DEF    will be recognized as 'ABC:DEF' because of the "
ABC:DEF     will now be recognized as 'ABC:DEF'
```

Once the name 'ABC:DEF' was defined in the second case, then in the third case, even though the double-quote was no longer present, the entire name including the colon will be recognized.

6.1.3 Recognition actions

Once a name has been recognized as a symbol, the EXEC now acts upon it. This action is very simple. If the recognized symbol is active (as recorded in the association structure AACT), then the symbol will be immediately executed by the EXEC. If the recognized symbol is passive, or if the symbol has been temporarily passivated by means of the single-quote operator (see below), then the symbol will be pushed onto the main data stack Z. This active/passive distinction for a symbol is in a sense another dimension of type: a symbol of any type can be made active.

One particular active symbol is noteweworthy: the exclamation point. The '!' is an active symbol which causes the topmost symbol of Z to be executed. Thus

```
A !       executes 'A'
A! B!     executes 'A', then 'B'
A B ! !executes 'B', then 'A'
```

In the third example, each exclamation point executes the topmost symbol on Z, and B is at the top of Z, so it will be executed first.

If a name is prefixed by a single quote ('), then the recognition system will assume that the symbol it names is passive, and will not execute that symbol even if it is in fact active. This feat is accomplished by having a program whose name is single-quote which is active. When the single-quote program is executed, it sets a flag for the recognition system so that the next (and only the next) symbol will be treated as passive.

One frequent use of the deactivation quote is to allow a symbol to be printed. For example, here is how to print out the left parenthesis program ('PR' is a program which prints programs):

```
'( PR!
```

Note the single-quote to deactivate the left parenthesis, and note the exclamation point to execute 'PR'.

6.1.4 Name contexts and context lists

A name context is a table which pairs symbols with names. To recognize a name means to find it in a name context. To create a name means to enter it in a name context.

A context list is a list of contexts. A context list is used to define the order in which contexts will be searched. When the EXEC is searching for a name during the recognition process, it will search all contexts on the current recognition context list. If it does not find the name in any of those lists, then it declares the name to be undefined and acts accordingly (see 6.1.2). If it finds the name in any of the contexts on the current recognition context list, then it takes the symbol found there as the symbol for the name, and continues.

L* allows users to manipulate the context lists used in recognition and creation, thereby effecting a kind of dynamic block structure. ZCXRGL is a list of context lists, the head of which is the 'current' context list. BCX, the base context, is the context in which most of the names in the L* basic system are defined. UCX, the user context, is provided for defining user names. Typically, ZCXRGL is defined as

```
ZCXRGL : ( (UCX BCX) )
```

i.e. it contains a context list which has in it only one context, namely the base context BCX. Let us suppose that the user has a context of his own (which is created with CRCX, see 10.2) and he would like L* to search his context before it searches BCX, so that he may redefine some names in the basic system without clobbering their definitions. There are two ways of effecting this. First, he could add his context at the top of the current recognition context list:

```
ZCXRGL : ( (MYCX UCX BCX) )
```

or he could push a whole new context list onto ZCXRGL:

```
ZCXRGL : ( (MYCX) (UCX BCX) )
```

In the first example, the recognition system will search for a name first in MYCX, then in UCX, and finally in BCX. In the second example, only MYCX will be searched.

There is also a context list used for creation, ZCXCRL, which is used to find the context in which a name will be created if it is found to be undefined. The name will always be created in the topmost context of the creation context list; the lower contexts will be ignored. Thus, in our example above, all creation would take place in MYCX.

6.2. Building program and data structures

In a compiler-based system, programs are built by the compiler under control of the source code: a loop statement in the source code, for example, causes the compiler to generate the necessary object code to execute the loop. In a growing system, programs are built dynamically by the interpreter under control of the command input (whether that input is from a file or direct to the terminal).

In a compiled system, the 'external' or 'source' language is compiled into an 'internal' or 'target' language. These two languages tend to be very dissimilar; for example, a FORTRAN compiler takes FORTRAN statements as its external language and produces machine code as its internal language.

L*, as a growing system, provides a means of building programs under control of the command input. This command input is L*'s external language, referred to as EL*. Although EL* is not a formally specified language, and it is not actually a programming language (it has no control actions, for example), EL* is syntactically similar to the internal language PL*. This similarity between the internal language and the external language is so uniform that beginning L* users have sometimes failed to see that there were in fact two languages, and treated the two as one. It is better to think of EL* as a command language rather than as a programming language. The print facility (see section 7.4) prints a PL* program (or data structure) in EL*; most of the time it will produce a faithful reproduction of the input.

6.3. Reading files

So far we have made no mention of program source files; rather we have always talked about typing in a program at a teletype. Obviously, no program of any reasonable size will be typed in from the keyboard, so there is a mechanism for causing the EXEC to take its input from places other than the teletype.

I/O in L* is performed through I/O interfaces. The L* kernel ensures that interfaces are uniform from one device to another: reading a character from a disk file or from a DECtape or a teletype is effected at the user level with the same call. Obviously, more information is required to open a disk file interface than a teletype interface, but once opened, all interfaces are accessed in the same way.

To read a source file containing a program, we would then do these things:

> Open an interface for the file, by whatever means appropriate. Usually this would involve taking the predefined 'Disk Read' interface and modifying it to read the requested file.
> Push the symbol for this interface onto ZRD, so that future input requests will be taken from the file.
> Call the EXEC recursively. When it encounters an EOF in the file, it will exit.

> Restore ZRD by popping the entry we put there.

In fact, there is in L*(I) a function that does all of the above things. Named RDF, it takes a file name as input and reads it into the core image:

```
SFILE,C14  RDF!
```

This will perform all of the correct actions to read the requested file.

Until such time as a file system becomes available on C.mmp, the primary storage place for L* source will be the PDP-10. There is a data link between the two machines, and L* knows how to use it to read files from the PDP-10. A server job is logged in on the PDP-10, and a file transfer protocol is executed to ship a source file over the link to C.mmp. The L*C.(C) end of the file transfer protocol is called SJEXEC, the sub-job EXEC. Use of the link and the file transfer server is described in section 11.5.2;

Part III: Programming in L*

7. Building and printing structures

L* systems are not deposited as with a compiler, but are grown from within: source code processed by the L* EXEC causes L* to create program and data structures within itself. There are numerous grown systems in existence, and each has its own syntax and its own mechanism for directing the system to grow.

The technique used to grow and build L* programs is simplicity itself: the user types (or reads from a file) a list of names of L* routines to be executed. The routines are interpreted if they are active or pushed onto Z if they are passive, in the order that they are received by the EXEC.

By using appropriate routine names, any surface syntax may be achieved.

Before we delve deeper into the details of the growing process, perhaps an illuminating example would be in order. Let's look at the input to L* which would cause it to build a list containing three numbers:

```
( 1 2 3 )
```

In this case, '(', i.e. left parenthesis, is the name of an active routine which begins the construction of a list, and ')' (right parenthesis) is the name of a routine which completes the construction of the list and leaves the address of the completed list on Z.

7.1. Building programs

A program is a particular data structure. All L* structures have types; a program is T/P, or type program. The details of an L* program list's internal structure are in section 4.2.1.

The L* basic system admits of two program types: T/P for symbolic program lists, and T/M for machine code. We shall discuss in turn how each of these structures is built.

7.1.1 Creating program lists

The act of building a program list is begun by a left parenthesis, and the building of a program list is completed by a right parenthesis. This is equally true for a data list. When a left parenthesis is encountered, it decides which type of

structure to build on the basis of the type which it is told to create. Types and how to control them are discussed in more detail in section 10.3. For now, we assume that the creation type is set up properly to create a program list rather than a data list.

As we have said several times, a list is created by enclosing the objects for the list in parentheses. Thus,

```
(A B C)          creates a list with A, B, and C

( () )           creates a list containing a null list

(XXX  WR)        creates a 2-element list
```

The second example above shows the creation of a list within a list, and shows the recursive nature of the list-building operators. The second left parenthesis, encountered before the first one is closed, simply begins a new structure. The first right parenthesis completes the inner structure, which is left on Z to be included as a symbol of the outer structure.

When a structure is created, we would like to keep a pointer or reference to it, so that we may access it again externally. We do this by giving a symbol a name. To give a symbol a name means making an entry in a name context with that symbol and its name. The L* recognition system provides an automatic means of giving names to structures. This is done using the ':' (colon) operator, which is one of the few infix operators in L*.

To build a list and give it a name, we type:

```
NAME : ( 1 2 3 )      build a list; call it "NAME"

N34 : (NAME 3)        build a list; call it "N34"
```

The colon is a fairly complex program; the details of its operation are left for the L* reference manual. Since the name to the left of the colon has already been defined by the time the colon is encountered, the colon and the parentheses must cooperate to create a structure whose head cell is at the address where the symbol is defined.

When a right parenthesis finishes building an unnamed structure (either program or data list), it leaves the address of the structure on Z. When a right parenthesis finishes building a structure which will be named (i.e. by a colon), then it does not leave the address on Z, the philosophy being that the address of the structure may be obtained from the name if so desired.

Since the structure-building operators involved in building a list are all actively-executed programs, and since the recognition system doesn't care about carriage returns, spaces, tabs and the like, it is perfectly legal to take great liberties with the physical format of a list as it is being read in. The following forms will all create the same list:

```
A: (X Y Z)

A : (
 X Y
 Z )

A : (
 X
 Y
 Z
 )
```

Remembering that the semicolon character when processed by the recognition system causes the remainder of an input line to be discarded, it is frequently desirable to separate an L* program source into several lines, with a comment on each line preceded by a semicolon.

7.1.2 Creating machine-code programs

7.1.3 Editing list structures

L* provides an editor which may be used to edit program and data list structures. Data lists are linked structures, and in L*(I) so are program lists; in L*C.(C) a program list is a compacted list. The L* editor can edit both kinds of structures, though obviously the editing actions that can be performed on a compacted list are somewhat limited.

The editor is a PL* program. It takes as input a symbol from Z, and leaves no output. Its action is to accept commands from the user and modify the list accordingly. It is totally recursive: if in the middle of an edit one decides to go edit another structure, he may do so without closing the first edit. In fact, the editor is an open program; it takes its input from the EXEC. Any L* program may be executed while inside the editor, and since the editor is an L* program, it may be executed while inside the editor.

To open an edit, type:

```
<name>  EDIT!
```

The editor will respond by printing the first symbol of "name" and waiting for input. The editor maintains a symbol cursor, which points to the 'current symbol' in the structure under edit. Initially the editor cursor is on this first symbol of the list. The following edit commands are implemented:

!R Replace the symbol at the cursor with the symbol on top of Z. In use, one would type "<symbol> !R", which puts the symbol on top of Z and then executes '!R', which is active.

!I Inserts the symbol at the top of Z into the list being edited just before the current cursor position. This command is not permitted when editing compacted lists. One would type "<symbol> !I". Like all editor commands, !I is an active symbol.

!D Deletes the symbol at the current cursor position. This command is not permitted when editing compacted lists. !D takes no input from Z; one would type merely "!D".

!N Moves the cursor to the next sequential symbol. A linefeed may be used to accomplish the same effect. !N takes no input from Z.

!B Moves the cursor backwards to the previous symbol. An altmode will accomplish the same thing.

!F Searches the list from the current cursor position to the end of the current list level, for the symbol found on top of Z. One would type "<symbol> !F". Compare this with '!FX' below.

!FX Searches the list from the current cursor position to the end, and searches all unnamed sublists as well. Like !F, !FX searches for the symbol it finds on top of Z. By comparison, !F will not find a search target if it is contained in a sublist of the list being searched.

!S Step down into a sublist. This causes the editor to save state information in the list currently under edit and open an edit of the sublist at the current cursor position.

!U Pop up from a sublist edit. Reverses the effect of !S.

!. Terminate editing. Close up the editor and return to the caller.

7.2. Building non-program structures

Other structures besides programs may be built in the EXEC with syntactic structue-building operators.

7.2.1 Data lists

The notation used to build a data list is exactly the same as the notation used to build a program list: left and right parentheses surrounding the symbols to be included in the list. Whether a program list or a data list is created is determined by the creation type set when the left parenthesis is encountered. Section 10.3 describes the means of controlling creation types.

7.2.2 Associations and association lists

An association list is structurally similar to a list, and it is created in a syntactically similar way. In L*C.(C), angle brackets (< >) are the structure building operators for association lists. In L*(I), where angle brackets mean something else (they are part of the assembler), association lists are built with ordinary parentheses. The parentheses are directed to build an assocation list rather than a normal data list by means of the creation type in effect when the left parenthesis is encountered. Section 10.3 describes creation type control.

In the examples in this section we will show both notations.

The following are all valid association lists:

```
< A B C >
AL\( A B C )          association list with 3 elements (L*C.(C))
                      L*(I)

<>
AL\()                 null association list

< A B < X Y > C >
AL\( A B ( X Y ) C )  nested association lists
```

Recall that an association list holds pairs of symbols; each 'cell' in an association list has a symbol and a value associated from that symbol. Our examples above do not tell the whole story, as they show only one symbol per list cell. The slash (" / ") character is used to separate values from symbols; the value field comes first:

```
< 1/A 2/B 3/C >
AL\( 1/A 2/B 3/C )    associate 1 from A, 2 from B, etc.

< NIL/X >
AL\( NIL/X )          associate NIL from X
```

The slash character is not an active program, but is only a marker; the actual processing of the contents of the association list is performed by the ">" operator in closing the list.

There are no 'syntactic' structure-building operators to build association structures; that is, there is nothing which corresponds to the parentheses which build lists and the angle-brackets which build association lists. There is, however, a routine which will copy an association list into an association and delete the original list, so that one may use the association-list structure-building operators "< >" to build an association. The routine SETA copies an association list into an association. it takes as input an association list as (1) and an association structure as (0), copies the association list into the association, then erases the association list:

```
AL\< 1/A 2/B 3/C >  A\ATTR SETA!
```

The association 'ATTR' will now have in it the three associations shown, and the association list will have been erased. The 'AL\' and 'A\' are creation type control symbols, whose precise meaning is described in section 10.3.

7.2.3 Word structures and blocks

Since words are not very complicated structures, no particular structure-building operators exist to build them. The only kind of 'structure-building' that you might want to do with a word is to give it an initial 'compile-time' contents. Recall that L* draws no particular distinction between 'compile' time and 'execution' time, so that the normal RW (replace word) operator may be used to give a word a contents at compile time:

```
3  W\WORDA  RW!  Give 'WORDA' the value 3

-15 W\WORDB  RW! Give 'WORDB' the value -15
```

'RW' is a word manipulation operator, detailed in section 9.5.

The '[[' and ']]' operators are used to create blocks. A block is a contiguous group of words which all have the same type. The symbol for the block is the address of the first word of the block.

When the recognition system encounters a '[[' symbol, it begins a block, and when it encounters a ']]' symbol it finishes creation of the block. The type of the block is determined by the prevailing block creation type. The size of the block is determined by the number of symbols left on Z between the '[[' and the ']]'. L*(I) and L*C.(C) differ somewhat with respect to the handling, naming, and typing of blocks; see section 10.3 for a description of type control. L*(I) allows one to name a block with the ":" (colon) operator, while L*C.(C) does not allow this construct. Either version allows any word in a block (including the first) to be named using the :L operator (see section 10.2.4).

By way of example:

```
[[ 0 0 0 0 ]]                 creates a 4-word block of zeros

[[ A 12 ROUTINE! 3 ]]         block with 3 or more, depending
                              on what 'ROUTINE' leaves on Z

NAMEDBLK : [[ 2 X ]]          L*(I) only: named block

[[ NAME :L 2 3 4 ]]           block with first cell named
```

No mention is made of block type in the examples above; it is assumed that they are all of type word. The second example shows a routine call inside the block

creation. Recall that the block is created out of those items found on Z by the]] operator. If the call to 'ROUTINE' causes anything to be left on Z, then the item(s) will be included as part of the block. As an example of this technique carried to an extreme, the following example creates a 20-word block containing the numbers 101 to 120:

```
[[ 101 W\*W1 RW! P\(*W1 CW *W1 1 +W U) 20 .XN! ]]
```

That example uses several features that have not been documented yet. .XN is an iterative control action; see section 8.2. CW is the copy-word operator, see section 9.5. *W1 is a local name, see section 10.2.3.

7.3. Storage management

7.3.1 Word storage management

The word-level storage allocation routine is responsible for allocating and deallocating chunks of memory of one word or more. The creation routine is called CRBN, and the erasure routine is called E. CRBN is called with a type and a word-count as arguments. It finds a block of storage of the requested type and size, and returns the address of that block to the caller by leaving it on Z.

CRBN first looks at the available-space list for the requested type. This is a list of blocks of availble space. It is not structured like normal L* lists, rather its format is special to the available-space mechanism. If the available-space list contains a block of the correct size, or if it contains a block that can be chopped up giving one of the correct size, then the space is removed from the available-space list and returned to the user.

If CRBN cannot find the requested storage in the existing typed available-space list, it executes the error event ESPX. Normally ESPX contains a routine which will request a block of T/AV (block available space) and change its type and link it into the available-space list, and continue. If, however, there is no block available space remaining, then the second call will cause a normal error and break to the EXEC. It is very rare to run out of space on the PDP-10, and quite frequent to run out of space on C.mmp.

7.3.2 Block storage management

The largest unit of storage allocation common to L*(I) and L*C.(C) is the type block. Since C.mmp is a page-oriented machine, L*C.(C) also has a mechanism for allocating whole new pages of storage, but this mechanism is not integrated into the storage allocator to the extent that it will create a new page if it runs out of space on an an old one.

A type block is a block of 128 consecutive words, so situated that the high-order bits of the address are identical for every word in the block. Until specifically allocated, all unused type blocks are of type 'available space' (T/AV), and are in the word available-space list for T/AV. When the word storage allocator runs out of block space to allocate, it gets a block of T/AV and changes its type as needed.

7.4. Printing

The L* print facility is in a sense the inverse of the structure-building facility: it takes as input an L* structure, and produces as output an EL* representation of that structure. The output of the print routines is usually very similar to the input which created the structure; in fact, in most cases it is identical except for the formatting characters like spaces and tabs and CRLF's.

7.4.1 Printing whole structures

The highest-level L* print routine is named PR. It takes as its one input a structure, and descends recursively down that structure, printing out all of the substructures by type. A 'structure' in this sense is really just a symbol, an L* datum. A single word is a structure, and so is a program or a list. The printed output appears on the current output interface (see section 11.1.1). PR is a passive symbol designed for use inside programs. Since structure printing is so very useful and done so frequently by a user while debugging, there is an active symbol, ?, which performs the same action. ? is equivalent to PR !.

When PR prints a structure, it prints each element of the structure according to the type of the element. Some structures, such as words, have only one element; others, such as lists, can have any number of elements. When a list is printed, each symbol of the list will be considered in turn and printed according to its type. If the symbol has a name, then only the name of that symbol will be printed. If the symbol does not have a name, then PR will be called recursively to print the sub-structure. Consider the following example:

```
> L: (X Y (L 1 Y) Z)  Define a list with 4 elements
> L PR!               Print it
> L: (X Y (L 1 Y) Z)  Note output identical to input
```

The list in our example has 4 symbols. The first two, 'X' and 'Y', are named. The third, '(L 1 Y)' is an unnamed list. The fourth, 'Z', is named. When PR printed this list, it printed the name of X, the name of Y, then called itself recursively to print '(L 1 Y)'. This recursive subcall caused the 3-symbol sublist to be printed; its first and third symbols are named and are printed accordingly. The second symbol of the sublist is a number; the number has no name, so its structure was printed, and its structure is just its integer contents.

This scheme for printing might sound unnecessarily elaborate, but it yields a mechanism which can reconstitute the source form of a program from the internal representation, a valuable tool indeed.

7.4.2 Printing symbols by type

When the high-level print routines print a structure, they print each of its elements by type, as described in the previous section. This scheme of printing by type is effected by having a collection of low-level print routines organized by type, and a pair of type tables which select the routine to use for each type. In L*(I), a type table is an attribute, in L*C.(C), a type table is an array. Both are used in the same way: to print a symbol, its type is determined, then that type is used to select a print routine from the type table, and that print routine is used to effect the printing of the symbol.

Of the two type tables mentioned above, one is used for structure-printing and one for symbol-printing. In general, printing a symbol means printing its address followed by a percent sign, the percent sign indicating address rather than number. Printing the structure of a symbol is the process performed by PR as described in the previous section.

The low-level print routines for each of the basic-system types are shown below:

Type	Routine	description
T/L	PRL&	print list structure without CRLF
T/W	PRW&	print a word in current conversion radix
T/J	PRJ&	print a stack without CRLF
T/P	PRL&	(L*(I) only): print program list
	PRY&	(L*C.(C) only): print compacted pgm list
T/A	PRA	print association structure
T/AL	PRAL	print association list structure
T/AV	(none)	not a printable type
T/I	PRI&	(L*C.(C) only): print an indirect symbol

7.4.3 Printing numbers and addresses

Often it is useful to print a symbol as an address, or to enter a numeric address. L* uses a postfix notation to denote an address: a number followed by a percent sign, as 20072% is an address. The routine PR% (and its companion PR%& to print without a CRLF) prints a symbol as an address. The numeric part of the address is always printed according to the current numeric conversion radix in ZBN. PR%& is normally the routine entered in the type table TTPRS (described in the previous section).

8. Program control

L* programs normally execute sequentially; the interpreter moves down a program list and executes each symbol in turn. Certain operators, called control operators can affect the behavior of the interpreter, causing it to change the flow of execution control.

These control operators usually begin with a period as the first character of their name. They fall into three general categories, each of which will be documented in detail in a section to follow.

> Conditional operators, which cause the program to selectively execute or bypass a portion of itself.

> Iteration operators, which enable the program to loop.

> Special control operators, which perform specialized control tasks.

8.1. Conditionals

Conditionals are used in any programming language to effect selective execution of pieces of a program depending on the results of computation. Like all programming languages, L* has the ability to do conditional execution.

L* conditionals work by causing the interpreter to bypass a portion of a program list. Before delving into a description of conditional operators, let us define some of the relevant terminology.

Recall that a program list is a list of symbols to be executed. Any of those symbols can be a reference to another program list. When the interpreter is interpreting a symbol of a program list and finds that the symbol is in fact another program list, it calls itself recursively to evaluate the sub-program. This recursive call of the interpreter on itself is called descending a level, and the corresponding return back to the previous program list is called ascending a level. The 'current level' is the list level being executed by the interpreter at the time that a conditional operator is encountered, and the 'higher level' is the program level which called the current level.

8.1.1 Conditional operators

A conditional operator works by causing the interpreter to bypass the remainder of the current level, and/or to bypass the remainder of the higher level. Obviously, this bypass must be selective on the basis of a test: conditional operators have one effect with a TRUE input and another with a FALSE input.

The simplest conditional operators are .- and .+. '.-' ("dot minus") causes the remainder of the current level to be bypassed if the signal is false (i.e. NIL), and has no effect if the signal is true (i.e. not NIL). '.+' ("dot plus") has the opposite effect: it causes the remainder of the current level to be bypassed if the signal is true, and has no effect if the signal is false. The rationale for the names comes from combining the name "." (a single dot) with '+' for TRUE and '-' for false. "." is a singularly useless program which causes the remainder of the current level to be bypassed unconditionally. "." may be read as 'exit', and hence ".+" may be read as 'exit if true'.

As an example, the following program will print 'YES' if the contents of A is zero, otherwise it will print 'NO':

```
(A 0 =W (.+ S"NO" WR) (.- S"YES" WR) )
```

Notice that there are two sublists to the program, one of which will be executed if the '=W' comparison yields 'true', the other if it is yields 'false'. A more elegant control scheme could be implemented with more sophisticated control operators, and indeed L* has done so.

'.H' ("dot H") is an unconditional control operator which deletes the higher level. This means that whenever '.H' is executed, the interpreter will bypass one level of return. It is probably best to explain by example:

```
(A B ( C ( D .H ) E F G) H )
```
will execute 'A B C D H'.

```
(J ( K ( L .H .H ) M ) N O )
```
will execute 'J K L'.

In the first example, the '.H' will cause the interpreter to delete the remainder of the higher level which called the .H. The remainder of this higher level is 'E F G', so the E, F, and G will be bypassed. In the second example, each '.H' will delete one level: the first will delete the 'M', and the second will delete the 'N O', so that there will be no remaining levels to execute after the second '.H', and only J, K, and L will be executed.

'.H' by itself is marginally useful. In combination with conditional tests, it becomes the major L* conditional operator. These combination control actions are '.+H' and '.-H'. They combine '.-' or '.+' with '.H' in the following way: '.-H' will exit the current level if the signal is false, else it will delete the higher level (but continue the current level) if the signal is true. '.+H' has the opposite effect; it will delete the higher level if the signal is not true.

Using these conditional operators, our example above becomes:

```
(A 0 =W  ( (.-H S"YES" WR) S"NO" WR) )
```

Notice that only one conditional is required now instead of two, and that it takes the form of an if/then/else construct.

'.+H' is a combination of the '.+' action and the '.H' action. In a similar vein, there exist conditional operators which are a combination of the '.-' action and the U operator which pops the top of Z: '.U+' will exit the current level and pop from Z in the presence of a 'true' signal, and will have no effect in the presence of a 'false' signal. Similarly, '.U-' will exit and pop if the signal is false.

Let us summarize the conditional operators described in this section:

> .+ : exit the current level if signal false
> .- : exit the current level if signal true
> .U+ : exit and unstack if signal true
> .U- : exit and unstack if signal false

8.1.2 Signals and the signal stack

L* maintains a stack namd ZS, called the signal stack. It is used to hold true/false values produced by comparisons. Its value is tested by conditional operators. ZS is an ordinary L* stack, so its contents may be modified or examined to the satisfaction of the user.

Two routines exist to set and clear the value of the top of the signal stack. Routines exist to copy values in and out of it, and to push and pop it. The signal is defined to be the value of the top of ZS.

$+ ("dollar plus") is the routine which sets the signal TRUE. It accomplishes this by replacing the top of ZS with the value 'TRUE'. $- ("dollar minus") is the routine which sets the signal 'false'. False is represented by NIL. The comparison operators, such as the =W used in the examples of the previous section, all use $+ and $- to signal the result of their test.

Because ZS is a stack, it can be pushed and popped, and there are even primitives to help do so. PS will push the ZS stack (pushing a copy of the previous top), and US will pop the old value. Using these primitives, the programmer may save the value of the signal around calls to routines which might change it, simply by pushing the old value down the stack and letting the routine change the copy.

8.2. Iteration

Iteration control operators are used to effect iteration in L* program lists. L* has two kinds of iteration control operators: internal and external. An internal iteration control operator is one which appears inside a program, and causes that program to repeat itself. An external iteration control operator is one which 'surrounds' a program like an ALGOL do statement and executes the program repeatedly.

8.2.1 Internal iteration

An internal iteration control operator is one which appears inside the program being iterated, and causes that program to repeat. The primary internal iteration control operator is called .R. When .R is executed in a program list, it causes the interpreter to return directly to the head of the list, at the same level as the .R. Consider this example:

```
( A B C (D E F .R G ) H)
```

The sequence of program steps executed will be

```
A B C D E F D E F D E F . . .
```

which is an infinite loop. Notice that 'G' and 'H' will never get executed.

.R is an unconditional repeat operator. There are two operators, .R+ and .R-, which conditionally repeat depending on the signal. .R+ will repeat if the signal is TRUE, and will do nothing if the signal is FALSE. .R- has the opposite effect: it will repeat if the signal is false and do nothing if the signal is true.

Here are some examples of the use of internal iteration control operators:

Delete all elements of input list (0):

```
DELETEALL: (P F .- D .R)
```

Pop from Z all elements down to 'FLR'

```
CLEAN:  (FLR =S .R-)
```

'=S' is a list manipulation operator, described in section 9.2.

8.2.2 External iteration

An external iteration control operator is one which 'surrounds' the routine being iterated, and executes it until either the iteration terminates or the routine stops the iteration. In general, the external iteration operators take as input a program and a structure, and execute the program once for each element of the structure. This is called 'executing the program over the structure'. For example, the control operator .XL executes a program once for each element of a list. The program is called with the list element sitting on top of Z:

The sequence

```
PROG L (1 2 3 4 5) .XL!
```

produces the same results as the sequence

```
1 PROG! 2 PROG! 3 PROG! 4 PROG! 5 PROG!
```

namely, the program is executed once for each element of the list, with that element as its input.

The following external iteration operators are defined in the L* basic system:

> .XN Executes routine (1) through (0) iterations. The routine is called with no inputs.
> .XL Executes routine (0) once for each element on list (1), with the list symbol as input to the routine.
> .XA Executes routine (0) once for each (symbol,value) pair in association (1). The routine is called with 2 inputs, the symbol as (1) and the value as (0).
> .XAL (L*C.(C) only) Executes routine (0) once for each (symbol,value) pair in association-list (1). The routine is called with 2 inputs, the symbol as (1) and the value as (0).
> .XKS Execute routine (0) once for each character in character-string (1). The routine is called with the character as its input.
> .XB (L*(I) only) Executes routine (0) once for each symbol in block (1). The routine is called with the block symbol as input.

Any routine being iterated by one of the above external iteration operators can escape from the iteration if it sees fit by executing .L. Similarly, .L+ will escape the loop only on a TRUE signal, and .L- will escape the loop only on a FALSE signal. The iteration control operator will return a FALSE signal if the loop was escaped, and a TRUE signal if the loop ran to completion.

8.3. Special control operators

8.3.1 Symbol quote operators

Normally the interpreter executes all symbols on a program list. If a symbol is a nonexecutable (i.e. 'data') type then 'executing' it will mean pushing it on Z. If the symbol is executable, then it will be interpreted by type. Sometimes we want to treat executable types as data, i.e. have them placed on Z instead of executed. The quote operators .Q and .QH perform this task.

.Q causes the symbol following it in a program list to be treated as data. For example,

```
(.Q PR PR)
```

causes the first 'PR' to be treated as data, and the second one to execute normally. This program when executed will print out the program 'PR'.

.QH quotes the next symbol at the higher level. This is useful for writing ones' own quoting routines. For example,

```
XYZ: (.QH PR)

P\(XYZ A  XYZ B) !
```

will cause 'A' and 'B' to be printed out.

.Q and .QH work by modifying the interpreter's internal pointers to bypass the quoted item, then copying it to Z. The quoting action of .Q is only needed inside program lists. To print out the program 'PR' from the terminal (in EL*), one merely types

```
PR  PR!
```

Since the first 'PR' is not active, it will be placed on Z, then the second 'PR' will be executed and print the symbol on top of Z.

8.3.2 Execute control actions

.X is a control action which causes the symbol on top of Z to be executed. It performs the same action as does the exclamation point. ..X is a combination control action; it combines '.' and '.X'. ..X causes the program to exit the current level and the symbol at the top of Z to be executed at the higher level. Consider the following example:

```
(A B C (D E .Q .R ..X F) G H)
```

this will loop forever, executing the sequence

```
A B C D E A B C D E A B C D E A B C D E . . .
```

8.3.3 Infix operator control

L* is a postfix language; however, it does provide the ability to delay operators so that an infix notation may be achieved. There is an operator delay stack named ZQ, and a series of routines to manipulate it. Most important of the ZQ routines is .XQ, which executes everything it finds in ZQ down to a floor, then deletes the floor from ZQ.

As an example of the use of ZQ, consider the colon operator ':' used in defining a name. This colon is an infix operator. When first executed, it puts the name being defined onto ZQ, then a special postfix name definition routine onto ZQ, and exits. At one point during its execution, the right parenthesis program pops the delayed colon operator from ZQ and executes it, thereby effecting the intended definition.

8.4. Address space control (overlays)

9. Data manipulation Operators

The L* basic system defines and supports several important data types. By 'supports', we mean that it provides a full complement of routines to build and manipulate structures of that type. This chapter is devoted to an enumeration of the service routines provided by the L* basic system for the types which it supports. We shall not attempt any great level of detail in the explanation of most of these routines, because for the most part they are obvious. If the reader finds any of these explanations inadequate, he is encouraged to experiment online to determine the behavior of the routines. See section 1.2.2 for a road map to online exploration of pieces of the L* system.

9.1. Operations on the central stacks

9.1.1 Operations on Z

Z is L*'s main data stack. It is used for most parameter passing and temporary storage. See section 3.3.1 for a full description of Z. Since Z is a stack, we would expect to find available the customary stack-manipulation operations like 'push' and 'pop'. Indeed so. The bracket notation ([10]), etc.) used here is defined in section 3.2.1.

P [12] (args: [12]) pushes a copy of the top symbol on Z. After executing a P, the top two symbols on Z will be identical and equal to the symbol that was previously there. Thus:

```
Z PR!                    print Z, see what is there
Z: [TEST1 27 (R S T)]
P! Z PR!                 execute a P, then print it again
Z: [TEST1 TEST1 27 (R S T)]
```

The structure of the symbol is never examined or copied. If the symbol at the top of Z is a number, then after executing a P, the top two positions of Z will both have a symbol for the same number.

U (args: [10]), the "up" operator, pops the top symbol from Z. It does not erase it or even look at the structure, it only pops. Thus:

```
Z PR!                    print Z, note contents
Z: [X Y Z]               three things on Z
U! Z PR!                 execute U, look at Z again
Z: [Y Z]                 two things on Z
```

V (args: [22]) exchanges the top two symbols on Z. After executing V, the new (0) is the old (1), and vice versa.

P1 (args: [23]) push a copy of (1). This is identical to the P operation, except that it pushes a copy of (1) instead of (0). After executing a P1, the top of the stack (0) is unchanged, and (1) and (2) are identical to what (1) was before. Thus:

```
Z PR!                 print Z
Z: [A B C]
P1! Z PR!             execute P1, then print Z again
Z: [A B B C]          note additional B
```

U1 (args: [21]) pops (1) from Z, leaving (0) unchanged. It works by saving (0) in a safe place, then executing a normal U operator, then restoring the saved (0). Thus:

```
Z PR!                 print Z
Z: [1 2 3 4]
U1! Z PR!             execute U1, print Z
Z: [1 3 4]            notice the '2' has been popped.
```

PP (args: [24]) push pair: (0) and (1) are pushed in tandem, the new (2) and (3) being the old (1) and (2). Thus:

```
Z PR!                 print existing contents of Z
Z: [-9 A B C]
PP! Z PR!             execute PP, then look again
Z: [-9 A -9 A B C]    notice tandem push of -9 and A
```

=S (args: [20$]) tests if the top two symbols on Z are equal. If symbol (0) equals symbol (1), i.e. if they are the same address, then =S will signal true, else it will signal false.

+SW (args: [21]) increments the address of a symbol. If (0) is a word, and (1) is any arbitrary L* symbol, then +SW will add the value of the word to the symbol. As an example, suppose that BB is a symbol which begins a 20-word block. Assuming that we are on the PDP-10, where adding 1 to an address gives us the address of the next word, then

```
BB 4 +SW!             gives the address of the 5'th word of the block
BB 8 +SW!             gives the address of the 9'th word of the block
    etc.
```

-SW (args: [21]) decrements the address of a symbol. If (0) is a word and (1) is any arbitrary L* symbol, then -SW will subtract the value of the word from the symbol. Compare with +SW above.

SS$ (args: [21$10]) selects one of two inputs depending on the signal. If (0) and (1) are arbitrary L* symbols, SS$ will output (0) is the signal is true and (1) if the signal is false.

9.1.2 Operations on Z$

Z$ is the signal stack. Its topmost symbol is the signal. Many L* routines set the signal, and the conditional program control operators (section 8.1) test it.

$+ (args: [00$11]) sets the signal true, i.e. it sets the topmost symbol of Z$ to TRUE.

$- (args: [00$11]) sets the signal false, i.e. it sets the topmost symbol of Z$ to be NIL.

P$ (args: [00$12]) pushes the signal stack. It performs exactly the same operation on Z$ that P performs on Z.

U$ (args: [00$10]) pops the signal stack. It performs exactly the same operation on Z$ that U performs on Z.

V$ (args: [00$22]) reverses the top two symbols on the signal stack. V$ performs exactly the same operation on Z$ that V performs on Z.

C$Z (args: [10$10]) copies the signal into a 'side cell'. A 'side cell' is in a sense just a variable, a place to store information. C$Z causes the side cell at (0) to be changed to equal the current signal at the top of Z$.

C$ (args: [10$01]) copies a signal in a side cell back into the signal stack. The top of Z$ is changed to equal the symbol of the side cell.

9.1.3 Operations on the scratch stacks

There are several stacks maintained by the L* basic system for use as scratch stacks, to be treated somewhat like local variables. These stacks are named Z0, Z1, Z2, and Z3. They are ordinary stack structures, of the garden variety, and are noteworthy only in that they are used extensively throughout the L* basic system. They are of course available for use by any routine.

IZ0 (args: [10]) pops the top of Z and pushes it onto Z0.

PIZ0 (args: [11]) pushes the top of Z onto Z0, but does not pop it from Z.

SZ0 (args: [01]) pushes onto Z the symbol which is at the top of Z0, without disturbing Z0.

SDZ0 (args: [01]) pushes onto Z the symbol which is at the top of Z0, and at the same time pops it from the top of Z0.

DZ0 (args: [00]) pop the symbol from the top of Z0. Do not push it on Z or anywhere else, just pop it from Z0.

RZ0 (args: [10]) replace the top of Z0 with the symbol at the top of Z, and pop it from Z.

Obviously, there are routines with similar names to perform the same operations on the other scratch stacks. As an example of the use of scratch stacks, the routine P1 described in the previous section looks like this:

```
P1: (IZ0 P SDZ0)
```

9.2. List facility

The L* basic system list facility includes routins to perform the following kind of operations on Z: following kinds of operations on lists:

> Creating, copying, and erasing list structures
> Examining and modification of list structures

> Searching and testing list contents.
> Deletion and insertion of list cells
> Changing the contents of list cells

> Iterating programs over the contents of a list.

In reading these descriptions, keep in mind that all L* basic-system routines use the Z stack (section 3.3.1) as a storage place. All 'outputs' are left on Z, and all 'inputs' are taken from Z. The structure of a list is described in section 4.2.2.

9.2.1 Creating, copying, and erasing lists

9.2.1.1 Creating lists

We know how to create a list statically, in the recognition system (7.2), with parentheses. A left parenthesis begins construction of a list, and a right parenthesis takes everything that it finds on Z after the left parenthesis and makes a list out of it. The very same method may be used inside a program, too. Left and right parentheses are perfectly ordinary programs, which may be referenced in a program list. The only catch is that they are active symbols, which means that they must be prefixed with the single-quote 'deactivate' operator when they are entered into the program list. Consider the following example:

```
TEST2: ( '( X Y Z ') )
```

Each time 'TEST2' is executed, it will create a list containing X, Y, and Z. The address of the list will be left on Z.

There is a certain fairly obscure hazard to using '(' to create a list during

the execution of a program. Referring to section 7.2 and its discussion of how the colon operator works in assigning a name to a structure, recall that the bulk of the processing of the colon is performed by the left paren. If one executes a left parenthesis inside a program while there is a colon outstanding, then the colon will be 'eaten up', quite to the surprise of the user. If that sentence didn't make sense to you, please don't worry about it right now, it is really quite subtle. The only reason we mention it at all is to explain why there exist multiple routines to perform what seems like the same task. There is a special routine named '\(' whose purpose is to begin the creation of lists inside of programs without disturbing outstanding colons. It takes a type symbol as input, as (T/L '\(, , , ')). There are several composite routines which call \(, including L\(, P\(, and (L*(I) only) AL\(. Their action is to begin an unnamed list of the requested type without disturbing an outstanding colon.

9.2.1.2 Copying lists

There are several routines which copy lists. Which one to use depends on what we mean by 'copying' a list. Minimally, the act of copying a list means to create another list of the same length, each of whose symbols is the same as that of the copied list. The routine CL copies a list:

```
L\(A B C 1 2 3) CL!
ZCXRGL  CL!
```

The first example will cause a 6-element list to be created, with elements A, B, C, 1, 2, and 3. Since only symbols are being copied, and not the structure of the symbols, the 1, 2, and 3 referenced in the copy list will be the same symbol as in the source list. Similarly, if we copy a list which has a sublist:

```
L\(A B (X Y Z) C )  CL!
```

then the sublist will not be copied: the new list will reference the same (X Y Z) sublist as did the source.

Sometimes we want to copy the structure of a list, too; that is, copy not only its symbols, but the data structures which they represent. We probably don't want to copy all of the named structures, so CLX, copy list structure, has the following behavior: It will copy each symbol on its input list. If the symbol is unnamed, and is T/M, T/P, T/L, T/W, or T/KS, then the structure of the symbol will be copied, and the copy list will contain the symbol of the copied structure rather than the symbol of the original structure. The association ATCLXE associates a structure copy routine or a NOP from each type. The structure-copy routine for T/L is CLX, so that it will recursively copy all sublists in copying a structure.

In L*(I), data and program lists have the same structure, so either may be copied with CL and CLX. In L*C.(C), program lists have a different format, hence there must exist different copy routines: CY and CYX are the corresponding routines for L*C.(C) compacted program lists.

Because the CLX routine tests whether symbols are named, it is quite slow, as this test requires an elaborate lookup. The same considerations hold for CYX, ELX, etc. It is not a good idea to use these recursive structure-copy routines in code that must be fast.

Two special routines exist for copying lists onto Z. They are active symbols; their effect is to place on Z all of the symbols in a list:

←L Empty list (0) onto Z

←Y Empty compacted list (0) onto Z

9.2.1.3 Erasing lists

In very much the same way that CL and CLX copy a list and a list structure, the routines EL and ELX erase a list and a list structure. Like CLX, ELX is fairly slow because it must check each symbol to see if it is named. As might be expected, there are separate erase routines for compacted lists in L*C.(C); they are EY to erase a compacted list and EYX to erase a compacted list structure.

9.2.2 Examining and modifying list structures

Numerous support routines exist to examine and modify the contents of L* lists. For purposes of description, we will divide them into two categories: routines to examine or search a list and routines to modify a list. The bracket notation ([11], etc.) used in these descriptions is described in section 3.2.1.

9.2.2.1 Examining and searching lists

S (args: [11]) gets the symbol of its input list. That means that it accepts a list symbol as input (from Z), and outputs to Z the symbol of the head of the list. Thus:

```
L\(A B C)  S!        outputs 'A'
L\TEMP : (NIL)       outputs 'NIL'
```

N (args: [11]) gets the next of its input list. This means that it accepts a list symbol as input (from Z), and outputs to Z the symbol of the next cell in the list. For example:

```
L\(A B C) N!         outputs '(B C)'
L\(A B C) N! S!      outputs 'B'
```

F (args: [1V8]) tests whether its input list is empty. If F is given the symbol of a termination cell of a list (i.e. one whose next is NIL), then it will signal false and produce no output. If F is given any other list symbol as input, it will signal true and output its input. Thus:

```
L\(A B C) F!      outputs (A B C) and signals true
L\(C) N! F!       outputs nothing, signals false
```

LE (args: [11S]) locates the end (i.e. the last symbol) of its input list. If the input list is not empty, then LE will signal true and return the last symbol in the list (which will look like a 1-element list). If the input list is empty, then LE will output its input and signal false. For example:

```
L\(A B C) LE!     will output (C) and signal true
L\() LE!          will output () and signal false
```

LTC (args: [11]) locates the termination cell of its input list. If a list is non-empty, then the sequence (LE N) is equivalent to LTC.

LST (args: [11]) locates the starting cell of its input list. The symbol of the termination cell of of list is the symbol of the head of that list, hence LST is equivalent to (LTC S).

LSL (args: [21S]) locates a symbol on a list. Given a list (0) and a symbol (1), it searches the list for a cell containing that symbol. If found, it outputs the symbol of the list cell and signals true. If not found, it outputs the symbol of the termination cell and signals false. Compare with FSL, below.

FSL (args: [2VS]) finds a symbol on a list. Given a list (0) and a symbol (1), it searches the list for a cell containing that symbol. If found, it outputs the symbol of the list cell and signals true. If not found, it outputs nothing and signals false. Compare with LSL, above.

=SL (args: [20S]) signals whether a symbol is contained in a list. If symbol (1) is found on list (0), then it will signal true; otherwise it will signal false.

#L (args: [21]) counts the length of a list into a word you provide. If (0) is a list and (1) is a word, then the word will be set to the number of cells in the list (exclusive of the termination cell). The word will be output on Z. Thus:

```
W\WRD  L\(1 2) #L!  sets WRD to 2 and outputs it
W\WRD L\() #L!      sets WRD to 0 and outputs it
```

9.2.2.2 Modifying lists

R (args: [20]) replaces the symbol of a list cell. If (0) is a list cell and (1) is any L* symbol, then R causes (1) to become the new symbol of that list cell. Thus:

```
L\LIST: (1 2 3)      define a list
XY LIST R! LIST PR!will show
LIST: (XY 2 3)
```

RN (args: [20]) replaces the next of a list cell. Since it modifies the structure of a list rather than its contents, RN is to be used with caution. If (0) is a list cell and (1) is any L* symbol, then RN will set the next of (0) to be (1). If (1) is not of type list, then the resulting structure will be ill-formed.

I (args: [20]) inserts a symbol into a list. If (0) is a list and (1) is any L* symbol, then I will create a new list cell, link it into the front of list (0) and make its symbol be (1). Thus:

```
L\LIST: (A B C)          define a list
47 LIST I! LIST PR!      execute I, then print the list
LIST: (47 A B C)         notice our symbol at head of list

63 LIST N! N! I!         move down the list and insert 63
LIST PR!                 then print the list
LIST: (47 A B 63 C)      note our symbol in the list
```

IN (args: [20]) inserts a symbol in a list as the next of the head cell. Essentially this amounts to inserting after the input cell rather than before. If (0) is a list and (1) is any L* symbol, then IN will create a list cell, link it into (0) after the current head of (0), and make its contents be (1). Thus:

```
L\LIST: (A B C)          define a list
-8 LIST IN! LIST PR!     perform IN, then print the list
LIST: (A -8 B C)         notice our symbol in list after head cell
```

IE (args: [208]) inserts a symbol at the end of a list. If (0) is a list and (1) is any L* symbol, then IE will create a list cell, link it in at the end of list (0), and cause its symbol to become (1). If the list (0) was previously empty, then IE will signal false; if the list (1) had prior contents, then IE will signal true.

IC (args: [10]) inserts into a list a second copy of the head cell. If (0) is a list, then IC will create a new list cell, link it into the list after the head cell, and make its symbol be the same as the symbol of the head cell. Thus:

```
L\LIST: (A B C)          define a list
LIST IC! LIST PR!        execute IC, then print
LIST: (A A B C)          note second copy of head symbol
```

PI (args: [21]) inserts a symbol into a list, and also leaves that symbol on Z. If (0) is a list and (1) is any L* symbol, then PI will behave exactly like I with the same inputs, except that it will leave (1) on top of Z as output.

D (args: [10]) deletes a cell from a list. If (0) is a list, then D will cause the head cell to be unlinked from that list and erased. The actual mechanism

involved in D is both subtle and important: the cell that D operates on is not the one which is actually erased. Rather, the contents of the next cell are copied into the head cell, then the next cell is unlinked and deleted. This distinction might seem trivial, but it ensures that the address of the head of a list never changes. Tricky. By way of example:

```
L\LIST: (A B C)      define a list
LIST D! LIST PR!     delete the head and print it
LIST: (B C)          notice that the A is gone.
```

SD (args: [11]) deletes a cell from a list, but leaves its symbol on Z. SD is identical to D in every way, except it leaves the symbol of the deleted cell sitting on Z.

DSL (args: [20]) searches a list for a specified symbol, and deletes it if found. If (0) is a list and (1) is any L* symbol, then DSL will search list (0) for symbol (1) and delete it if found. If not found, no signal will be given. DSL is equivalent to (FSL ,- D).

DE (args: [10]) deletes the last cell of its input list. If (0) is a list, then DE will find the last cell of the list and execute a D (delete) on it. DE is equivalent to (LE D). It is not a good idea to execute DE on empty lists, as it will delete their termination cell, and they will no longer be well-formed list structures.

9.3. Stack facility

The L* basic system supports the expected collection of stack operations. They are designed and named in such a way as to be comparable with the list operations, thereby (hopefully) making both sets easier to remember. The bracket notation ([11], etc) used in this section is defined in section 3.2.1.

IJ (args: [20]) pushes (inserts) a symbol onto a stack. If (0) is a stack and (1) is any L* symbol, then IJ will push the symbol onto that stack. Compare with the I operation defined on lists.

DJ (args: [10]) pops (deletes) the top symbol from a stack. If (0) is a stack, then DJ will pop its topmost symbol. Compare with the D operation defined on lists.

SDJ (args: [11]) pops the top symbol from a stack and pushes that symbol onto Z. If (0) is a stack, then SDJ will pop its top symbol and push that symbol onto Z.

SJ (args: [11]) pushes onto Z the topmost symbol of a stack without disturbing that stack. If (0) is a stack, then SJ causes its topmost symbol to be pushed onto Z. Compare with the S operator defined on lists.

RJ (args: [20]) replaces the topmost symbol of a stack without pushing or popping that stack. If (0) is a stack and (1) is any L* symbol, then RJ will cause the topmost symbol of that stack to be changed to equal symbol (1). Compare with the R operator defined on lists.

PIJ (args: [21]) pushes a symbol onto a stack and also leaves that symbol on Z. If (0) is a stack and (1) is any L* symbol, then PIJ will push the symbol (1) onto the stack (0), and also leave the symbol (1) on top of Z. Compare with the PI operator defined on lists.

ICJ (args: [10]) pushes onto a stack a second copy of its topmost symbol. It has the same action on an arbitrary stack that P has on Z. Compare with the IC operator defined on lists.

CLRJ (args: [10]) clears a stack; i.e. resets it to have no contents. All symbols in the stack are popped. They are not erased, only popped. Compare this with the CLRL operator defined on lists.

FJ (args: [1V$]) tests to see if a stack is empty. If (0) is a stack, then FJ will output (0) and signal true if (0) is non-empty, else it will produce no output and signal false if (0) is empty. Compare this with the F operator defined on lists.

9.4. Association and Association List facility

Service routines exist to modify and search association list and association structures. Unlike data types we have discussed thus far, association lists and associations are executable. This means that they have a non-null interpreter action when executed in a program list. Thus, the interpreter for these types is an important service routine, and is included in this enumeration of the service routines.

9.4.1 Association-list manipulation

Association lists are structures of type T/AL. There aren't really very many things to do to an association list. One can make entries, delete entries, and search for entries. Hence these routines exist:

RAL (args: [30]) replaces an association in an association list, or creates one if there was none. If (0) is an association list and (1) and (2) are arbitrary L* symbols, then RAL will cause symbol (1) to be given value (2) in the association list (0). If there was a previous value for symbol (1), it will be replaced by the new one.

DAL (args: [20]) deletes an association from an association list. If (0) is an association list and (1) is any L* symbol, then DAL will delete from (0) any

association from symbol (1). If there was no association from symbol (1), then DAL will have no effect.

.I/AL (args: [2V3]) is the interpreter for T/AL. Its interpreter action is to search association-list (0) for symbol (1). If (1) has a value in (0), then the value will be output on Z and the signal will be set true. If (1) has no value in (0), then nothing will be output and the signal will be set false.

9.4.2 Association structure manipulation

An association structure is a collection of association lists with a hash table 'front end'. The symbol of the hash table is of type T/A. When an entry is made in in association structure, the symbol is hashed into the hash table to find out which of the association lists to use. Similarly, when performing a lookup of a symbol, it is also hashed into the table. This technique allows faster searching of structures in which a large number of symbols have an associated value. A set of service routines exists for T/A structures which is very similar to those for T/AL.

RA (args: [30]) replaces an association an an association structure, or creates one if none was present. If (0) is a T/A structure, and (1) and (2) are arbitrary L* symbols, then RA will cause symbol (1) to be given value (2) in the association structure (0). If a value previously existed for (1) in (0), it will be replaced with the new value provided as (2).

DA (args: [20]) deletes an entry from an association structure. If (0) is an association structure and (1) is any L* symbol, then DA will delete from (0) the association from symbol (1), if any.

.I/A (args: [2V3]) is the interpreter for T/A. Its behavior is identical to that of .I/AL, described in the previous section, save that it operates on T/A rather than T/AL.

9.5. Word facility

The L* basic system supports an 'arbitrary word facility' as an escape mechansim to allow the construction of any data structures with any contents. Currently the arbitrary word facility is used also for arithmetic computation; this is clumsy but workable.

There are several scratch words defined in the basic system for temporary use inside a program. They are calllld W0, W1, and W2.

RW (args: [20]) replaces the contents of a word. If (0) is a word and (1) is a word, then RW will copy the contents of (1) into (0). Thus:

```
3  W\WORD RW!        sets WORD to 3
```

W1 W2 RW! sets W2 ← W1

=W (args: [20$]) tests two words for equality. If word (0) is equal to word (1), then =W will signal true. If word (0) is not equal to word (1), then =W will signal false. In neither case will it leave any output on Z.

<W (args: [20$]) tests if word (0) is less than word (1). If word (0) is less than word (1), then <W will signal true, otherwise it will signal false. Be careful when coding calls to <W; a very frequent coding error is to get the order of its arguments confused.

>W (args: [20$]) tests if word (0) is greater than word (1). If word (0) is greater than word (1), then >W will signal true, otherwise it will signal false.

+W (args: [21]) adds two words and outputs the sum. If (0) is a word and (1) is a word, then (1) will be set to the sum of (1) and (0); i.e. '(1) ← (1) + (0)'. Word (1), containing the sum, will be left on Z.

-W (args: [21]) subtracts two words and outputs the difference. If (0) is a word and (1) is a word, then (1) will be set to the difference of (1) and (0); i.e. '(1) ← (1) - (0)'. Word (1), containing the difference will be left on Z.

*W (args: [21]) multiplies two words and outputs the product. If (0) is a word and (1) is a word, then (1) will be set to the product of (0) and (1). In L*C.(C) this will be an unsigned 16-bit product. Symbol (1) containing the product will be left on Z.

/W (args: [21]) divides two words and outputs the quotient. If (0) is a word and (1) is a word, then /W will set (1) to the quotient of (1) divided by (0). Symbol (1), containing the quotient, will be left on Z. On L*C.(C) this will be an unsigned 16-bit quotient.

/RW (args: [21]) divides two words and outputs the remainder. If (0) is a word and (1) is a word, then /RW will set (1) to the remainder of (1) divided by (0), and output symbol (1) on Z. On L*C.(C) this will be the remainder from an unsigned 16-bit division.

There are numerous other operations defined on words, they tend to be analogues of machine instructions available on the host machine. For example, in L*C.(C) there is a BISW operator which corresponds to the BIS (bit set) machine instruction. Consult appropriate reference documentation for details.

9.6. Character-string facility

L*(I) and L*C.(C) support a character string type, and both provide a minimal number of support routines for manipulating character strings. Users desiring support routines of the level availble in some other string-oriented languages are encouraged to code their own; the routines described in this section should form a basis from which to code support routines.

Remember that string 'constants', as created by the recognition system, are constructed with the S" operator. S" is an active symbol which reads all characters up to but not including the next " character and places them in a character string, which it leaves on Z. Thus, S"Test string" will create a character string and output it on Z.

CR/KS (args: [11]) creates a character string of a specified length. If (0) is a T/W character count, then CR/KS will return on Z the address of a character string (0) characters long. Its initial contents will be undefined.

E/KS (args: [10]) erases a character string. If (0) is a character string, then E/KS will erase it.

&KS (args: [21]) concatenate two character strings. If (0) and (1) are character strings, then &KS will concatenate them and output the concatenated string on Z. Neither input string will be erased.

CKS (args: [11]) copies a character string. If (0) is a character string, then CKS will create a copy and output it on Z, without erasing the original.

=KS (args: [208]) tests two character strings for equality. If (0) and (1) are character strings, then =KS will signal true if they are identical and false if they differ.

CKSW (args: [21]) create a character string from a word in a specified number base. If (0) is a T/W positive integer less than or equal to 10, and if (1) is any L* word, then CKSW will convert word (1) to ASCII according to base (0), and output the created character string on Z.

CWKS (args: [21]) converts a string of digits into an integer according to a specified number base. If (0) is a T/W valid number base and (1) is a character string, then CWKS will create a word whose contents are the integer conversion of string (1) in base (0). The word output by CWKS must be erased when it is no longer needed, as a new word is created for each call on CWKS.

There is a character accumulator in L*, used extensively by the recognition system, which allows the assembly of a string from individual character symbols

without incurring the overhead of repeated concatenation. The character accumulator is a block of storage in the L* kernel with enough room to store 130 characters. The following operations are defined on the character accumulator. Recall from section 8.2 that the .XKS operator will execute a routine over a character string. It is in conjunction with the character accumulator that .XKS is most useful.

ACCKS. (args: [00]) reset the character string accumulator. When executed, ACCKS. clears out the accumulator and prepares it to receive a character in its leftmost character position.

ACCKS (args: [10]) accumulates a character into the character accumulator. If (0) is a T/K symbol, then executing ACCKS causes the character represented by (0) to be concatenated at the right of the character accumulator.

UACCKS (args: [01]) un-accumulates a character from the character accumulator. When executed, UNACCKS causes the rightmost character in the accumulator to be removed, and the T/K symbol for that character to be output on Z.

CACCKS (args: [01]) creates an L* character string from the current contents of the character accumulator. Executing CACCKS will create a character string and output it on Z. This string must be explicitly erased when it is of no further use.

9.7. Block structures

A block is a contiguous group of symbols. The L* basic system supports blocks with a small set of support routines. As uses for block structures arise, it is expected that the user will create his own block manipulation routines from these.

CRBN (args: [21]) creates a block. If (0) is a word and (1) is a type symbol, then CRBN will return on Z the address of a block of (0) words of type (1). Thus:

```
T/W  20  CRBN!     creates a 20-word block of T/W
T/P  17  CRBN!     creates a 17-word block of T/P
```

EBN (args: [20]) erases a block. If (1) is the first symbol of a block (i.e. the symbol returned by CRBN), and (0) is a word count, then EBN will erase the block.

RBN (args: [30]) replaces the contents of a block with that of another. If (1) and (2) are block symbols, and if (0) is a word count, then RBN will rewrite the first (0) words of block (1) with the contents of block (2).

ABN is an attribute (association structure) associating lengths from blocks. It is used only in L*(I). L*C.(C) does not store block lengths, primarily for reasons of space economy.

CRB (args: [21]) creates a block and records its size in the association structure ABN. CRB exists only in L*(I); L*C.(C) users should use CRBN directly.

EB (args: [10]) erases a block whose size is recorded in ABN. If (0) is a block symbol, then EB will erase it according to the size recorded in ABN. If there is no entry in ABN for the input block (0), then the error event 'EEB' will be triggered.

10. The Recognition System

10.1. The EXEC in detail

10.1.1 Character actions and name assembly

10.1.2 Recognition actions

10.1.3 Modifying the user interface

10.2. Names and contexts

10.2.1 Creating and using a context

10.2.2 Defining and redefining basic-system names

10.2.3 Local names

10.2.4 Names within blocks

10.3. Creation type control

10.3.1 Type-control operators

10.3.2 The forward-reference problem

10.4. Pertinent data structures in the recognition system

11. Operating-system interface

11.1. Reading and writing files

11.1.1 I/O interfaces

11.1.2 Character stream I/O

11.1.3 Binary word I/O

11.2. Time accounting

11.3. Sub-jobs

11.4. Parallel processing

11.5. Odds and ends

11.5.1 PDP-10 notes

11.5.2 C.mmp notes

12. Debugging tools

12.1. Breakpoints

12.2. Tracing execution

12.3. Error detection and recovery

15. A program library: existing code

15.1. dummy section

Part IV: Practical L*: hints, tools, and techniques

13. Programming Style

13.1. Source program formatting

13.2. Names

13.3. Technique

INDEX

%, address notation 33
:, colon operator 26
<W, test word inequality 52
=KS, test equality of chr strings 53
=S, test symbol equality 42
=W, test word equality 52
>W, test word inequality 52
[[, block creation operator 30
\(, begin unnamed list 45
]], block creation operator 30
↑Z (control Z), character action 17

!, 'execute' operator 21
!, editor command 28
!B, editor command 28
!D, editor command 28
!F, editor command 28
!FX, editor command 28
!I, editor command 28
!N, editor command 28
!R, editor command 27
!S, editor command 28
!U, editor command 28

$+, set signal true 43
$-, set signal false 43
$STEP, step-control flag 16

&KS, concatenate chr strings 53

' (single-quote) recognition action 21

(, structure-building operator 25
(0), (1), etc: defined 6

), structure-building operator 25, 39

*W, multiply word 52

+SW, increment a symbol 42
+W, add word 52

-SW, decrement a symbol 42
-W, subtract word 52

.+, conditional operator 35
.+H, conditional operator 35
., control operator 35
.-, conditional operator 35
.-H, conditional operator 35
.X, execute control action 39
.H, control operator 35
.I/A, interpreter for T/A 51
.I/AL, interpreter for T/AL 51
.L, .L+, .L-: loop escapes 38
.Q, symbol quote operator 14, 38
.QH, symbol quote operator 38
.U+, control operator 36
.U-, control operator 36
.X, execute control action 39
.XA, iteration control operator 38
.XAL, iteration control operator 38
.XB, iteration control operator 38
.XKS, iteration control operator 38
.XL, iteration control operator 38
.XN, iteration control operator 38

/RW, remainder word 52
/W, divide word 52

:L, block piece naming operator 30

?, structure print operator 32
?ZX, call stack display 8

AACT 21
ABN, attribute for block size 55
ACCKS, accumulate a character 54
ACCKS., reset character accumulator 54
Active symbols 19, 21, 25
Addition of words 52
Address, numeric notation 33
AL\(, begin unnamed T/AL list 45
Arbitrary word structures 13
Argument passing 8
Arrays 13
Ascending, in program list 34
Association lists, building 29
Association structures 13, 14
Association-list structures 13
Associations, building 29
ATCLXE 45
ATI, type attr. for interpreter 16
Attribute lists 13
Attributes 13
Available space 9
Available space list 8, 31

Base context 22
BCX 22
BCX, base context 6
Block storage management 31
Block structure creation 30
Blocks 13
Boundary character 20
Bracket notation defined 5

C$Z, copy signal out 43
C.mmp 15
Character accumulator 53
Character actions 19
CKS, copy character string 53

CKSW, create KS from word 53
CL, copy a list 45
CLRJ, clear stack 50
CLX, copy list structure 45
Colon operator 39
Command language 23
Compacted list, editing considerations 27
Compacted lists 11, 12
Concatenation of strings 53
Conditional boundary character 20
Conditional operators 34
Context list 22
Context, definition of term 6
Control actions 7
Control operators 34
Copying a list 44
Copying lists 45
CR/KS, create a character string 53
CRB, create and register a block 55
CRBN, block creation routine 54
CRBN, symbol creation routine 31
CRCX 22
Creating T/W structures 30
Creation routine, typed 11
Creation type 28
Current level 34
Current recognition context list 22
CWKS, create word from KS 53
CY, copy compacted list 45
CYX, copy compacted list structure 45

D, delete a cell from a list 48
DA, delete from association structure 51
Data lists, building 28
Data structure building 23
Data structures, building 28
Defining a symbol 20
Deleting levels 35
Descending, in program list 34
Design philosophy 3
Division 52
DJ, delete from stack 49
Double-quote character, recognition action 21
DZ0, delete symbol from Z0 43

E, symbol erasure routine 31
E/KS, erase character string 53
EB, erase a recorded block 55
EBN, block erasure routine 54
Editor 10, 27
EJOV 13
EJUF 13
EL*, external language 23, 32
EL, erase a list 46
ELX, erase a list structure 46
Erasing a list 44
Erasing of storage 8
Erasure routine, typed 11
Error detection 17
Error event 17
Error events 17
Error recovery 17
Escape from a loop 38
ESPX, available-space error event 31
ESTEP, interpreter step event 16
EUND/P, undefined program error 17
Event 16
Examining lists 46
Exclamation point 21
EXEC 9, 17, 19, 20, 21, 22, 25
Executable data types 14
Executable types 16
External iteration control 37
External iteration control operator 36
External language 9, 23
EY, erase a compacted list 46
EYX, erase a compacted list structure 46

F, test if list empty 46
FJ, test for stack empty 50
Floor 39
FSL, find symbol on list 47

Garbage collection 8
Generalized breakpoint facility 16

Hash table 51
Hash table, association 14
Higher level 34
History of the L* language 1

I/O interface 23
IC, insert copy 48
ICJ, insert copy onto stack 50
IE, insert symbol at end of list 48
IJ, insert onto stack 49
IN, insert next of list 48
Interface, user 19
Internal iteration control operator 38
Internal language 23
Interpreter 9, 16
Interpreter action 12, 14
Interpreter action by type 9
Interpreter, user-coded 11
IPL-V 1
Iteration control operators 36
IZ0, insert on Z0 43

L\(, begin unnamed T/L list 45
L*, origin of name 1
LE, locate end of list 47
Level 37
Level, definition of term 7
Level, of a list 28

Level, of a program list 34
List manipulation routines 44
Lists, data 12
Lists, data, building 28
Lists, description of structure 12
Lists, program 11
Local name 31
Locate end of list 47
Loop escape 38
LST, locate start of a list 47
LTC, locate termination cell 47

M-FILE 2
MERLIN 1
Modifying lists 47
Multiplication 52

N, get next of list 46
Name and Symbol, difference defined 6
Name assembly, recognition system 19
Name character 20
Name context 19, 22, 26
Name recognition 20
Next of a list, defined 12
NIL 7, 11, 12
NIL, defined 7
Nonexecutable types 16
Notation, definition of 5

Output interface 32
Overflow, stack 13
Overlays 12

P\(, begin unnamed T/P list 45
P$, push signal stack 43
P, push operation on Z 41
P1, push operator on Z 42
Page, C.mmp 12, 31
Parenthesis notation for Z symbols 6
Passive symbols 21, 25
PDP-10 12, 14, 15
PDP-11 12, 14
Percent sign, address suffix 33
PI, push and insert 48
PIJ, push and insert on stack 50
PIZ0, push and insert on Z0 43
PL* 23
Pop, from stack 13
PP, push pair on Z 42
PR%&, print an address 33
PR%, print an address 33
PR, structure print operator 32
PRA 33
PRAL 33
Print facility 2, 23
Print routine, typed 11
Printing a structure 32
Printing addresses 33
Printing by type 33
PRJ& 33
PRL& 33
Program construction 23
Program list 9
Program lists, description of structure 11
Programmer-defined types 11
Programs, creating 25
PRW& 33
PRY& 33
Punctuation as part of names 20
Push, onto stack 13
Pushdown stack 13

Quote operator 38

R, replace symbol of a list cell 47
RA, replace in association structure 51
RAL, replace in association list 50
RBN, block replacement routine 54
RDF, input a source file 24
Recognition of names 20
Recognition system 6, 9
Reference documentation 2
Rigid boundary character 20
RJ, replace in stack 50
RN, replace next of a list cell 48
RW, rewrite word 51

S, get symbol of a list 46
Scratch stacks 43
Script, interactive tutorial 2
SD, delete list cell and push symbol 49
SDJ, get and delelete from stack 49
SDZ0, get and delete symbol from Z0 43
Searching lists 46
Service routines, for structure 11
SETA, create T/A from T/AL 29
Side cell 43
Side signal 16
Signal 37, 43
Signal stack 36
Signal, definition of term 7
Single-quote recognition action 21
SJEXEC, sub-job exec 24
Source files, reading 23
Stack overflow and underflow 13
Stack support primitives 49
Stacks, structure 13
Step event 16
Stepping monitor 2, 7, 10, 16
Storage allocation 8
Structure building operators 29
Structure editor 27
Structure printing 32
Structure-building operators 9

Subtraction 52
Symbol 12, 32
Symbol actions, recognition system 21
Symbol and name, difference defined 6
Symbol of a list, defined 12
Symbol, defining 20
Symbols 22
SZ0, get symbol from Z0 43

T/A 14, 33, 51
T/AL 14, 33, 50, 51
T/AV 32, 33
T/I 12, 33
T/J 13, 33
T/KS 14, 45
T/L 12, 33, 45
T/M 9, 12, 45
T/P 12, 16, 33, 45
T/T 11
T/W 13, 33, 45
Termination cell 11, 12, 47
Test for list empty 46
TRUE, value defined 7
TTI, type table for interpreter 16
Type association 14
Type association-list 14
Type block 31
Type blocks 9
Type character-string 14
Type creation 11
Type creation routines 8
Type indirect 12
Type list 12
Type machine 12
Type map 9
Type program 12
Type stack 13
Type system 8
Type table 16
Type tables 11, 33
Type word 13
Type, block creation 30
Type, printing by 33

U$, pop signal stack 43
U, pop top of Z 41
U1, pop operator on Z 42
UACCKS, unaccumulate character 54
UCX, user context 6
Underflow, stack 13
Unexecutable type 12
User interface 19

V, reverse top 2 symbols of Z 41
Value field, association list 14

Word manipulation operators 51
Word storage allocation 31
Word structures 13
Words, creating 30

Z 5, 9, 13, 14, 19, 21, 25, 28, 31, 44
Z$ 5, 7
Z$, operations on 43
Z$, signal stack 36
Z, main data stack 8
Z, operations on 41
Z0, scratch stack 43
Z1, scratch stack 43
ZBN 33
ZCXCRL, creation context list 22
ZCXRGL, recognition context lists 22
ZQ, operator stack 39
ZX 13, 17, 18
ZX, control stack 8
Zzzzzzzzz : last item in the index 62

[10] etc., notation defined 5

←L, empty list onto Z 46