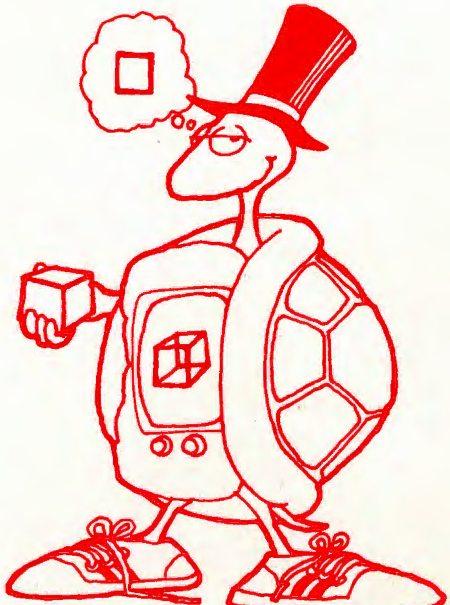
tlc-lisp86



THE lisp™ COMPANY (T.(L.C))

TLC-LISP™ DOCUMENTATION

Primer, Metaphysics,

and

Reference Manual

The LISP Company
POB 487
Redwood Estates, CA 95044

(408) 354-3668

Table of Contents

PREFACE

This document is a much revised version of our original documentation done in 1979. In fact the current TLC-LISP system is a much revised version of the original system, also done in 1979. Furthermore, the field of LISP appreciation is a much revised version of the state of the world in 1979. In short, much has happened in the last five years.

Locally, TLC-LISP and its associated documentation have been revised to include vector objects, packages, a class system, and new input/output protocols including turtle graphics if appropriate hardware is available.

Globally, the world has become aware of artificial intelligence and as a result has become aware of LISP. Combined with this has been a near hysterical fascination with the language Logo, a LISP dialect with a graphics interface. This preface is here to add a bit of perspective on what is happening to LISP-ish things. "Caveat Empty" -- beware of the vacuous.

The high-end energy was focused by the Japanese announcement of their Fifth-Generation Project. Suddenly, people who the week before couldn't spell AI were now becoming experts; programs that did form-filling for the generation of BASIC programs were suddenly "AI based"; P. T. Barnum had arrived. Not to be outdone, the educational community ingested strange turtle-shaped mushrooms and swept Logo into their classrooms as the panacea to "(re)vitalize American education".

Yet beneath all the hype and hysteria lies some substance. Many educators are well-aware that the superficial gloss of turtle graphics has to be reinforced with some substance -- some directed mind-training. Many researchers are aware that the world of expert systems is still at the research level, and that if real progress is to be made, we need to develop new talents that understand the pitfalls of the past. It is with this hopeful note in mind that we continue to develop products and documentation, so that those talents may develop.

In that light we have expanded the functionality of our language; in particular, we have added a class system much like that found in Smalltalk-80 and have added a package system like that found in LISP Machine LISP. This gives the TLC-LISP user the ability to experiment with an elegant descriptive programming tool without having to purchase an exceedingly expensive machine.

Futhermore, we have strengthened the general-purpose base of TLC-LISP, adding vectors as first-class data objects. Besides being a valuable tool for structuring data, vectors are a potent way to demonstrate the importance of first-class data to those who have become accustomed to the weaker notions in other lanuages.

Also, since the quality of hardware has improved in the intervening years, we now include a display-based text editor in the TLC-LISP package.

We view this LISP as a good tool for bridging the gap between traditional programming languages and Fifth Generation languages that will be based on "purer notions" of functional or relational languages. For though the kernel of LISP is functional, several concessions have been made to make LISP more effective on traditional machines (and traditional minds). As sophistication grows, both in the architectures and in the user community, fewer of those concessions are necessary. In TLC-LISP we have retracted some of those compromises, replacing them with new concepts for exploring some of the modern ideas in computing. However, a language is just a tool; as such, it must be used in a context of a set of ideas. Books, support materials, and experience are all sources for those ideas. Our Logo work has much to say about this; the book "Thinking About TLC-Logo" coupled with our TLC-Logo documentation supplies much of the background on LISP-ish ideas. And our (as yet uncompleted) LISP book will place those ideas directly in the TLC-LISP world. Until that work is completed, we will have to depend on other LISP books and this documentation.

* * *

This manual is organized to satisfy the needs of a wide class of readers, ranging from the novice who wants to know that LISP is an acronym for LISt Processing, to the experienced LISP user who only wants to know how this LISP differs from other LISPs.

The table of contents gives a reasonably accurate picture of what each section covers. Since this LISP dialect -- as all other LISP dialects -- presents its own idiosyncracies, it is imperative that all prospective users read Part I, <u>An Introduction to TLC-LISP</u>.

Though this manual does have a collection of examples and catalog of the LISP library, it is not organized as a cook book that can stamp out LISP programmers like chocolate chip cookies. It is unfortunate that no suitable TLC-LISP primer exists yet, though we do plan to provide a self-contained instructional primer for TLC-LISP. In the meantime we will emphasize style and elegance in this manual, leaving your skill with the language to come from your exposure to existing LISP texts and the result of practice with the LISP tools.

On what might seem to be a tangential topic, we reference our Logo book, _Thinking about TLC-Logo_; that book discusses the applications and philosophy behind our version of Logo. The other TLC reference is the TLC-Logo Primer, which contains a blow-by-blow, step-by-step introduction to the contents and style of TLC-Logo. That document is a prototype of the document that will accompany this LISP. That language shares a large cultural base with our LISPs. The distinction between our two languages is concentrated in:

(1) the graphics that Logo tends to thrive upon -- though we make our Logo available without graphics and make our LISP available with graphics.

(2) the syntax wherein Logo replaces the need for parentheses for grouping with the need for visual acuity and stronger human memory to provide accurate grouping in complex expressions. TLC-LOGO is a good introduction to (not a replacement for) TLC-LISP.

This close coupling between our LISP and Logo products stems from the ancestry of the languages. LISP began life in the late 1950s. Logo spun off from LISP in the mid 1960s. At that time, the LISP community was moving from their initial abode on the IBM704 to a new home on the DEC PDP-6. To celebrate their new residence, the MIT LISP folks wrote a new LISP, called MacLISP, that ran on their own operating system, called ITS. When Stanford received their PDP-6, MacLISP was converted to run under the DEC operating system. Several modifications and embellishments were performed and this LISP became LISP 1.6, also known as Stanford LISP.

Stanford LISP was exported to the Irvine campus of the University of California and evolved into UCI LISP. At Irvine it was further modified and enhanced, receiving the editing and debugging packages of a different LISP strain called BBN LISP. BBN LISP soon became known as InterLISP. From UCI LISP we get the LISP variant that appears in "Artificial Intelligence Programming" book. All of these transformations span about ten years.

Meanwhile, the MIT people rewrote MacLISP. The LISP-based tasks at MIT were becoming quite large and the issues of efficient execution could not be ignored. The new implementation, known as BIBOP (Big Bag Of Pages), consolidated about five years experience with the old MacLISP. In this same time span, an MIT group was designing a LISP-like language, called Muddle. It was to be the implementation vehicle for an AI language called

Planner. As it turned out, Muddle became an elegant language in its own right. It has been released and documented as MDL. It contains a consolidation of many ideas that extend the LISP design. Both MDL and the BIBOP version of MacLISP influenced The LISP Company LISP.

Another major influence is the MIT LISP Machine experience. The problems of efficient execution were being compounded by a growing concern with space; LISP-based tasks were swamping the address-space of the PDP-10. So in the mid 1970s, a group at MIT began the design of a "LISP Machine". In conjunction with this hardware effort, the design team developed a refinement of MacLISP called LISP Machine LISP. That language has grown to include generalizations of several of the notions present in Smalltalk -- intellectually, a generalized class system, and cosmetically, a window system of substantial visual flexibility. That machine and its LISP dialect is again a consolidation: this time including architectural considerations in the equation. Progeny of this MIT LISP Machine architecture are now being sold commercially.

Commercialization (and militarization) of AI-related tools has raised the specter of LISP standards. While such efforts make good sense from a commercial and applications perspective, standards have a checkered history at best. They tend to be based on either a lowest common denominator mentality or an ill-chosen technological quirk. Suffice it to say that there are several contenders for a LISP standard and of them, TLC-LISP is most closely allied to the Common LISP camp. Regardless, anyone who understands one dialect of LISP should have no trouble moving from one faction to any other.

This TLC-LISP is a preview of things to come. It represents the initial strand in a sequence of powerful LISP dialects for the next generation of microcomputers. We have consolidated some of the twenty-years experience with LISP 1.5, and later with MDL, Conniver, MacLISP, and the MIT LISP machine, to present a capable and expandable dialect which will allow non-trivial LISP experimentation within the confines of the current processor, while preparing for the more hospitable and lively environment of the new processors. Thus we have avoided many of those features of preceding LISP's which represent anachronisms. Furthermore, the future LISP machine will be a personal general-purpose computing environment. Therefore, we have included a full complement of arithmetic features as well as including the character, string, and vector data types with their associated operations.

We have retained the "dotted-pair" as the basic structured data type of LISP. The major practical benefit of dotted pairs over lists is one of slight storage efficiency. Newer techniques for representing LISP lists have all but erased that advantage.

The benefits of smoother notation, coupled with the easing of storage requirements, combine to suggest lists as the basic data type for LISP.

Even with lists and list operations, LISP's free and open nature is susceptible to abuse. Don't write code like

```
(CONS (CADDAR X) (CDDADR Y))
```

when you mean

```
(PUTON (THUMB X) (FOREHEAD Y))
```

This CADDA...R style of LISP programming is the equivalent of hexadecimal hacking on traditional computers. This style of programming is hard to read, hard to maintain, and downright anti-social. Invent names for the structure-manipulating operations that reflect the semantics of the programming task. For further elaboration of this view see "An Overview of LISP" in the August 1979 BYTE, or see the books, _Anatomy of LISP_ or _Artificial Intelligence Programming_.

This abstract approach to LISP programming is gracefully supported by LISP macros. Macros, added to LISP around 1963, represent an elegant exploitation of the program-data duality of LISP. A similar, but not identical, feature called read macros was invented still later. Both of these features are included in TLC-LISP.

TLC-LISP also acknowledges the progress made in the last twenty years of language design by including more structured forms of iteration than those supplied by LISP 1.5. We have included an extended version of the MACLISP DO for structured iteration, and the CATCH-THROW pair for non-structured program control. We have also included a version of the ancient LISP SELECT-expression, rediscovered by the Algol folks, and called CASE. These explicit control constructs, coupled with LISP's implicit control (call-by-value and recursion) give the programmer a powerful set of tools for structuring solutions to complex problems.

We have supplied a basic stock of operating system hooks, plus a very general class system from which more complex I/O configurations can be defined. The design of input and output systems seems to have a sufficiently rich, but concrete, context that one can build an appreciation for the ideas of classes, instances, and inheritance. This understanding can be nourished by building on experience with another TLC-LISP feature, property lists. The combination of approaches aids in recognizing class-like situations in more abstract settings.

A final technical remark: in anticipation of very large address spaces (and therefore very large projects) we have included the LISP machine-like package system that supports separate name spaces in a monolithic address space. This technique will become of substantial use as the size and number of embedded LISP applications expands.

And a final philospophical note: there are two reasons for studying a computer language--first, to apply that language to solve a set of problems, and second, to study that language as a notation in its own right. The former case is most typical; much like an engineer applies mathematics to solve physical problems, a programmer will gravitate to a particular language because of the kinds of problems that language tends to address. A scientific problem may suggest FORTRAN, while a business application may suggest COBOL. Many applications programmers will go no further with a language than this, except to perhaps curse a particular idiosyncrasy of the language. But such doubting is a hopeful sign; for it may lead one to the deeper, second issue of a (programming) language--a notation for thought.

Fortunately, mathematics predated FORTRAN; otherwise the road to abstract mathematics would have been more difficult than it otherwise was. But, just as one can look at FORTRAN as a corruption of a mathematical notation, it is important to question whether there is a comparable degree of abstraction lurking beneath the surface of computational languages--a mathematical theory of computation. And yet we don't want to simply reverse the path (from FORTRAN back to mathematics, say); we should expect some new insights based on the unique perspective of computation. This is where LISP comes in; its origins are mathematical, but not traditional numerical mathematics; its areas of application are themselves related to thought, and thus the structures that LISP grew up on--lists, symbolic expressions, and objects--offer potentially different paths to abstraction.

It is this second light--the study of notation itself--that we tend to emphasize in our LISP work. Given the choice between the elegance of an idea and compatibility with a "standard", we gravitate toward elegance. For the important points to us at TLC involve the growth of LISP-ish ideas; not the creation of armies of LISP programmers. Thus it is with some "fear and trembling" that we view the rising popularity of LISP, its standardization, and its commercial applications.

The preceding paragraphs should give both the novice and the expert some food for thought. Now it's time to add some substance to our metaphysics. So, dear reader, understand the past, enjoy the present, and anticipate the future!

The LISP Company (T . (L . C))

T L C - L I S P   D O C U M E N T A T I O N

P A R T   I

An Introduction to LISP-Like Ideas

Part I: Introduction

An Introduction to TLC-LISP

LISP is the second oldest higher level programming language, predated only by FORTRAN. The initial implementation effort began in 1958 under the direction of John McCarthy, currently of Stanford University, but at that time of MIT. McCarthy had just become co-founder, with Marvin Minsky, of the MIT Artificial Intelligence Project. One of McCarthy's concerns was the need for a precise notation for expressing problems of Artificial Intelligence. These problems differed from the traditional computational concerns in that they emphasized structural interrelationships, rather than simple numeric quantities. Of course, any non-numeric problem can be reduced to an "equivalent" numeric one; however much of the naturalness of a problem statement and its solution can be lost in the transformation. McCarthy recognized that the representation and manipulation of objects must be handled at a more abstract and primary level. An example will help to put this discussion in perspective.

An early test-bed for these ideas involved the design of algorithms for the manipulation of algebraic expressions; for example, a simplifier might rewrite 2*(x+6*y)+x as 3*(x+4*y). The design of such algorithms involves the solution of two problems:

(1) A representation for algebraic expressions, and

(2) The specification of algorithms to manipulate expressions in that that representation.

The Representation Problem: How do we encode objects in a notation so that the properties that are important to our study are most easily accessed? When dealing with the symbolic manipulation of algebraic expressions the properties most valuable to us are those involving the recognition of the sub-expressions of an expression.

In a traditional language like FORTRAN, we could assign numbers to each component of the expression and then encode the expression as a vector of those numbers. We could, for example, encode the components in ASCII (or more likely BCD in 1959), and then express something like 2*(x+6*y)+x as:

```
[50 42 40 88 43 54 42 89 41 43 88],
```

that is, the ASCII form of a vector of characters:

```
[2  *  (  X  +  6  *  Y  )  +  X]
```

We would soon discover that our algorithm spends most of its time trying to recover the components of the expressions: "in 2*(x+6y)+x, what is the second operand of *, please?"

Clearly, this problem requires a representation that makes the interrelationships more apparent. That is one of the strengths of LISP's notion of Symbolic Expressions. We will discuss Symbolic Expressions and their representations in more detail later. For now, we will confine our attention to their application.

We choose to represent algebraic expressions as a special kind of Symbolic Expression called a "list". A list contains zero or more elements. An element of a list may also be a list, or it may be an atomic element. For the purposes of this example, an atom is either a number, as in most other programming languages, or may be an object called a "literal atom" or "symbol". Such atomic objects are built out of letters and digits (and perhaps special characters) such that the first character in the symbol is a letter. For example,

```
X       Y       ROCKET      Pierre        MUL-T-2-ED        A23
```

Given symbols and numbers, LISP creates lists by selecting some atomic elements and surrounding them with parentheses. Thus

```
(X Y 2)
```

is a list as is

```
(X 2 Y)
```

However, these are different lists because the elements appear in a different order. List aren't sets of elements, but are finite sequences of elements.

As we mentioned above, elements of a list may themselves be lists. Thus:

```
(X 2 (Y))
```

is a LISP list. As with the previous two examples, this list also has three elements, but in this case the third element is itself a list. This list is also not the same as (X 2 (Y)), because structure (as well as order) are used in determining equality of lists.

We can continue to build up complex collections of lists by applying the simple rules to build new elements. Thus:

```
(ROCKET (Pierre  MUL) (X 2 Y))
```

is a list of three elements, whose last two elements are also lists.

There is a mathematics of Symbolic Expressions having theoretical properties comparable to those of the natural numbers, and yet these same objects have a natural and efficient representation on traditional computers. This elegant blend of cultures -- the practical and the theoretical -- is one of the unique features of LISP. However, our main concern now is their effective exploitation in the solution of complex problems.

How can we use these list objects to represent algebraic expressions? For example, we could represent the expression 6*y as a list (MUL 6 Y) where we write MUL and Y as the representation of * and y, respectively. This representation chooses to make the first element of the list represent the operation (multiplication) and the remainder of the list represents the operands. Continuing the encoding process, the expression x+6*y would be represented as

```
(ADD X (MUL 6 Y))
```

Note that the notation still makes clear which components are operations and which are operands. Applying these techniques to our example, 2*(x+6*y)+x, we can represent it as:

```
(ADD (MUL 2 (ADD X (MUL 6 Y)) X)
```

The notation is simplicity itself. The first element of each list always represents an operation. The elements in the remainder of the list are either lists themselves, in which case they represent complex subexpressions, or they are numbers or identifiers, in which case they represent either numbers or variables of the original expression. Given this representation, it makes sense to ask questions like:

* "Is the expression a product?" Yes, if the first element of the list is the symbol MUL.

* "What is the first operand of the expression? It's the second element of the list.

So we proceed to our algorithm.

Design The Algorithm: Given a representation for the objects, how do we write an algorithm that encodes the phenomenon we wish to examine. The notion of algorithm transcends any specific programming language. That is, we should conceive our algorithm in an atmosphere that is as free as possible from syntactic and implementation considerations. At this level our thoughts should not be constrained by the stylistic anachronisms of a particular language. As our problem domains become more complex, this freedom becomes even more critical.

Speaking abstractly, an algorithm consists of two separate components: the logic which embodies the interrelationships between the elements in the problem, and the control component which specifies how the elements are used. Put another way, the logic component encodes the knowledge, while the control component contains the techniques for applying that knowledge.

The control structures tend to complement the data structures since the flow of control is often based on the structure of the data. The LISP control constructs that we need for our algebraic simplification problem are: the conditional expression and recursion.

The LISP conditional expression is analogous to the "if-then-else" construct of other languages, and is used similarly in LISP to indicate appropriate alternatives depending on the structure of an object. Returning to the current example, we can expect the terms of a polynomial to be either a variable, a constant, or a product of variables and constants. A constant or a variable is already in its most simplified form; but certain kinds of products can be simplified. In particular, the product of anything and 0 is 0. So we'd have to check:

"If a term is a product and either of its operands is 0, then we can simplify it to 0."

We could express this in TLC-LISP as:

```
(COND ((and (is-product term)
            (or (= (first-operand term) 0))
                (= (second-operand term) 0)))  0) )
```

With a little persistence (and a few leaps of faith) this notation can be recognized as a stylized translation of our statement about products. Notice too that the form of the LISP code is similar to that we selected for algebraic expressions: the operation preceding the operands. Thus, instead of

```
   x = y
```

we write

```
  (= x y).
```

So LISP's code looks like LISP's lists. This takes a bit of getting used to, but we'll see that this regularity will pay handsome dividends. Indeed, there's a regularity and simplicity within LISP that can encourage elegance. One testimonial to that is the conditional expression, itself. Many non-numeric problems involve a sequence of tests on an object and, finding one that is successful, we perform a particular computation. Most if-then-else operations expect the user to express this condition as a nested construction:

```
if variant-1? then expression-1
              else if variant-2? then expression-2
                                 else if ....
```

If the number of variations gets large, the nesting of if-expressions becomes quite deep and results in obscure code. The form that if-then-else takes may stem from its origins in numerical problems where the number of choices tends to be small (0, positive, or negative, for example). LISP, on the other hand, grew up expecting to deal with richly structured data. Thus a comparable LISP conditional is the following:

```
     (COND
       (variant-1? expression-1)
       (variant-2? expression-2)
       ...
       (variant-n? expression-n) )
```

where expression-i will be evaluated just in the case that variant-i? is true and no variant-j? is true for j less than i. This is a much more pleasing format for complex situations.

This conditional expression is LISP's mainstay for explicit control of computations. LISP uses two implicit control operations as well: recursion and call-by-value. We will address the issue of call-by-value as we proceed.

An application of recursion is appropriate when the solution to the original problem can be expressed in terms of a similar solution to subproblems. For example "the simplified form of an expression, e + 0, is the simplified form of the expression e". Here, the process involves the application of algebraic rules in the context of the informal notion of "simplification".

The algorithm will involve the manipulation of lists that represent algebraic expressions. For example, the simplification rule that expresses the property that x+0 or 0+x is x, for any x can be described informally as:

"if either summand is zero then the sum is equal to the other summand."

In LISP we could test for the occurrence of a sum by

```
(IS-SUM TERM)
```

and write the above simplification rule as:

```
(cond ((is-zero (first-operand term))  (second-operand term) )
      ((is-zero (second-operand term)) (first-operand term) )
      (T term) ) )
```

where T represents true and is used here as an "otherwise" or "else" condition.

Of course, before we can run any such program fragments we must construct definitions for all these sub-functions and we must give definitions of the data structures in terms of the LISP list structure. LISP does not supply any built-in data definition facilities, neither does LISP impose a "type structure" a la Pascal, with the corresponding declarations and assorted accoutrements. LISP leaves such discipline to the intellect of the user. Such a course places a certain burden on the conscientiousness of the LISP programmer. One should view LISP as an assembly language on which users may impose their own idiosyncratic systems. Therefore only minimal constraints are to be found within LISP.

Operations like IS-SUM and FIRST-OPERAND (called recognizers and selectors respectively) are a part of the specification (logic) of the data type "algebraic expression". In general, a data type specification contains at least three types of operations: the recognizers are used to test for the occurrence of an element of the type; the selectors are used to select components of an appropriate type, and a constructor is used to make a new element of the desired type. Data type specifications in LISP are handled through these constructors, selectors, and recognizers.

In LISP, data items have an associated type, while variables are type-free, meaning a variable may have values of any type, associated with it in a totally dynamic way. This means, for example, that a variable may have an integer value associated with it at one moment, and later in the same program that variable might be used to name a list value or even a function value.

One of the most distinctive features of LISP is its representation of programs as data items. For example, if we had values 3 and 2 associated with X and Y, respectively, we could evaluate the list (ADD X (MUL 6 Y)) receiving the value 15. This duality of program and data is more than an historical anomaly; it is more than an expediency based on the lack of available character sets to support an Algol-like syntax for LISP. It is an important ingredient in any application that expects to manipulate existing programs or construct new programs. Such applications include editors, debuggers, program transformation systems, as well as symbolic mathematics systems and Artificial Intelligence applications (a system that learns must be expected to change its behavior).

We'll not get into a discussion of the merits or feasibility of AI; rather, we stay in the (slightly) less emotionally charged area of interactive systems design. The critical ingredient here is how to handle debugging and modification of a program "on-the-fly". That is, we cannot afford to stop the program, rebuild parts of it and restart the computation. The system may be running a real-time or life-threatening system, or it may be that the intermediate computation has taken a substantial time to develop and the changes that are desired will invalidate only a small segment of that computation. The point here is to envision a situation where the traditional edit, re-compile, link and load model will not suffice. In such a case we have to modify existing programs and data. In such a situation we need to have access to a form of the program that we can manipulate not as instructions, but as data. Those who have had experience with traditional machine-level debugging will recognize this type of situation. LISP is one of the few other "machines" that will support such "on-the-fly" modification.

We can build what are called "structure editors" that will let us examine and modify the list-structure representation of the programs. We can build LISP debuggers that can interact with such structure editors to do the necessary detective work to isolate problems. This latter situation involves more than just having programs available as data, it involves having the dynamic state of the "LISP machine" available for exploration and modification. Such situations border on the introspective.

There are other, less dramatic, situations where an "intelligent" form of programs is useful. Even in the more traditional situation of editing a textual form (rather than structural form) of program, we should be able to recover the source form. Compare this with the typical compiler-based situation wherein there is a wide gap between the running code and the user's source; this situation is exacerbated by optimizing compilers that widen that chasm beyond recovery.

In contrast, even those LISP's that do not deal directly with the structure will allow recovery of the source from the running code. The simplest tactic is to print the list representation into an edit buffer, edit that text, and then pass the new text through the LISP reader. A more substantial task involves the partial compilation of the source into an intermediate code, which is "reversible" in the sense that it can be de-compiled into the user's code. These latter two solutions are partial implementations of the more comprehensive solution of making the program and the dynamic state of the computation (called the control structure) directly available as data objects in the language.

Program transformation systems are another example of treating program as data. Such systems span the spectrum from traditional compilers to source-to-source program improving systems. In its general form, a compiler expects a program as input and produces a program for another machine as output. Again, if the language supports programs as data objects, this compiler can be expressed in the language. Most other languages obscure the problem by describing the compiler as a program which takes a string as input, converts the string to an internal non-executable form, and produces another string as output. This is a very localized view of the world of computing. A healthier approach views compilation as the last phase of the program construction process where the compiler is to transform a correct program into one that will execute more rapidly. Earlier phases of the programming process are responsible for the construction, debugging, and modification of the program. The unifying perspective of a program as a data structure cleanses the intellectual palate.

We will also see another program transformation technique in a few sections hence: LISP macros. Macros in LISP function similarly to macros in assembly language, allowing us to abbreviate sequences of code that the language processor can expand for us automatically. In assembly language the macros are expanded when the source code is transformed into machine code. LISP tends to operate in a slightly different manner, expanding the macro to new LISP code dynamically, and then executing that code. In compiler-based LISPs the analogy to assembly language is exact: the code gets expanded once, and compiled; even without a compiler, we can indicate that we want the macros "displaced", meaning that they expand once and the expanded code actually replaces the application of the macro.

In summary, LISP is best thought of as a "high level assembly language for complex programming". It contains a library of operations, including the system-level components like symbol tables, scanners, parsers, and unparsers, with a processing unit to evaluate the combinations of these ingredients. Yet it imposes little structure on the programming process, believing that discipline is best left to the intelligence of the programmer. LISP is a tool, no better or worse than its user. One goal of this documentation is to develop and reinforce an appreciation for self-discipline as well as reveal the elegance and beauty of LISP.

Data Objects in TLC-LISP

Introduction

This section begins a more thorough and detailed treatment of LISP data. As we have seen, LISP data comes in at least two flavors: atomic objects and composite objects. Atomic objects are further divisible into numeric and non-numeric objects. The most interesting non-numeric atomic object in TLC-LISP is called a symbol or literal atom. Symbols are a versatile naming structure for LISP. They are used as non-numeric constants of the programming language (T and NIL), as primitive data objects (MUL, ADD, and the variable names in the previously discussed algebraic examples), and as references for all the programming language constructs (as in COND, EQ, and IS-ZERO). Furthermore, as we will see momentarily, symbols can also be used to capture collections of data using a symbol like a name in a dictionary.

Many LISP implementations (including TLC-LISP) supply character and string data types. This allows the manipulation of character sequences and, with conversion programs, allows the dynamic generation of new symbols just as numeric operators can introduce new numbers into the programming environment. This dynamic creation of data objects is a hallmark of LISP's attitude toward data objects. The idea, called "first class treatment of objects", stems from a desire to free the programming notation from the details of its implementation. We will dedicate a section to the implications of "first-class-ness" after we have had some experience with the specifics. The size of objects and the duration of their "lifetime" (their creation and deletion) are two such specifics. In elementary mathematics, we deal with integers and "create new ones from old ones" by the successor operation (from n, generate n+1). Programming languages will usually restrict the implementation of the successor operation to those integers that will fit some hardware notion of "largest integer". Strings and vectors are typically constrained in a comparably arbitrary fashion. Not so in LISP. LISP's most general "first class" object type is also its first and most unique data object--the dotted pair.

Dotted Pairs

One characteristic of LISP is its ability to take two existing objects and build a new composite structure from them. Thus given the symbols ROCKET and PIERRE we can create two new structures

```
(ROCKET . PIERRE)    and  (PIERRE . ROCKET)
```

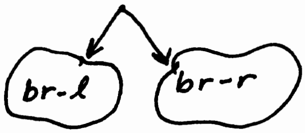
Since this construction operation can be repeatedly applied to pre-existing objects, we can define quite complex structured objects, as for example:

((ROCKET . PIERRE) . (1 . (V . X)))

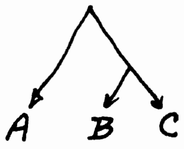
The objects created in this way always have exactly two parts, and those parts are separated by a dot (.) and are enclosed in parentheses. Such objects are called "dotted pairs."

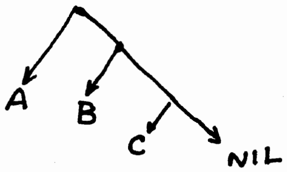
It is sometimes helpful to visualize these dotted pairs graphically. We can view this dot notation as a linear representation for a stylized tree-like structure that always has exactly two branches. Pictorially, it represents a tree with left and right branches br-l and br-r respectively, as

(br-l . br-r)



We can use this graphical representation to display arbitrarily complex dotted-pairs. For example:





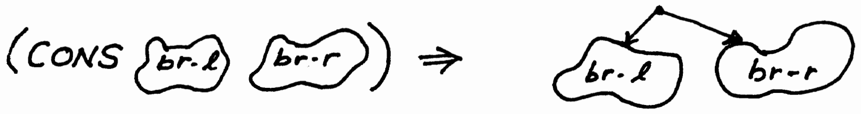
(A . (B . C))   (A . (B . (C . NIL)))

The most general form of these binary trees are called Symbolic Expressions, S-expressions, or S-exprs for short. Though we'll add more kinds of objects in a few moments, we've now been exposed to the fundamental notions of LISP's original set of data: numbers, symbols, and dotted pairs.

What we have so far, however, is just a representation for constants. Before we have anything resembling a language we need operations that will let us build new objects, test them and manipulate them in various ways, similar to the ways that traditional languages let us make new numbers (using +, *, etc.) and test them (is a number equal to 0?). We'll assume we have arithmetic operations, so that we can build new numbers. Later we'll show how to build new symbols, but now we want to introduce ways of building new dotted pairs.

Constructors

Our primary constructor for dotted pairs is called CONS. It expects two objects as arguments (and is therefore called a binary function) it constructs a new dotted pair that, if interpreted as a binary tree, has the first argument of CONS as the left branch and the second argument as the right branch:



where the symbol => means "evaluates to" or "simplifies to".

Besides the graphical form, we need a textual representation. Following the tradition we've established earlier, we'll indicate the application of a LISP function by writing it as a list whose first element is the function and whose succeeding elements are the arguments. Thus:

```
(CONS 1 2) =>  (1 . 2)
```

We can build up such function applications using functional composition. Thus:

```
(CONS (CONS 1 2) 3) => ((1 . 2) . 3)
```

and

```
(CONS 1 (CONS 2 3)) => (1 . (2 . 3))
```

LISP's functional composition is a powerful programming technique, directly derived from the mathematical notion of the same name that we experienced in high school algebra. There are two distinctions: one deep and one superficial. Superficially, in algebra we write x+y and thus might expect to write x CONS y, rather than (CONS x y). More substantially, we will tend to think of LISP's expressions as computational requests rather than as abstract functional expressions. However one very important benefit of LISP's functional basis is the ability to intuit expressions computationally, and simultaneously view them as denoting an object in the sense of a functional mathematics. It is this latter sense that opens new doors for multi-processor, parallel architectures for LISP evaluation. For viewed functionally, a LISP expression imposes few constraints on how a computation must be performed. A few explicit constraints appear as control structures (DO, FOR, etc.), but implicit constraints (like composition) are subject to re-interpretation (evaluate both arguments to CONS simultaneously, for example). We'll explore several of these topics throughout this first Part.

Recognizers

Besides the constructors, we also need the ability to recognize the occurrence of different types of objects. Currently, we've only introduced numbers, symbols, and dotted-pairs, but soon a whole flood of types will appear.

These operations are called recognizers and, in the case of S-expressions, require that we be able to tell an atom from a non-atom. Thus:

```
(ATOM 1)  =>  T
```

where the LISP symbol T is an indication of truth.

```
(ATOM T)  =>  T
```

False, the other truth-value, is represented by the symbol NIL. So:

```
(ATOM (CONS 1 2)) => NIL
```

since the result of (CONS 1 2) is a non-atomic object.

This last example also shows that these truth-valued functions (called predicates) can be composed with other functions. So:

```
(CONS 1 (ATOM 1))  =>  (1 . T)
```

Given that we have an atomic quantity, we need to discover what kind of atomic object it is.

```
(SYMBOLP 1)  =>  NIL

(NUMBERP 23)  =>  T
```

Of course we have to represent tests like:

"Is the atom A a symbol?"   Answer: T

We might try writing (SYMBOLP A)

but we'd discover that

```
(SYMBOLP A)  =>  Error, Unbound-atom A
```

We have to look more closely at the process that LISP performs in evaluating these expressions. Clearly, LISP knows how to recognize constants. If you ask the value of 1, or NIL, for example, the system will respond with 1, or NIL. But what is the value of SYMBOLP (or A)? SYMBOLP's value is a primitive function, and A's value? Well unless we give it one, A has no value. Before we do that, we need to introduce a way of saying:

"Don't look for a value, but take this symbol as it stands"

Natural languages have such a device, called quoting. So for example, we can say:

"Chicago is a city, but 'Chicago' is a seven-letter word."

LISP has a similar solution for the similar problem. When we want to talk about a specific symbol, we quote it by writing

'A, for example.

So we can now express "Is the atom A a symbol?" as:

```
(SYMBOLP 'A)    =>  T
```

The reason for quoting in LISP is to stop evaluation. We must be able to specify the use of a symbol and also the mention of a symbol, and we have to have unambiguous ways of designating these two uses. So we can say

```
(SYMBOLP 'SYMBOLP)  =>  T
```

and know that the first reference is to the function SYMBOLP (a use) and the second reference is to the symbol SYMBOLP (a mention). You might think about what (SYMBOLP SYMBOLP) means.

This problem of quoting does not occur in traditional computing languages because the distinction between data elements (numbers, strings, etc.) and components of the language (symbols--known as identifiers in traditional languages) is enforced by the language.

Since there is no ambiguous use of numbers in LISP, we don't quote them; and since T and NIL are used as Boolean constants, we don't have to quote them. But quoting must be done in any situation that requires that the symbol be taken literally.

```
(CONS 'CONS 'CONS)  =>  (CONS . CONS)
```

Furthermore, in preparation for things to come, we require that dotted pairs be quoted even though the syntax makes it clear that the pair is a data object. Thus:

```
(ATOM '(A . B))  =>  NIL

(CONS '(1 . 3) 2)  => ((1 . 3) . 2)
```

If we neglected the quote-mark and wrote

```
(ATOM (A . B))
```

any LISP evaluator would respond with an error message since it will try to evaluate the pair (A . B) as a function (A) followed by some arguments. If A has no function definition, the system will complain here; if A does have a definition, then the evaluator will complain when it sees B, since (A . B) does not look like an expression.

One final point about quoting (') needs to be made: we advertised that every expression in LISP had to be a list, but

```
'<expression>
```

does not seem to follow the rule. In actuality, the quote-mark is an abbreviation for a LISP-style list of the form

```
(QUOTE <expression>)
```

So

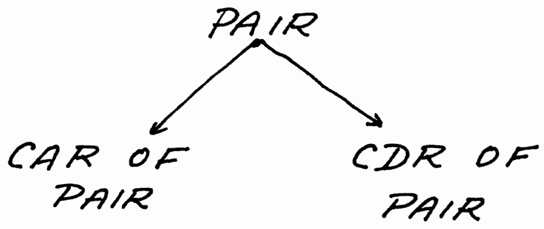
```
'(1 2 3 4)   is really (QUOTE (1 2 3 4))
```

We will soon see what this regularity of notation gains us.

Selectors

Besides being able to construct new objects, we must also be able to examine the components of such constructed objects. Recall that we assumed the existence of such "decomposition" operations in our algebraic simplification example (FIRST-OPERAND and SECOND-OPERAND, in particular). Operations that allow such examination of components are called selector functions.

Since dotted pairs have exactly two components, we have two selectors for S-expressions: one to select the left branch of a tree, called CAR, and one to select the right branch, called CDR.



Given that an object is a non-atomic S-expression, we then know it is safe to operate on it with CAR and/or CDR. Conversely it is not safe to operate on (most) atoms with these selectors.

```
(CAR 1)  =>  error, list-expected.


(SYMBOLP (CAR '(A . B)))  => T
```

since

```
(CAR '(A .  B))  => A,  and A is a symbol.
```

So for example:

```
(CAR '(1 . 2))  => 1

(CDR '(CONS . 2))  => 2

(CDR (CONS 1 (ADD 1 2)))  =>  3
```

An historical note: the names CAR and CDR are derived from the machine representation of the first implementation of LISP. This was done on an IBM704. That machine --a micro in terms of the capabilities of today's hardware-- had its 36-bit word divided into several subfields. Two of those field were the "address field (15 bits) and the "decrement" field (also 15 bits); those fields were used to encode the CAR-branch and the CDR-branch, respectively.

So to summarize the current situation, we now have a simple TLC-LISP calculator at our disposal. It can do arithmetic operations--type in an expression and get its value, as in

```
(ADD 4 5)  => 9
```

It's therefore a functional calculator in the sense that the basic building blocks are functions, which are applied to arguments, and produce values. Thus:

* Every expression has a value; there is no such thing as a "procedure" that has no value.

* Operations can be "cascaded" using the notion of functional composition--the result of one operation can be fed directly into another operation, as in:

```
(ADD (MUL 2 4) (SUB1 5))  =>  12
```

Furthermore, we now have Symbolic Expressions that have a mathematics all their own, and have a set of calculator primitives to construct, select, and test these objects we call S-exprs.

Finally, we're still not at the level of a programming language, only a fancy calculator. For though we can test results, we have not yet shown how to express conditional actions or decision-making in the notation. Before we do that, we want to introduce some other data objects of TLC-LISP.

Lists

Now that we have the basic set of LISP data and operations under our belt, it's time to look at some further data objects and some more operations on those data. Recalling our earlier discussion of programs that manipulate programs, and noticing that LISP programs are represented as lists, it seems only fair to explore the possibilities of developing the same kinds of functions for lists as we just did for dotted pairs. In particular, constructors, selectors, and recognizers will be most appropriate.

Before doing that, we should make a slightly more formal disclosure of what it means to be a list.

* The empty list, designated by empty parentheses ().

* a non-empty list, having elements that may themselves be lists, or may be symbols, numbers, or perhaps dotted pairs. Such a list is simply designated by writing down the elements in the desired order, separating each with space if ambiguity would arise otherwise, and finally decorating the collection front and back with parentheses. Thus, the following are lists:

```
(1 2 3)

(1 2)

(1 2 (2 3) A)

(CAR 1 2)

(CONS (CAR (QUOTE (A B))) 2)
```

A few remarks: the first, fourth, and fifth examples each contain three elements, while the second contains two, and the third contains four. The second example, (1 2), should not be confused with (1 . 2). The fourth example (CAR 1 2) is a list of three elements that just happens to look like a LISP expression. The final example again is a list that looks like an expression, but in this context is just a list of three elements, whose second element is also a complex list.

A special note to those who have prior traditional programming experience: as we introduce the list-operations in the next sections, relate them (and lists) to operations you've seen. For example, how to lists compare with vectors or arrays? Could you implement lists as vectors or strings? Could you get the notions of functional composition to operate with your implementation?

Constructors for Lists

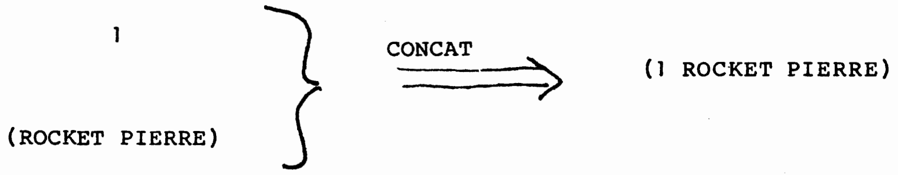
CONCAT is the primary constructor for building up lists. CONCAT, like CONS for S-expressions, is a binary function; however CONCAT expects its second argument to be a list. CONCAT will construct a new list whose first element is CONCAT's first argument, and the remainder of the list is CONCAT's second argument. Thus:

```
(CONCAT 1 ( ))  =>  (1)
```

As with previous LISP operations, we can compose the results of CONCAT:

```
(CONCAT 1 (CONCAT 'ROCKET (CONCAT 'PIERRE ()))) => (1 ROCKET PIERRE)
```

We'd also like to explicity represent, for example, the creation of the list (1 ROCKET PIERRE) by using CONCAT with 1 as first argument and the list (ROCKET PIERRE) as its second argument. The graphical interpretation is clear,



but the textual form is more problematic. For if we try:

```
(CONCAT 1 (ROCKET PIERRE))
```

then, following our convention that an expression is represented as a list whose first element is the function, the previous line must be interpreted as saying:

"Apply the function CONCAT to the arguments 1, and the result of applying the function ROCKET to PIERRE."

But that's not what we had in mind.

Since we can have lists that represent function application, mixed in with lists that represent data objects, we need to have some way of distinguishing between the two. Our quoting problem has returned; but the same solution will hold in this case.

```
(CONCAT 1 '(ROCKET PIERRE))  =>  (1 ROCKET PIERRE)
```

For a more complex situation to highlight the quoting distinction, compare the following two expressions:

```
(CONCAT 1 (CONCAT 1 '(2)))   and  (CONCAT 1 '(CONCAT 1 '(2)))
```

The first one yields (1 1 2)

since the inner CONCAT creates the list (1 2) that is what gets passed to the outer CONCAT.

The second situation is quite different. Recall that it is an abbreviation for

```
(CONCAT 1 (QUOTE (CONCAT 1 (QUOTE (2)))))
```

Because of the QUOTE, the second argument to the CONCAT is a constant--the list,

```
(CONCAT  1 (QUOTE (2)))
```

So the value of the second example is the list:

```
(1 CONCAT 1 (QUOTE (2)))
```

Another important constructor for lists is LIST. This function will take an arbitrary number of objects as arguments and build a list with those objects as elements. Thus:

```
(LIST (ADD1 2) (ADD 3 5) (CONS 1 2) (CONCAT 1 '(3 A)))

               => (3 8 (1 . 2) (1 3 A))
```

The final constructor we'll highlight here is APPEND. It takes two arguments, both lists, and builds a new list with the elements of the first list pasted onto the second list. Thus:

```
(APPEND '(1 2 3) '(A S D F))  =>  (1 2 3 A S D F)
```

In the Functions section, we'll derive the TLC-LISP function for APPEND.

Each of these constructors (CONCAT, LIST, and APPEND) will prove useful; each performs a (sometimes subtly) different function. So, continuing the latest example:

```
(CONCAT '(1 2 3) '(A S D F))  =>  ((1 2 3) A S D F)

(LIST '(1 2 3) '(A S D F))  =>  ((1 2 3) (A S D F))
```

List Selectors

As with dotted pairs, lists also have their own set of selector functions. The basic selectors are called FIRST and REST, and select (respectively) the first element of a list and all of a list but the first element.

Thus:

```
(FIRST '(A S D F))  =>  A

(REST '(A S D F))  =>  (S D F)
```

REST actually can be used more generally: if a second argument is supplied, it indicates how far down the list we should go. Thus:

```
(REST '(A B C D E F) 2)  =>  (C D E F)
```

So (REST '(A S D F)) is the same as (REST '(A S D F) 1)

We'll see several other TLC-LISP functions that have this ability to take a variable number of arguments, called "optional parameters". Later we'll show how user-defined functions can also capitalize on this feature.

TLC-LISP also supplies a generalization based on FIRST, called NTH.

```
(NTH '(Q W E A S D) 3)  =>  E,
```

showing that NTH gets the element of the list specified by the index. So we could have defined

```
    (FIRST <list>)   to be the same as   (NTH <list> 1)
```

or we could have defined FIRST as we did REST, to take an optional argument. But we didn't; we leave that as an exercise.

As with the list constructors, there are several other selector-type functions available in TLC-LISP; see the Reference Manual for details.

Recognizers

In a fashion analogous to the situation involving dotted pairs, we must have predicates that will tell us the condition of a list-type object. Is a given object a list? LISTP will tell us that.

```
(LISTP 1)  =>  NIL

(LISTP 'A)  =>  NIL

(LISTP '(ABC)) =>  T

(LISTP '())  =>  T
```

As with S-expressions and their selectors, we need some way to tell if a list is non-empty and therefore has a FIRST-part and a REST-part. NULL will do that for us:

```
(NULL '(A S D))  =>  NIL

(NULL (REST '(A)))  => T
```

These few functions represent only the tip of TLC-LISP's built-in iceberg. Examine the Reference Manual and the examples on the files accompanying the LISP and you find many more list-manipulating functions. The major point of this section is to introduce you to the basic operations and the general style that we wish to exercise while writing TLC-LISP programs. In particular, now when we think about the programming task, we should view the data as a collection of rather concrete objects--numbers, dotted pairs, and lists, for example. Each type of object has its own collection of operations for creation, testing, and selection. This triad of operations is more recognizable with lists and pairs than with numbers, but is there nevertheless. Later, this technique will come into stronger play when we build abstract objects out of these concrete objects; when we create algorithms to manipulate real world objects, we'll still use constructors, selectors, and recognizers, but now over chairs, and trees, and even mental structures.

Given these operations on objects, we can then conceive of our programs with some sense of abstraction and implementation independence. We deal with the objects through this interface, passing objects between functions using the functional composition pipeline, the result of one computation feeding directly into the next. Unfortunately, the computational environment is not very robust; all we have to work with are the primitive functions and constant data objects. We need to expand this simple but elegant view of computation, retaining the functional flow, while adding an ability to define new functions and name objects. Fortunately, these tasks are related.

Names, Values, Objects, and Aliases

So far we have steadfastly adhered to the calculator motif: an expression is constructed, then evaluated and so yields an immediate result. There's been no sense of memory and no sense of permanence other than the implication that the primitive functions are somehow known to the LISP calculator. This is a mixed blessing: the language is incredibly simple, but it is also quite limited in power since all we can refer to are constants (numbers, S-expressions, lists, and primitive functions). Two key ingredients are missing:

* we have to add programs to the memory of the LISP machine, increasing its capabilites.

* we have to be able to name (or somehow save) partial results so that we can refer to them in a later calculation.

Both of these problems are symptoms of the same phenomenon--that of giving names to objects so that the language may refer to them later. In the first case, we name functions, in the second case, we name data.

Names and naming conventions are as problematic in computing languages as they have been in natural languages. Though we can pass these concerns off in everyday life as irrelevant musings of philosophical overachievers, we must address the issues in computational languages. The specific computational (and philosophical) problem involves equality: when are two objects equal, and in fact, what does equality mean? This involves the issues of indentical versus indistinguishable objects. We can talk about two objects being indistinguishable: if every test on one yields identical results on the other, but to talk about "identicalness" involves talking about names for objects. This is the issue of synonymity: are two names referring to (or aliases for) the same object?

To put this discussion in more concrete terms, we will introduce a technique for associating a name with an object in LISP. Given an object, say the list (A B), we can associate it with the symbol A using the operation SETQ. Thus:

```
(SETQ A '(A B))
```

All traditional languages have a similar kind of operation called an "assignment statement". A LISPish FORTRAN might say:

```
A = '(A B)
```

while a LISPish ALGOL or Pascal might phrase it as:

```
A := '(A B)
```

The major distinction between a LISP SETQ and another language's assignment statement is that the SETQ is an _expression_, meaning that it has a value (besides accomplishing the assignment). The value of the SETQ is the value assigned to the variable. Thus:

```
(SETQ A '(A B))  =>  (A B)
```

Later we will see ways of exploiting this value, and soon we will see more elegant ways of associating objects with names, but for now let's exploit the situation. Given the ability to name objects, what added perspectives on modern computation can we illustrate? The critical issue is that of "object versus value". And the pathway to understanding that distinction leads through the shades of meaning around the word, "equality".

We have two predicates to deal with equality, EQ and EQUAL:

* EQ -- a predicate that compares two references to objects and will tell if they are referring to the same object.

For example, assuming that X names an object, then

```
(EQ X X)  =>  T  regardless of the object associated with X.
```

* EQUAL -- a predicate that compares two references and tells if the objects are indistinguishable. So

```
(EQUAL X X)  =>  T
```

because any object is indistinguishable from itself. Of course, this discussion is totally vacuous until we explain how we differentiate between identical and indistinguishable objects.

So let's complicate matters. Assume that we have a function (which we do) called COPY that will take an object and "clone" it. That is, make a new object that is a carbon copy of the original. But like any copy, at some level there is a distinction between the copy and the original; EQUAL is unable to make the distinction, but EQ can. To be more concrete:

```
(SETQ X '(A . B))   and (SETQ Y (COPY X))
```

Now (EQUAL X Y) still gives T since the objects are indistinguishable. However (EQ X Y) gives NIL since X and Y are not references to the same object.

This may still appear to be a non-problem for computation; if objects and their clones are indistinguishable why bother with EQ? The practical problem involves objects that change (called "mutable" objects). We say "practical" because in object-based languages like LISP, objects can change.

In contrast, languages like mathematics are "value-based" since objects in these languages, once created, cannot change. Such non-changeable objects are called "immutable" objects. We'll talk later about the actual mechanisms in TLC-LISP that allow us to modify objects, but now we want to emphasize what change does to our language.

Practically, the possibility of change (or mutation) says that we cannot "cheat" on the implementation of operations like COPY by simply sharing a reference to the object, because changes to the original (copy) would be reflected in the copy (original). Notationally, the possibility for change is more fundamental. It makes us think more carefully about the notions of object versus value and begins to separate the static mathematical languages (value-based) from the dynamic languages of computation that are object-based. We will return to this topic several times since it represents an important perspective in modern LISP.

So, in summary, names in LISP should be thought of as naming objects, not as naming values. Some objects act like values in the sense that they are immutable. In particular, numbers are immutable; we won't find 1 suddenly turning into 2. Symbols are also immutable in LISP. But general objects (like dotted pairs and lists) can change, just like the city of Chicago changes--unpredictably, and often.

This ability to modify objects introduces a new category of data functions: besides the constructors, selectors, and recognizers, we also have "updater" functions. Dotted pairs, lists, and in a moment vectors and strings, have updater fuctions available to them. Before we do that we'll consolidate our discussions on lists and dotted pairs.

Comparison of Lists and Dotted Pairs

Lists

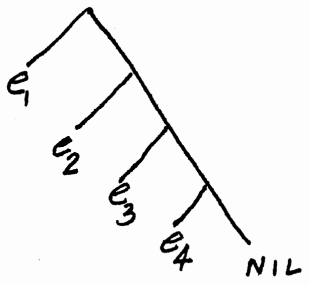
As things stand, we have a small intersection between LISP data and LISP programs; namely symbols are able to migrate across that boundary. So far, that situation seems to be more trouble than it's worth. This section will dispell that opinion by closing the gap between data and program. We will show how to represent lists as dotted pairs, thus mapping all LISP programs and data onto one uniform structure. This is much like traditional machines map program and data onto linear collections of locations that contain either a 0 or a 1. Here we map everything onto binary trees whose tip nodes are atoms (numbers or symbols). We'll exploit this machine analogy further in the Evaluation section. Here we'll concentrate on the details of the mapping.

Recall a list was either empty --denoted by ( )-- or was of the form (e1, ... en) where each ei was either an atom or a list itself.

We may represent list notation as an S-expression by the following rules:

1. Map () onto the symbol NIL

2. Map (e1 e2 ... en) onto
   (e1 . (e2 . (... (en . NIL) ...)))



(e1 e2 e3 e4)

So a list maps onto an S-expression whose right-most (or deepest) node is the symbol NIL. Indeed, within the TLC-LISP system NIL and () are synonyms.

Several other synonyms also appear in LISP courtesy of this mapping. In almost every implementation of LISP, FIRST, REST, and CONCAT are identical to CAR, CDR, and CONS.

However it is good style to program at the dotted pair level using operations based on CAR, CDR, and CONS, and program at the list level using FIRST, REST, and CONCAT.

```
(CONS 1 2) => (1 . 2)
(CAR (CONS 1 2)) => 1

(CONCAT 1 (CONCAT 2 NIL)) => (1 2)
(FIRST (CONCAT 1 (CONCAT 2 NIL))) => 1
```

This sheltering of the dotted pair representation, separating it from the list-related functions, is much more than mere sleight-of-hand. It is our first example of "abstract programming".

By "abstract programming" we mean that we wish to encourage a style of programming that uses data-manipulating operations, without consideration for how these operations are implemented in terms of lower-level constructs. The connection between operations and their implementations is made by simple "interface" specifications, done in terms of the implementation of the ubiquitous constructors, selectors, recognizers, and updaters.

There are several benefits to this programming style: first, programs tend to become small, modular units. This improves readability and maintenance. Second, separation of conception from implementation gives one the freedom to vary the implementation without as much danger of destroying the correctness of the program. All one need do is modify the interface specifications when the lower-level representation is changed. The algorithms above this specification "firewall" need not be changed. This kind of modularity and freedom from implementation is becoming more popular as the complexity and sophistication of programs increases. The idea appears in the guise of "information hiding" and "abstract data structures." Some languages try to enforce these ideas with "strong type structure"; LISP, on the other hand, supplies the tools that a self-disciplined person may use to build such abstractions, but leaves the ultimate responsibility in the hands of the user.

Let's step back a bit from the pecularities of the data representation for a moment, and see what other general lessons we can learn.

Graphical Languages

First, the rather arcane syntax of LISP data--the reams of delicately balanced parentheses--have a method to their madness. They allow us to give a linear representation to a complex tree-structure. This notation was born of historical necessity, since display technology is only now becoming sufficiently proficient at representing graphical information. The thing to remember when viewing LISP's parentheses is that they represent tree-like information, and soon we'll be able to express LISP-like information on display screens in a form more indicative of their true graphical nature.

The deeper lessons of LISP's notation involve its anticipation of more visual, two-dimensional, and pictorial ways of presenting data and programs. Character-at-a-time input of programs and data need not persist now that we have cost-effective means for manipulating graphical information. However, the practical problems of representation, and the theoretical problems of semantics of such languages, are both open issues.

Program as Data

No discussion of LISP data would be complete without a mention of LISP's famous property of "program as data." We've seen that function calls in LISP are represented as lists, whose first element represents a function and whose remaining elements represent actual parameters to the function. We've seen that conditional expressions are represented as lists whose first element is COND and whose remaining elements represent lists of expressions--tests to be made followed by actions to be carried out. Similarly, we'll soon represent function definitions as lists whose leading element indicates that the remaining elements represent name, formal parameters, and body. So all components of our language can be represented as lists. And what does this buy us? It allows us to construct programs that manipulate programs.

A LISP program can access another piece of LISP code to analyse, transform, or even execute it. This is not just a "cute hack". It is a practical advantage born of theoretical beauty. For, editors, debuggers, and compilers are all examples of program manipulation systems. All of these tools are easily written in LISP.

Use, Mention, Object, and Value

One slightly more philosophical point is exemplified in this program-as-data situation; that is the old problem of use versus mention. Whenever a language (natural or otherwise) is strong enough to talk about its own components (words), then precise distinctions need to be made between the use of a name (like the city named Chicago) and the mention of a name (like 'Chicago' has seven letters). Most programming languages don't have to deal with this issue because of the clear division between the programming language and the items that the language manipulates. We don't have that luxury of ignorance in LISP; and QUOTE (or its abbreviation of ') handles that problem.

Another problem that other formal languages (including programming languages and mathematics) have been able to side-step is the issue of value versus object. In mathematics, values cannot change; in programming languages numerical values cannot change, but items like arrays are expected to change in the sense that their components can be modified. Unfortunately there is a conspiracy of confusion in the programming domain that makes it difficult to realize that arrays are examples of what we will call mutable objects. Much of this stems from a fuzzy notion of what programming languages actually deal with--are they talking about values or are they talking about objects? Unfortunately, when languages try to talk about objects, they do so in a particularly weak, implementation-oriented way, rather than in a notationally elegant fashion. A fashion that has been called "first-class".

First-Class Objects

One further liberating idea has yet to be fully appreciated: the notion of first-class objects. One of the first statements of this notion appeared in thw writings of R. Popplestone, the originator of POP-2. In 1968 he wrote:

"... this brings us to the subject of items. Anything which can be the value of a variable is an item. All items have certain fundamental rights.

1. All items can be the actual parameters of functions.

2. All items can be returned as the results of functions.

3. All items can be the subject of assignment statements.

4. All items can be tested for equality. "

Several of these ideas were percolating around, and implied in the earlier LISP work, but this is one of the first explicit mentions of the notion of first-class objects. And yet sixteen years later, how well do languages live-up to that charter?

Even in the world of "abstract data", many modern languages make very concrete demands on the behavior of data objects. Those demands typically mean that each object be predefined and specified before a computation is begun. Think about Pascal, or BASIC, or FORTRAN for example. Can you pass in an array as a parameter to a function? Can you even talk about a function, or is everything a "procedure"? Can you assign an array as value to a variable? Can you create an array as the value of a function? No. The only kind of data objects that have this flexibility in these languages are numbers. Yet even here, constraints are placed on the sizes of integers and floating-point numbers. These constraints have nothing to do with the mathematics of the situation. They are imposed because of hardware limitations in the machine.

In a similar manner, limitations are imposed on data objects -- like arrays, strings, or functions -- because of difficulties in implementation. So for example, to use an operation like CONS in a traditional language we would have to pre-allocate space for CONS-like objects; or we would not be able to return a CONS-object as the value of a function; or we would not be able to pass a CONS-object as an actual parameter to a function; or we would have to explicitly "erase" a CONS-object when we were finished with it.

This kind of implementation-driven language design goes against the grain of using a language as a tool for expressing thought. What if we required that a natural language conversation begin with a definition of all words to be used in the conversation? Or if we required that all words be less than two syllables? The world would become very quiet. No, we use natural language in a very interactive "unstructured" fashion, defining words as needed, guiding the depth of the discussion by the level of understanding of the listener. We are definitely not advocating that natural language be used for computation. We are asking that the level of computational language be raised so that we can use the notation to describe our problems.

So just as natural languages are judged on their flexibility and level of expressiveness, we should expect no less from our computing languages. The notions of abstraction, information hiding, and particularly first-class objects, exemplify partial solutions to these concerns.

We're now going back to the less philosophical aspects of TLC-LISP to introduce more data types. However it will be useful to keep this discussion of first-class-ness in mind as we introduce new notions. In particular, those of you who have had experience with languages that support vectors (or arrays) should your vectors with TLC-LISP vectors.

Vectors

TLC-LISP includes two important classes of data objects commonly considered in discussions of general purpose languages. These are vectors and strings. The notion of vector is a special case of a list structure, being of fixed (rather than variable) length. We will also see in the next section that strings are in turn special cases of vectors, where each element is required to be a character. In either case, these objects represent finite sequence of elements. As such, vectors are venerable mathematical structures; and thought of in the historical scheme of modern computing, they were the next structure after numbers to be supported in hardware. The index registers that we all know and love support the rapid accessing of elements of a vector.

However, we prefer to think of vectors in the abstract mathematical sense and as such, we should be free to say "consider a finite sequence of n elements ..." at any time during our problem-solving task. This flexibility of discussion translates into the notion of first-class data objects when we move to programming languages. We mentioned that idea in the discussion of lists, but it bears repeating in the context of LISP's treatment of "typical data structures". In particular, the standard languages demand that vectors be "declared", meaning that before any program begins execution each vector must specify its intentions -- typically by specifying its name, its size, and the type of each of its elements. Such behavior is an anathema to those who wish to develop programs interactively. How often do we introduce new notions "on the fly" while we're problem-solving; why should programming languages be any less flexible?

Of course, our informal flexibility does not stop with the introduction of the new idea or terminology. We feel free to use that notion as the problem-solving continues. Translating this flexibility into the programming realm implies that we are free to pass our dynamically created vectors around throughout our computational conversation. It's the notion of first-class data again. And such is the behavior of vectors in TLC-LISP. Our vectors can be created dynamically, passed as values to functions, and returned as values for functions. They share with standard "general purpose" languages, the ability to access and set components. As notation and implementation, they represent a tradeoff when compared with lists. Vectors are less flexible, being of fixed size when created. However they are more natural to use in contexts that utilize a regular, predictable structuring between elements. For example, when describing a two-dimensional game situation, it is easier to visualize positioning and strategies in terms of a vector of vectors rather than a list of lists. The computation to move between rows and columns is easily described in terms of vector indices; the notion is highly unnatural when expressed as list operations.

In more detail, we express constant vectors in TLC-LISP as:

```
[<obj1> <obj2> ... <objn>]
```

where the <obji>'s can be any LISP object including a vector. So:

[1 2 3 4 5] is a vector of length 5.

[[1 2 3 4] [A S D E]] is a vector of length 2, each of whose elements is a vector of length 4.

Constructors

As with the previous data types, we also need ways of creating new vectors from a collection of elements. We supply two basic building blocks: VECTOR and NEWVECTOR.

First, with:

```
(VECTOR  <obj1> ... <objn>)
```

we can create a vector with the <obj>s as elements. For example,

```
(VECTOR 1 (ADD1 5) (CAR '(A . B)) (REST '(1 2 3)))

          =>  [1 6 A (2 3)]
```

We also support

```
(NEWVECTOR n init)
```

where the init expression is evaluated afresh for each element of the vector being created. Thus, if the current value of X is 0 then

```
(NEWVECTOR 6 (SETQ X (ADD1 X)))   =>  [1 2 3 4 5 6]
```

The result of these constructors can be used as components for any other LISP computation, as in:

```
(LIST (VECTOR 1 2 3) 1 2)  =>  ([1 2 3] 1 2)
```

Of course, such simple examples are pretty vacuous, but the point is that vectors, like lists, can be created dynamically within the course of a computation and then used for further computation. There is no sense of predetermining the size, or name, or type of components that may appear in a vector object. Furthermore, when we introduce user-defined functions we'll see that this spontaniety of vector creation carries over here too. Vectors can be passed as parameters, and can be created within a function to then be returned as a result--two characteristics of first-class objects.

Selectors

We select a component of a vector by:

```
(V I)          or equivalently      (VREF V I)
```

so

```
(VREF [A S D F] 2)  =>  S
```

The syntactic schizophrenia derives from two views of vectors. In the first view, a vector is a finite function (from integers to objects); as such, the vector is an applicable object and thus the act of applying an index to it "selects" the corresponding object. The second technique (using VREF) is the more common LISPish practice; it makes it obvious that the object we're dealing with is a vector (and not a function) and it helps a compiler generate code. This last reason is totally irrelevant to the notational issues, and is better handled as a compiler directive anyway; we'll let experience determine the preferred notation.

Updaters

Vectors and arrays are the first object-type to support the notion of mutation; ever since the days of FORTRAN, we have changed such objects by "storing" new elements at positions within the object.

We update a vector in TLC-LISP using the STORE operation:

```
(STORE <vector> <number> <object>)
```

replaces the indicated element of the vector with the designated element.

For example, executing:

```
(SETQ VEC-OBJ [A S D F E W])  =>  [A S D F E W]
```

then:

```
(STORE VEC-OBJ 3 1)  => [A S 1 F E W]
```

Two comments about the STORE updater should be made. First, since LISP evaluates each argument to STORE, we can see that STORE expects a vector-object as its first argument, not the name of such an object. That means we could write something (rather meaningless) like:

```
(STORE [A S D] 2 22)
```

and have it evaluate, getting a modified vector [A 22 D].

The second comment deals with the STORE-notation itself. STORE is the updating counterpart to VREF. What is the analogous notation for vectors-as-functions? Certainly

```
(<vector> <number> <object>)
```

suggests itself. Of course, that also looks like a function call with two arguments--which it isn't, thus now confusing the reader as well as the compiler. At least three options are open to us: (1) since such a "function application" will have either one or two arguments, we can make the convention that one argument means "selection" and two arguments mean "mutation"; this could even generalize to multi-dimensional vectors. (2) we could re-interpret the application of vectors to mean message-passing; we'll discuss this option later, and (3) we could stick with the STORE operation, pleading that neither of the other options offers a truly convincing case for adoption. We bow to this latter argument, with two suggestions; first, that TLC-LISP can be extended to support either of the other options (see AP, EAP, and the discussions of the Class System), and second, that we support this more regular selection and updating mechanism with environment objects (see the discussion a few sections hence). We do so there because the context in which environments tend to be used makes it clear that the update operation is something other than function application. Now, let's go back to a less problematic level.

Recognizers and other Functions

We can check the type of an object to insure that it is a vector using VECTORP.

```
(VECTORP [1 2 3 4])  =>  T

(VECTORP 3)  =>  NIL
```

We may determine the length of the vector by using LENGTH

```
(LENGTH [A X 1 2 4])  =>  5
```

Furthermore, we can use EQUAL and EQ to test the quality of two vectors. As always, EQ checks for identical objects, and EQUAL tests for indistinguishability. Thus:

```
(EQUAL [1 2 3 4] (VECTOR 1 (ADD1 1) (ADD 1 2) 4))  =>  T
```

Be aware that STORE is an updater, and thus the object represented by the vector is modified when the STORE is performed. So two (distinct) vectors that were EQUAL at one time may not be EQUAL at another instant. For example:

```
(SETQ X [1 2 3 4 5])  => [1 2 3 4 5]

(SETQ Y (COPY X))  =>  [1 2 3 4 5]

(EQUAL X Y)  =>  T          (EQ X Y)  =>  NIL


(STORE X 1 'A)  =>  [A 2 3 4 5]
```

and now

```
(EQUAL X Y)  =>  NIL
```

So to summarize: when compared with lists, vectors represent a more structured, and therefore less flexible technique for storing information. Once specified, a vector may not change its length, whereas lists may grow and shrink dynamically. However, both lists and vectors allow mutation (though we have not demonstrated this facility for lists yet). Both lists and vectors allow their elements to be arbitrary LISP objects. The next structuring unit restricts that flexibility.

Strings

Strings are a special case of vectors. A string is simply a vector of characters. Thus the string "aBc12" could be represented as [\a \B \c \1 \2] -- an accurate, but hardly pleasing countenance. Since user convenience is no small issue, we introduce strings as a separate type of object. A major point here is the attention to the input and output portion of a data structure definition; these considerations should be included with the constructors, selectors, and recognizers that we've already identified as being critical to abstract programming. In later sections we'll address the problems of extending the input and output facilities of TLC-LISP to accommodate new printed representations.

As with vectors, string objects have a supply of construction and selection operators. We have:

* STRING to create a new string, thus

(STRING "A s D" \c) => "A s Dc"

* SUBSTRING to select a substring,

(SUBSTRING "A s Dc" 1 3) => "A s" selects a string of length three, beginning at the first character position

* STRINGP to tell if an object is a string.

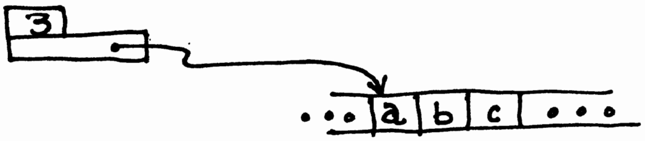
(STRINGP "AAA") => T

and finally, LENGTH, EQ, and EQUAL to determine the length of the string object or its equality to another object.

The Reference Manual contains many more examples of string-based functions, and so we'll let you experiment with the TLC-LISP calculator using that document. Here we'd like to dig deeper into some of the issues involving object versus values. We will couch the discussion in very concrete, implementation-level, details, not because that's the cause of the problem but rather it's an effect. But the concrete discussion may help clarify the abstract ideas.

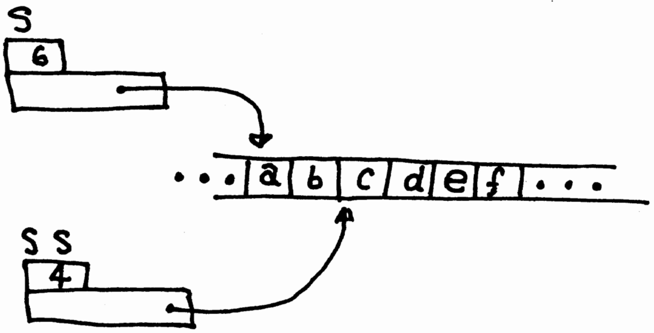
Vectors and strings in TLC-Logo share a common implementation technique called a descriptor-based object. The idea is that the descriptor carries information that will allow an external request to decode the internal representation of the object. One always refers to the object via the descriptor and thus the actual implementation of the object is hidden from the user. The system might, for example, move the object representation around in memory; but since the object references always pass through the descriptor, the user program will be unaware of any system reorganization. Of course, such discussion and insight involves the deep implementation level of TLC-LISP, and most LISP code will never get that far into the subsconscious.

Below is the TLC representation of the string "abc". It consists of two parts: (1) the descriptor, giving the type, size and location of the characters of the string; and (2) the actual characters.



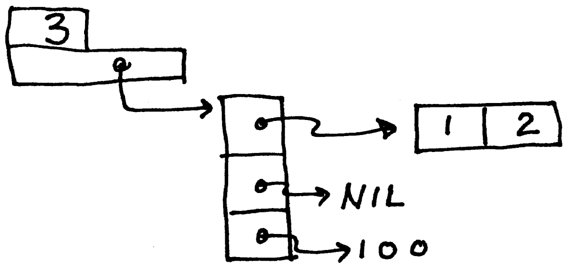
A String "abc"

In our implementation, we represent substring operations by building an appropriate new descriptor and share the actual characters. Thus:



(SETQ S "abcdef") (SETQ SS (SUBSTRING S 2 4))

Vectors are handled in a similar manner: a descriptor and a collection of object references (rather than characters). Thus:

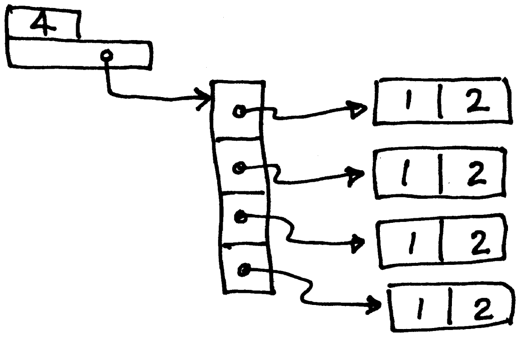


[(1 . 2) NIL 100]

Now let's look at the vector constructor NEWVECTOR. Consider:

(NEWVECTOR 4 (CONS 1 2))

Since NEWVECTOR evaluates the second expression each time it needs an element, and since CONS generates a new element each time it is called, we generate the following structure.

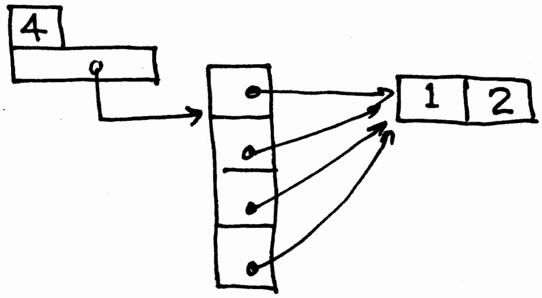


(NEWVECTOR 4 (CONS 1 2))

In contrast, consider the following:

(NEWVECTOR 4 '(1 . 2))

In this case, there's only one instance of the dotted pair, and so we build the following:

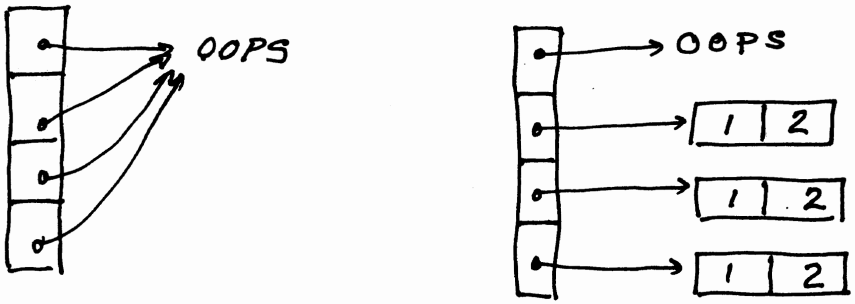


(NEWVECTOR 4 '(1 . 2))

Of course

(EQUAL (NEWVECTOR 4 '(1 . 2)) (NEWVECTOR 4 (CONS 1 2))) => T,

but the effect of updating an element is quite different; in one case, all elements are affected, and in the other case only the indicated element changes.



Two versions of (STORE <vector> 1 'oops)

Since applications that involve vectors are expected to perform many STORE operations (mutations), a "non-shared" element structure is preferred.

There are several issues involved here, but of particular interest for implementers is the "copy versus share" decision. Why _should_ CONS copy? Why shouldn't it automatically share a single instance of any dotted pair? Indeed, some versions of LISP have been built this way (called "hashed LISP" because the techniques employed to assure uniqueness of representation tend to involve "hash coding" all references to pairs--more about this in the next section). However, as the previous example indicates, such sharing can have very unforseen consequences.

This real issue is not one of implementation, but a question of what it is that is being represented in the language. If the language is value-oriented then, since values don't change, hashing implementations make good sense (for example, in a totally hashed system, EQ and EQUAL can become one). In contrast, an object-oriented language should expect to encounter changeable objects and so gratuitous copying and/or sharing cannot be tolerated. We believe that the emphasis should be on the notion of object, rather than value, and have tried to make the language reflect that concern.

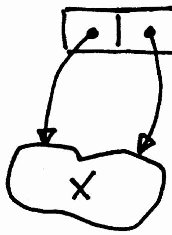
Even after all that, in most of LISP, the copy-versus-share arguments are irrelevant. Only when we venture into the "updating swamp" do we find this level of detail important.

Updaters and Mutation

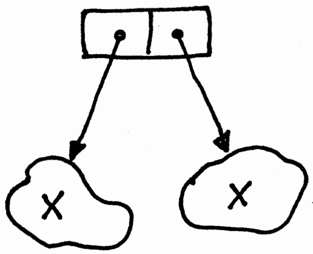
The issue of value versus object comes into sharper focus when we allow our data items to change. We introduced that problem in the context of vectors as objects, and continued the discussion in EQ versus EQUAL. We now want to extend this treatment to S-expressions and list-based objects.

To begin, let's highlight some of the differences between vector and S-expressions. A reasonable visual model of a vector is a (linear, fixed-length) sequence of elements. We may change the elements in the sequence, but not the length of the sequence. In contrast, one graphical representation of the S-expressions (and indirectly, of lists) is a binary tree whose tip-nodes are atomic quantities. We say "one representation", because there are others. In particular, the issue of value-versus-object and copy-versus-share have strong impact on the graphical model we'll consider.

For example, consider the expression (CONS X X). If we think of X as designating an object, then a binary tree representation does not faithfully describe our intention. For a binary tree, by definition, has no intersecting branches, and an object-interpretation of (CONS X X) says we have two references to the same object. What we really want here is called a "graph", rather than a tree, because graph structures can have intersecting branches. The following picture illustrates the two options:



Graph Representation



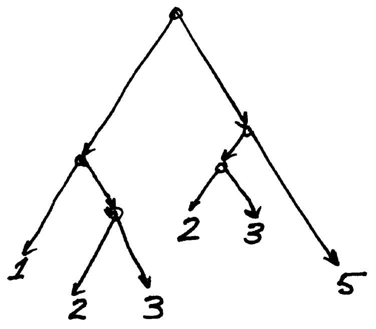
Tree Representation

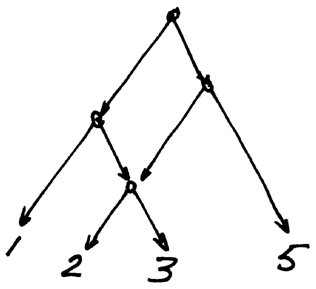
(CONS X X)

The issue of copy-versus-share can go deeper than simple CONSing. It can involve issues of unique representation throughout the system. For example, examine the following expression:

```
(CONS '(1 . (2 . 3))  '((2 . 3) . 5))
```

Notice that there are "common sub-expressions" in the arguments to CONS; (2 . 3) appears in both arguments. As a result, there are two ways to interpret the resulting CONS:





Copy Versus Share

The first implementation is what's traditionally done; simply grabbing a node and stuffing the components into the CAR- and CDR-portions. The second is more difficult, sharing all possible substructure when adding dotted pairs via CONS-type operations. Actually, the sharing described in this example would have occurred during the parsing operations that built up the internal form of the CONS-expression. Whenever any CONS operation is performed, the system would be required to examine its stock of existing nodes to determine if the desired dotted pair already existed. This latter behavior is a feature of "hashed-LISPs". Which interpretation do we want; naive or hashed? To some extent, we want neither.

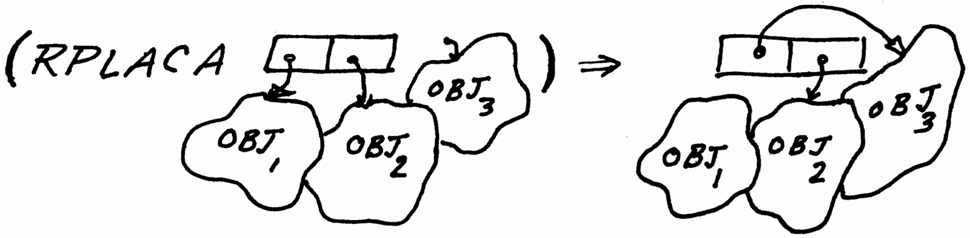
We want to think of lists and dotted pairs as the hexadecimal level of object management; as such, the gratuitous sharing can be quite problematic because that represents the assumption that surface similarities imply deep identity. So we'd rather not think of objects as lists or dotted pairs, but as abstract objects whose implementation just happens to fit into a LISP-style structure. If structure is to be shared, then that is a result of a deeper analysis that we have imposed on the representation.

Of course, all this concern for copy-versus-share, EQ-versus-EQUAL, and value-versus-object may seem like a tempest-in-a-teapot. Indeed, much of the discussion is academic until we consider data items that can change--what we've called objects (as compared to values). In the presence of change, all these issues come to bear. With this in mind, we'll introduce the two updaters--not as modifiers of dotted pairs (or lists) but as modifiers of representations of objects.

What does it mean to update an object? It means that the constituents of the object (actually its representatation) may be modified. We've seen this issue with vector objects; the components of a vector are its elements, not its length. Thus we could modify the vector elements, but not the topology of the vector (i.e. its length). To date S-expressions have been treated like they were values: we have techinques for creating new ones but no way to change existing ones. We now want to introduce operations to modify the components of dotted pairs.

Since pairs are created by combining two existing objects, it's appropriate to think of the updaters as changing these object-references. So, we have two operations--one to modify the CAR-component of a dotted pair, and one to modify the CDR-part. With their introduction, the object-versus-value distinction cannot be denied.

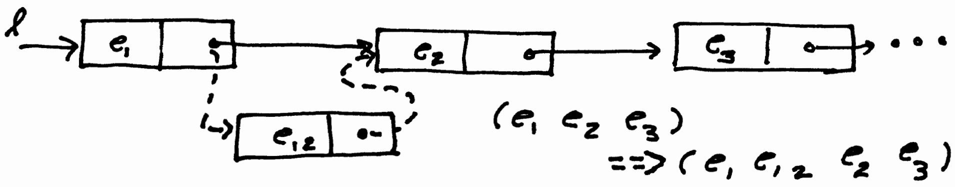
S-expressions are not values in LISP; they are objects. As such, they have two updater functions. RPLACA (standing for RePLAce the CAr-part of a pair), and RPLACD (standing for RePLAce the CDr-part of a pair). Below is a diagrammatic representation of RPLACA (RPLACD is totally analogous).



The Effect of RPLACA

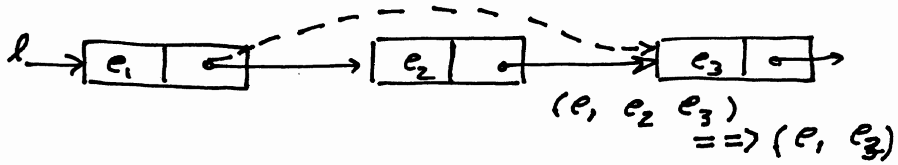
With these two functions in place, we can now open up the distinctions between vectors and lists (as represented by dotted pairs).

We can think of the elements in a list as being similar to children's "pop-beads"; using CONS and RPLACD, we can add new elements in between any two existing elements:



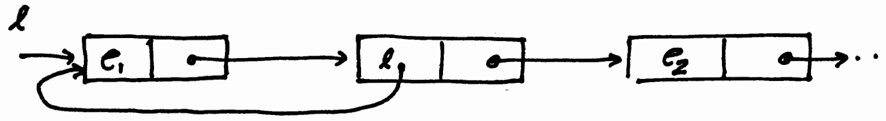
Insertion of an Element in a List

Using just RPLACD, we can remove an element from a list.



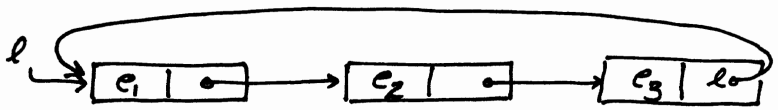
Deletion of an Element from a List

As with vectors, it is possible to make an element of a list be that list, itself; this requires RPLACA:



A List with Itself as an Element

However we can also work further down in the topology of the list or dotted pair; we can modify the successor relationship, even to the point of making circular lists:



A Circular List

The point of such exercises is not obscurantism, but that objects in the real world and their models in our minds--both things that Artificial Intelligence wishes to study--don't necessarily have the simple modular structure that vectors or lists supply. Rather, the interrelationships are more spaghetti-like, with inter-object references not just established at the time the object is created, but relationships that expand and contract dynamically as situations change and as our understanding of them change. Any language that expects to deal with, or model, reality must supply "mind altering" operations of the power of RPLACA and RPLACD.

However, what *we* must bear in mind is the appropriate context in which to apply these operations. Not as ways of hacking at arbitrary S-expressions, but as surgical tools for modifying components of objects represented by S-expressions.

These ideas about data give an adequate picture of TLC-LISP's attitude toward data so that we can safely begin to reveal the structure of LISP programs. The essential notion to keep in mind is "resist compromise". Keep the notation expressive and implementation independent; we'll see that this slogan leads to the notion of "first-class" functions to complement our "first-class" data.

The two components we need to introduce are the ideas of control structure--how to describe decisions, or more accurately, prescribe actions; and second, how to enlarge the vocabulary of TLC-LISP by adding new functions. The next two sections cover these points.

Explicit Control

What distinguishes a calculator from a computer? Though any distinctions are rapidly fading, one historical difference has been that calculators were restricted to sequences of straight-line instructions, while computers contained instructions that allowed the conditional execution of instructions, based on the outcome of some test; one result implied the execution of one set of instructions, and a different outcome implied that a different calculation would occur.

These explicit indications of conditional computation have appeared in all traditional programming languages--including LISP. In this section we'll indicate a few of these constructs, and in the next section (on functions) apply them to the construction of user-defined functions.

The simplest form of conditional computation in TLC-LISP is the IF expression:

```
(IF <predicate> <true-expression> <false-expression>)
```

as in

```
(IF (ATOM X) (CONS 1 X) (CAR X))
```

Actually there is a simpler form:

```
(IF <predicate> <true-expression>)
```

that will compute NIL if the <predicate> is not true. And there is a more complex form:

```
(IF <predicate> <true-expression> <false-exp-1> ... <false-exp-n>)
```

meaning that the false part may contain a whole sequence of expressions. Note that there may only be one <true-expression>. To carry out a sequence of computations in the true case, use PROGN (a grouping and sequencing operation--See the Manual for more details)

```
(IF <predicate> (PROGN <exp-1> ... <exp-n>)  <false-part ...).
```

As we illustrated in the introduction, use of the IF-expression has a tendency to infect LISP with a kind of "Mouse's Tale" appearance. To subdue this effect, LISP supports the COND expression:

```
(COND (<predicate> <exp-11> ... <exp-1n>)
      (<predicate> <exp-21> ... <exp-2m>)
                  .  .  .               )
```

where the predicates are evaluated sequentially, stopping with the first one that evaluated to a non-NIL value. Finding such a predicate, we then evaluate the corresponding expressions (from left-to-right) and the value of the COND is the last such value. The COND expression will linearize and clarify many instances of conditional computation.

Further clarification can be catered to when the conditional behavior is based on the particular value of a single expression. In other languages, this is expressed as a CASE statement[sic]; in LISP, such an expression [sic] first appeared in 1958 as the SELECT, and in deference to history we retain this notation. In particular, TLC-LISP supports:

```
(SELECTQ <exp> (<pattern> <exp-11> ... <exp-1n>)
               (<pattern> <exp-21> ... <exp-2m>) ... )
```

where the value of <exp> is compared against the <pattern>s in a first-to-last manner. The <pattern>s are not evaluated, but are used directly. For specific details, see the Reference Manual. The point here is that we supply several different control operations, each of which is patterned after a specific (and common) pattern in symbolic data. For example, SELECTQ acts like a "dispatch table", and will seem natural in situations where there are several alternative sub-types of a class of objects; in contrast, DO is more natural in situations where the data object is homogeneous, but has several components. The DO is an iterator or generalized looping construct; see the Reference Manual for details.

In a more program-control (as compared to data-control) situation, we offer CATCH, THROW, and UNWIND-PROTECT to "bullet-proof" computations. See the Reference Manual for examples of these non-structured control operations.

As a final, general note, about control structures and predicates, TLC-LISP follows traditonal LISP usage by allowing any non-NIL value to be treated as "true". This frequently has application in situations where we wish to test for the existence of an object and if such an object exists, do something with it; if there is no such object, we do something else. "non-NIL=true" allows us compute a reference to the object, use it as "true" and then use the object itself.

One constant theme has been present throughout this section: the control operation is explict in the sense that there is a "reserved word" that signals the occurrence of a control operation. Notice, first, that control operations are not the same kind of creature as function invocation. The "arguments" to these control operations are expressions or lists of expressions, and their execution (or lack of execution) is dependent on evaluation of other components in the "argument list". In contrast, functions like CAR, and LIST always utilize all of their arguments, and all of their arguments denote data objects, whereas the arguments (such as they are) to DO, IF, and friends, denote LISP expressions. The difference is more than cosmetic.

However, LISP functions are not without their own idiosyncrasies in the control area. Though pure LISP functions denote mathematical functions, the programming language LISP encodes some substantial decisions about how values for functions are discovered. We will go into these areas in the section called Evaluation, but we must mention here that those decisions also impose some notions of control on LISP expressions. The effect of these decisions is more subtle (and perhaps more pernicious) since they do not appear in the notation, but are embedded in the evaluation mechanism. They are called "implicit control" structures; call-by-value and left-to-right evaluation of expressions are two examples of such control. Before we delve into this area we need to explore the computational interpretation of function more deeply.

Functions

Functional Objects in LISP serve the purposes that procedures serve in other languages: they express what we tend to think of as the active part of the computation. In most high-level languages, one expects to describe these active objects by:

1. a name -- a way of referring to the object.

2. formal parameters -- a way of passing information to the object.

3. a body of executable forms -- the actual computational component of the object.

For example, in an Algol/Pascal-like language, the active unit is the "procedure", a name is required, formal parameters must declare the expected type of the arguments passed to the procedure, and the executable forms are statements. Finally, if we expect the subprogram to return a value then we must suitably decorate the interior of the program with an indication that a value is to be returned, and we must declare in the prolog to the definition that the value is of a specific type. Even then, severe restrictions are placed on the kind of result that may be returned, and limits are placed on the way we can use the "function subprograms".

In contrast, in LISP the active unit is the "function", a functional unit (being first-class) need not be named, the formal parameters need not be declared, and the executable forms are expressions (rather than statements). Let's analyze these points in detail:

* Procedure versus Function
* Names versus first-class values
* Typed names versus typed objects
* Statements versus expressions

together they express the different "state of mind" that is present in LISP and other modern languages.

1. Procedure versus function. The essential notion of a procedure is that of process or side-effect. One performs a computation to effect change. It is a highly state-oriented notion; the result of a computation is placed in some location, and those who wish to apply the result must know how to find that location. In opposition, we find the notion of function --one performs a computation to supply values for a function. Function is a mathematical notion, independent of time, order, or state. Thus a LISP function is expected to compute a value, that may be composed with other functional units to build up a complex

notational description of a phenomenon. On the other hand, since procedures do not return values, the notion of composition makes no sense. Procedural computation consists of a sequence of procedure invocations, whose results are stored in the machine state; any sense of communication of results to future elements in the sequence must be accomplished by opaque use of non-local variables.

2. Names versus first-class values. Names are convenient most of the time. They are the stuff out of which we are able to build algebra, for example. Instead of having to deal with specific values we can say "We know that x has value 2*z", for example. In this context, "x" and "z" are variables, and it's quite useful to use these variables as abbreviations. On the other hand, it is equally convenient to use constants like "2" in this algebraic conversation. It would be awkward to require that we write "We know that x has value two times the value of z", even if we could discover that "two" names the numeral 2. The point is that constants -- pure values, in other words -- are of equal importance for notational clarity. We allow constants in most programming languages --2, (A B), [1 2], "Abc", for numerals, lists, vectors, and strings, for example -- why not allow constants for functions? Most languages answer "why would anyone want a thing like that?" and promptly dismiss the idea. LISP is more reasonable. LISP functions are first-class, and thus are available as manifest constants (like 2, rather than "two"). We will demonstrate these ideas after we dispense with the next two points.

3. Typed names versus typed objects. Most languages require that the names for the formal parameters to a procedure or function be categorized with respect to the type of object that will be accepted as an actual parameter. This requirement has some merit. The notation makes clear what is expected, and thus simplifies the reader's problem in understanding the programs. It also makes the job of the compiler writer easier; knowing what types of arguments are expected, the implementor can generate optimized code. In this scheme of things then, the names of objects contain the information that determines the type of an object. In LISP, on the other hand, it is the object itself, that carries this type information. Given LISP's insistence on first-class objects, it makes sense that objects contain their own type information. But the issue goes deeper.

In the ancient days (thirty years ago) typed names may have made sense. They helped the compiler writer generate better code --execution time was a scarce resource. Typed names also helped the programmer control error propagation --programmer time was relatively inexpensive with respect to machine time. Also, in those days, programs were complex and data was simple--integers, arrays, and strings.

Times have changed. The emphasis is on interactive program development now, in an attempt to minimize programmer time. Programmers are now expensive, relative to the cost of hardware. More importantly, the programming task has become much more complex. No longer are we just doing numerical calculations; we're computing with complex objects --with the emphasis being on the interractions between these objects. Message-passing, data-driven programming, object-oriented notions,... all reflect this change of perspective from code to data. Objects are created dynamically, new types of objects are constructed, relationships between objects are modified. The emphasis is on change, on run-time variability, and neither of these notions fits the old model of typed names.

4. statements versus expressions. Finally, there is the issue of what constitutes the body of the procedure/function. In the procedural world, these elements are statements -- units that change the state of the computation (like an assigment statement for example). In the functional world, the elements are expressions -- gentle souls that compute a value and graciously offer that value to whomever wishes to accept it.

It should be clear where our allegiances lie -- procedures, names, and declarations are voices from the past. Functionality, first-class values, and self-governing objects are some of the voices of the future computing generations.

# # #

Given this spirited introduction to functional objects, exactly what constitutes a description of a LISP function? Well, we'll at least need the following:

i. An indication that the object being described is a function. We do this by wrapping the functional components in a list prefixed with LAMBDA. Thus:

```
(LAMBDA   ....   )
```

ii. The list of formal parameters is the next component of a definition. Thus:

```
(LAMBDA (X Y) ... )
```

designates a function with two formal parameters, X and Y.

iii. Finally, we must specify the body of the function. For example:

```
(LAMBDA (X Y)
  (SUB (MUL X X)
       (MUL Y Y)) )
```

This collection of notation describes a function of two arguments that squares both arguments, and then subtracts the second square from the first. In this case, there is only one expression in the body of the function. In the general case, there many be a sequence of expressions. Thus:

```
   (LAMBDA <parameters> <exp-1> <exp-2> ... <exp-n>)
```

Notice that the functionality is completely contained in the representation; that is, the LAMBDA-construct denotes a function-constant. We might be tempted to name the function in the last example DIFFERENCE-OF-SQUARES, but we may apply this functional LAMBDA-expression anonymously (without giving it a name). Thus:

```
((LAMBDA (X Y)
    (SUB (MUL X X) (MUL Y Y) )
  3 2)
```

will evaluate to 5.

We perform this evaluation like any other: first evaluating the actual parameters--in this case, both are constants (3 and 2); then we associate these values with the formal parameters (3 with X, and 2 with Y), and evaluate the body of the function.

We want to concentrate on the ability to define new LISP functions. This is at the heart of the LISP system, since it allows the user to interactively enlarge the vocabulary of words that are recognized by the LISP machine. For a mahine is what we have: the evaluation process that we've been talking about represents part of the "basic cycle" of the processor--the CALL instruction if you wish. You may think of the primitive operations (CAR, CDR, etc.) as the instructions of a machine and the forms of the conditional expression are the JUMP instructions. We will talk about how these operations are executed in the section titled _Evaluation_. Here, we will first build up some familarity with LISP's notion of function, and then explore a few of the potentially problematic areas of functional computation.

First, though we can refer to un-named (or anonymous) functional quantities, it is most usual to supply names and refer to the functions via names. For example we could define a function that computes the square of its argument by:

```
(DE SQUARE (X) (MUL X X) )
```

or we could name our last example function by:

```
(DE DIFFERENCE-OF-SQUARES (X Y)
         (SUB (MUL X X) (MUL Y Y)))
```

where it's not too difficult to see that

```
(DE name (variables) body)
```

is an abbreviation for: "make name synonymous with the function (LAMBDA (variables) body)". Thus DE "assigns" the function to the name as value.

This ability to define functions and pass parameters is the heart and soul of any LISP-like language. With these facilities, we need not worry about iteration and "goto" and all the other trappings of a so-called traditional language. However, just like any "natural" language, a powerful language must be able to introduce abbreviations to improve expressibility. So too with TLC-LISP. In particular, we have included some of the more succinct notations for controlling parameter-passing derived from MDL, Conniver, and the MIT LISP Machine.

Most programming languages require that there be a one-to-one correspondence between the actual parameters and the formal parameters before binding those parameters and evaluating the function body. Several LISP dialects have relaxed that restriction. However, that freedom is usually purchased at the cost of some very helpful parameter-checking information. In these other schemes, if too many arguments are supplied, their values are discarded. If too few are supplied, the missing parameters are gratuitously bound to NIL. Such behavior is hard to predict or debug, and therefore hard to condone.

However, some relaxation of parameter passing is desirable; in particular, the ability to supply an arbitrary (therefore variable) number of arguments. For example we would rather write

(ADD X Y (ADD1 Z)) than (ADD X (ADD Y (ADD1 Z))).

Many instances of this variadic call can be accomplished by macro expansion (a technique we'll introduce in a succeeding section), yet the problem begs for a general solution.

Finally, a common application of the "PROG-feature" (for those of you who have LISP experience) is the declaration and immediate initialization of "PROG-variables". These variables are to be used locally and discarded upon function exit.

We can accomplish all of these desirable features with variations on a small set of conventions.

Parameter Specifications

The traditional list of formal parameters in a LAMBDA definition will be called required parameters -- a one-to-one correspondence between actual parameters and required parameters must be fulfilled or an error is signalled. We extend the parameter syntax using the reserved words:

```
&OPT (or equivalently, &OPTIONAL)
&REST
&AUX
```

with the most general formal parameter description as follows:

```
(<required> &OPT <optionals> &REST <rest> &AUX <auxs>)
```

In the usual case, some of these groups may be absent.

Now let's see what these conventions mean.

<Required> is a sequence of zero or more symbol names.

<Optionals> and <auxs> are non-empty sequences of either atoms or lists whose first elements are symbols. In this second case (a list), the remainder of the list is to be interpreted as a value to be assigned to the variable represented by the first element of the list. For example:

```
(<required> &OPT (X (ADD Y N)))
```

would mean "assign the sum of Y and N to X if the optional parameter is not supplied."

<Rest> must be a single symbol.

To put the process in most general terms: first, we evaluate all of the actual parameters; then we apply the following algorithm for matching this extended form of parameter passing:

1. First, the required parameters must be matched. If these requirements cannot be satisfied, an error is signalled.

```
(X Y Z) will match (1 2 3), but not (1 2) or (1 2 3 4)
```

2. If (evaluated) actual parameters still remain and <optionals> were provided for, then we continue binding actuals to the optionals. If we exhaust the actual parameters in this process then any remaining optionals are bound to their default value or to the distinguished object UNBOUND if no default value was supplied.

For example,

```
(X Y Z &OPT U (V 2)) will match

(1 2 3), (X gets 1, Y gets 2, Z gets 3, U gets UNBOUND, V gets 2)
(1 2 3 4) and  (as above, but U gets 4)
(1 2 3 4 5),   (as above, but V gets 5)
```

but not

```
(1 2 3 4 5 6) or  (1 2)
```

3. If after step 2, actual parameters still remain and a &REST parameter was declared, then we create a list of the remaining parameters and bound it to the <rest> variable. If no REST parameter was supplied then a TOO-MANY-ARGS error will be signalled.

(X Y &REST Z) will match (1 2 3 4), binding Z to (3 4).

(X Y &OPT U &REST Z) will match (1 2 3), binding Z to UNBOUND ; it will match (1 2 3 4 5), binding Z to (4 5). In both cases U is bound to 3.

4. Finally, the auxiliary parameters declared by &AUX are processed. If initial values were specified, they are used. Otherwise the parameter is initialized to UNBOUND.

All of these various binding styles are governed by the LAMBDA-binding mechanism; that is, the old bindings of these variables are saved on entry to the function. After the body of the definition is evaluated the old bindings of these LAMBDA variables are restored.

The combinations of these various options gives the programmer a clear, concise, predictable, and powerful mechanism to control the passing of parameters. Now let's do a few examples using these ideas.

Earlier we advertised a function named APPEND that took two arguments, both of which were lists, and created a new list with the elements of the first list tacked onto the front of the second list. For example:

```
(APPEND '(1 2 3) '(A S W))  =>  (1 2 3 A S W)

(APPEND () '(A S D))  => (A S D)
```

Here's APPEND:

```
(DE APPEND (X Y)
  (IF (NULL X) Y
      (CONCAT (FIRST X)
              (APPEND (REST X) Y))))
```

This function has only required parameters.

We can use optional parameters to let a single function play double duty; first, to initialize parameters, then on successive recursive calls, to update those parameters. For example, a common definition of a function to reverse a list is:

```
(DE REVERSE (X) (REV1 X ()))

(DE REV1 (X Y)
   (IF (NULL X) Y
       (REV1 (REST X)
             (CONCAT (FIRST X) Y))))
```

With optional arguments we can combine these two functions as:

```
(DE REVERSE (X &OPT (Y ())
   (IF (NULL X) Y
       (REVERSE (REST X) (CONCAT (FIRST X) Y))))
```

We can use mulitple optionals as well. For example, we could define a function to compute the sum of the elements in a vector by:

```
(DE VECTOR-SUM (V &OPT (I (LENGTH V))
                       (SUM 0))
       (IF (ZEROP I)
           SUM
           (VECTOR-SUM V (SUB1 I) (ADD (V I) SUM))))
```

then for example (VECTOR-SUM [1 2 3]) gives 6.

Auxiliary parameters are used as local variables. For example, several sections ago we mentioned the use of the NIL, non-NIL representatations for false and true. The ideas was that if a predicate returned a non-NIL element, then that element could be used as an indication of success (true) and then could be used (as an object) in further computation. An auxiliary parameter is a good receptacle for such an element. Thus:

```
(DE TESTER (X &AUX (Z (LOCATE-FACT-ABOUT X)))
  (IF Z
      (DO-SOMETHING-WITH Z)
      (DO-SOMETHING-ELSE-WITH X)))
```

identical to the "add 1" example, but with a different function applied to each element, we should be able to abstract out that specific operation and describe it as a formal parameter.

And so we arrive at:

```
(DE MAP (F L)
  (IF (NULL L)
      ()
      (CONCAT (F (FIRST L))
              (MAP F (REST L)) )))
```

with MAP-ADD1 defined as:

```
(DE MAP-ADD1 (L)  (MAP ADD1 L) )
```

and MAP-SUB1 as:

```
(DE MAP-SUB1 (L) (MAP SUB1 L) )
```

What could be simpler? The notation certainly is elegant. It is, however, an extension of what most programming languages allow; namely the first argument to MAP is a function, not a data object as we usually think of data objects.

We could also envision functions that create new functions as their values. Assume, for example, that we wanted to define ADD1 and we had the binary addition function ADD at our disposal. Then:

```
(DE ADD1 (N) (ADD N 1) )
```

But now, what if we decided to generalize ADD1, ADD2, ... ADDn as we decided to generalize the mapping functions? We might soon wish for a function-generating function. Something like:

```
(DE ADDN (L)
      (LAMBDA (M) (ADD M L)) )
```

that would create a function as value -- (LAMBDA (M) (ADD M L)) with a specific value associated with L. So for example we might try to define ADD1 using (ADDN 1) as:

```
(DE ADD1! (X) ((ADDN 1) X) )
```

where the value of (ADDN 1) appears in the function position within the body of ADD1!. This may seem a bit surprising, but the value of (ADD1! 1) _is_ (or tries to be) a function. We'll try to call this function in the usual way as

```
(ADD1! 3),
```

and we'll find that ADD1! has not remembered that L should have the value 1.

This problem arises on most LISP systems that try to use functions as first-class objects. The issue is the correct tracking of variables like L, above--called non-local variables. Historically, LISP implementations have been inadequate. What is really called for as the functional value of (ADDN 1) is the function

```
(LAMBDA (M) (ADD M 2))
```

where we have explicitly replaced every (in this case, one) occurrence of L in (LAMBDA (M) (ADD M L)) with 2.

Unfortunately, explicit substitution is time-consuming and difficult to specify accurately when that substitution is being made into expressions, not simple lists. For example, it is easy to write a function to do substitution into arbitrary S-expressions (and therefore into lists):

```
(DE SUBST (X Y Z)
  (IF (ATOM Z) (IF (EQ Y Z) X Z)
      (CONS (SUBST X Y (CAR Z)) (SUBST X Y (CDR Z)))))
```

This function creates a list with the same structure as Z but wherever Y appears in Z, X appears in the new S-expr.

However, the story is quite different if the substitution is "semantic". Consider applying SUBST to the following list:

```
(LAMBDA (M &AUX (N 3)) ... (SETQ N (ADD1 N)) ...)
```

replacing N with 2. The result is nonsense.

This doesn't mean that we cannot specify a correct LISP function for that kind of substitution; indeed we can. However, the scheme is complex but more importantly as we shall see later, explicity substituting values for variables will destroy some very interesting computational possibilities.

Computer science has cooked up an alternative to real substitution: we simulate substitution. Instead of substituting values for variables, we keep notes that associate values with variables. Then, when we come across a variable during the evaluation of an expression, we take a look at our notes; we supply the value for the variable, and continue.

This mechanism that encapsulates the note-taking is called a "symbol table" or an "environment". A few sections hence we will give a detailed description of how to use environments to solve the functional object problem. Now we want to describe what environments are.

Environments

An environment (or symbol table) is a collection of symbols and associated values; they are first-class objects in TLC-LISP.

An environment is built from an alternating sequence of symbols and expressions:

```
(ENV 'A 1 'B (ADD1 5))  =>  (ENV 'A 1 'B 6)
```

where the first occurrence of ENV represents a call to the environment constructor, and the second occurrence indicates the printed representation of an object of type environment.

Besides acting as a (passive) symbol table for a function (the application that prompted their creation), an environment object can also act as a (active) finite unary function. If an environment appears in the function position of an application, and is followed by one of the variable names, then the current value of that variable is returned.

```
((ENV 'A 2 'B 6) 'A)  =>  2
```

We may also update the environment by supplying two arguments: a variable name (in that environment) and an object.

```
((ENV 'A 2 'B 6) 'A '(A B))  =>  (ENV 'A '(A B) 'B 6)
```

Another interpretation of this application of environments is to look at it as "message-passing" When we send the single message of a symbol name, the environment "knows" that we want to extract the value; when we send the pair--symbol and object--the environment knows that we want to update the state. Compare this treatment of environments with our discussion of vectors, VREF, and STORE.

Environments are our first indication of a duality between functions and data; sometimes they're data, sometimes they're functions. We'll see more of this mixing of behaviors in the class system when we represent dotted pairs as functions, and their constructors, selectors, and recognizers as data. The key ingredient in this investigation is a firm understanding of functional objects that have local state. These are called "closures" in TLC-LISP.

Closures

Given the symbol table or environment structures, we can return to the problem of characterizing functional objects.

Recall that our attempts to specify a LISP function that returned a function as value drove us to a position of either

(1) making explicit substitutions of objects for variables in expressions, or

(2) simulating the substitution by using a symbol table.

We opted for the second, but the scheme requires that we view a functional object as a complex of information:

(a) the program text

(b) an environment -- or symbol table -- of names and values.

In our particular example we had to associate the number 1 with the variable L in any the application of the function

```
(LAMBDA (M) (ADD M L))
```

The particular construct in TLC-LISP that supports this notion of functional object is called a "closure object". A Closure object makes explicit reference to the two above mentioned components: the function text, and the environment. Thus, a closure is:

```
(CLOSURE function-description environment)
```

where the environment is a representation of a symbol table of names and values.

So we could define ADDN as the following closure:

```
(DE ADDN (L)
   (CLOSURE (LAMBDA (M) (ADD M L))
            (ENV 'L L) ))
```

Thus the value of (ADDN 1) is

```
(CLOSURE (LAMBDA (M) (ADD M L)) (ENV 'L 1))
```

Whenever a Closure object is applied, as in (ADD1! 3), or equivalently

```
((CLOSURE (LAMBDA (M) (ADD M L)) (ENV 'L 1)) 3)
```

the local environment takes precedence over any other name-value associations that might be in effect before the function application.

So:

```
((CLOSURE (LAMBDA (X) (ADD X N)) (ENV 'N 2)) 3)

     => (ADD 3 2) => 5
```

We will discuss the evaluation of expressions involving closure objects in more detail later.

The symbol table, or local environment, captures a local world for the function to manipulate. So any changes to values of variables that appear in the environment will be duly recorded when we leave the context of the closure. So,

```
((CLOSURE (LAMBDA (X) (SETQ Z (ADD Z X))) (ENV 'Z 3)) 2)
```

will have the effect of setting the value of Z to 5 within the environment of that closure object; the value of Z outside the closure is not changed. This notion of a local environment first appeared in non-LISP languages as "own variables" in Algol-60.

The requirements for first-class functions make demands on the implementation of a language. Thus many languages completely prohibit dynamic functional objects; some allow a subset of functional arguments; others allow the free discussion of functions, but restrict the class of functionals that may be discussed dynamically.

Though these closure objects may seem complex, arcane, and perhaps even useless, we assure you that they are of substantial practical value, and not just a language exercise to force everything in a language to be first-class. Closure-objects are the "stuff" out of which one can understand the ideas within class systems and their implementation.

We're seeing a further clouding of the distiction between functions and data; we had functions appearing as arguments to other functions, and being returned as values; we had the dual use of environments as symbol tables and as finite functions; in the next section we'll continue that trend, introducing a powerful data structuring idea that can be used to drive data into execution.

Property Lists

A property list, also known as a "p-list", is a data structure consisting of a collection of pairs: one element of the pair is called a property name or "attribute". The other element is called a property value. Perhaps something like:

```
((COLOR RED) (DIAMETER 6.5) (WEIGHT 123))
```

Such a description will be associated with a LISP object--in this case, perhaps a particular heavy, red ball. Property-lists are not first-class objects in TLC-LISP (and most other LISPs as well). Rather, a p-list is associated with the name of an object; so we might find the properties associated with a symbol like BALL-1.

Typically one accesses the property list using a property name, and extracts a property value or changes that value. Thus, for example:

```
(GETPROP 'BALL-1 'COLOR)  =>  RED

(PUTPROP 'BALL-1 'WEIGHT (ADD (GETPROP 'BALL-1 'WEIGHT) 2))
```

In this regard, a property list is similar to a more traditional record structure. However LISP p-lists differ in several important ways. First, they are dynamic. They may grow and shrink while the program is running. This makes them an extremely flexible storage mechanism, since their storage need not be declared ahead of time. So their natural storage representation is more list-like than vector-like.

A property-list is also much like an environment. However, property-lists and environments differ in several important ways. Environments are first-class objects, p-lists are not. Property-lists are expected to grow and shrink, environments are not. We expect to see an object of type ENV in the context of a function object; we expect to see a property-list in the context of the attributes of a (named) data object.

Yet property-lists and environments are a pivotal point between the notions of data and the notions of function. For example, the flexibility of property-lists combines beautifully with LISP's program-data duality, giving rise to a technique called data driven programming. The basic idea is to place functions in the value positions of a p-list, and use the attribute as a "symbolic index" to select the appropriate function. In a data-driven situation, that symbolic index will be supplied by the data and will be used to dispatch to the appropriate function. Let's look at an example.

Data Driven Programming

Recall our sketch of algebraic simplification. There we organized the program as a large conditional expression, each branch testing for a type of term -- variable, constant, sum, or product. That type of organization can be characterized as a monolithic algorithm that tests and decomposes its input, taking actions according to some property of the actual parameter.

But what if we were building a system incrementally and wanted to be adding new types of objects as we went along? The current scheme would require that we modify the code, adding a new recognizer and associated code for each new case. This can become quite a sizeable undertaking if several pieces of code utilize this kind of type dispatch.

In particular, let's assume that we what to add a simplification rule for "squares", simplifying

```
(SQUARE <number>)  =>  <number> * <number>

(SQUARE <symbol>)  =>  (MUL <symbol> <symbol>)
```

We would have to add a new clause to the simplifier to check for the new list whose first element is SQUARE, and also add the appropriate simplification code.

We can organize this kind of problem in an orthogonal manner, viewing the fragments of the algorithm which pertain to specific types as, in fact, properties of the type itself. We can bring property-lists to bear here by placing the specific simplification routines on the property-list of the symbol SIMPLIFY, using the various operators (MUL, ADD, SQUARE ..) as attributes. Then, given an expression like (SQUARE 4), we'd look up the symbol SQUARE (the FIRST of the list), extract a function and pass the object (SQUARE 4) to that function, which would then extract the argument (which is the SECOND of the list) and proceed. For example, a property-list pair might look something like:

```
((SQUARE (LAMBDA (OBJ &AUX (X (SECOND OBJ)))
       (COND ((NUMBERP X) (MUL X X))
             ((SYMBOLP X) ... ) )
```

Of course, it now looks like we're falling into the same trap: what happens, for example, if we decide to add complex numbers to our system? We'd expect to apply SQUARE to them, so another clause would go into the function associated with SQUARE? Surely not; in this case, we'll perform the same data-driven trick, but now the dispatch will be on implicit information--the type of the object--rather than on explicit information--the prefix of the data being the symbol SQUARE.

In more detail, all integers can be said to compute their "square" in the same fashion: multiply themselves by themselves. Similarly, all complex numbers will use the same algorithm (or will they? Some may be cartesian, some may be polar.) And so in general, the class of all objects of a specific type can utilize the same algorithm for performing a specified task.

Using LISP property lists, we could, for example, implement the notion of "squaring" by placing a "square" property name on the property list of the class "integer", and associated with that name, a LISP function to multiply an integer by itself. Then, given a request for the square of an object, like:

```
(SQUARE 4)
```

the system would recognize 4 as an integer; it would look on the property-list of the symbol INTEGER for the attribute SQUARE, and finding it there, would apply the associated function to the integer 4. This would only require that we decorate the type symbol (INTEGER, COMPLEX, etc ..) with a property name of SQUARE, and add the appropriate function as property value.

```
((SQUARE MUL) ... )
```

Such a programming technique is at least provocative; can it be implemented? Surely it can. Assume we're given an action (like SIMPLIFY or EVALUATE (which is a subset of simplification)), and we're given a specific object to act upon; we have to extract the "type indicator" from the object. If it's explicit in the data, like (SQUARE 4) then a simple selector will do the job; if it's implicit (as is the information that 4 is an integer) then we have to make it manifest (as in (TYPE 4)). Given that information, then the rest is straightforward: go to the property-list associated with the action; get the function defined for the category of the given object; and apply that function to the object. Thus:

```
(DE DISPATCH (ACTION OBJ &AUX (CATEGORY (EXTRACT OBJ))
                              (PROP (GETPROP ACTION CATEGORY)))
   (IF (NULL PROP)
       (ERROR (STRING "No Action for " OBJ))
       (PROP OBJ) ))
```

where we could have written the messier

```
&AUX (PROP (GETPROP ACTION (EXTRACT OBJ)))
```

but chose to spread the computation, and use the sequential evaluation of &AUX variables. Notice too, the occurrence of the (PROP OBJ) in the last line; it represents the function application. Of course, within that action-function we may encounter a situation that will require yet another DISPATCH.

So the implementation is feasible; it opens some interesting possibilities, since we can add new types of objects and new actions without recoding massive parts of existing programs.

Of course the syntax is messier since we have replaced

```
(ACTION OBJ)    with

(DISPATCH ACTION OBJ)
```

but the benefits are non-trivial: we have "decoupled" the functional object from its application by using DISPATCH to discover the actual function. It would be nicer not to see the physical call on DISPATCH. We could cover it up by writing something like

```
{ACTION OBJ}
```

instead, letting the system translate the brackets into the call on DISPATCH; later we'll show how TLC-LISP read-macros can do this for us.

But it would be nicer still to generalize this idea, letting the system do more that just fondle property lists. We'd rather see the system handle much more of the decision-making about which function(s) are appropriate. Notice that the places we apply our technique are those that involve a conditional expression--LISP's decision-making operation. Let's continue with an example from the world of simplification, this time using lists and a reformulation of our earlier definition of APPEND:

```
(DE APPEND (X Y)
  (IF (NULL X)
      Y
      (CONCAT (FIRST X) (APPEND (REST X) Y))))
```

We can look at this definition as a specification of two simplification rules:

```
(APPEND () Y)  simplifies to   Y

(APPEND (X Y) Z)  simplifies to  (CONCAT X (APPEND Y Z)))
```

which could be incorporated into our p-list scheme (with a small act of faith) as the following properties of SIMPLIFY:

```
SIMPLIFY
 ((APPEND-NULL (LAMBDA (OBJ) (THIRD OBJ)))
 ((APPEND-XY  (LAMBDA (OBJ)
               (CONCAT (FIRST (SECOND OBJ))
                       {SIMPLIFY '(APPEND ,(REST (SECOND OBJ))
                                          ,(THIRD OBJ))}))
```

where DISPATCH has to be clever enough to extract APPEND-NULL from (APPEND ( ) ...) and extract APPEND-XY from any other (APPEND ...) list; more about this in a moment.

The other mystery is the incantation inside {SIMPLIFY ...}; it is meant to build up a new (APPEND ...)-object.

So {SIMPLIFY (APPEND '(A B) '(C)) goes to

```
   (CONCAT 'A {SIMPLIFY (APPEND '(B) '(C))})
```

This second issue is easily handled by read-macros. The appearance of the names APPEND-NULL and -XY really indicates that we're reaching the limit of the p-list representation. We don't really want names; what we really need instead of DISPATCH is a general pattern-matcher, that can recognize, decompose, and dispatch to routines. This brings us into the realm of

* Pattern-directed invocation of functions

* Data bases of simplification (or reduction) rules that are accessed by pattern matching

* A potentially "reversible" control structure mechanism, since several rules may present themselves as matching the pattern.

Such are the possibilities that spring from our generalization of property-lists. Such are the techniques that drive the implementations of languages like PROLOG, like CONNIVER, and like the precursor of both--PLANNER. We'll not go into these areas here; they're a topic in their own right. We simply want you to be aware of the ideas that make such languages possible. Now we want to drive the data-driven techniques in (what seems to be) a different direction--Class Systems.

Examination of the previous examples reveals a deep similarity: the operation for a specific object is performed by extracting a function that works for any object that shares appropriate similarities with the individual. The similarity may be as general as "it's an integer" or as specific as "it's a non-empty list". Regardless, we can think of these similarities as partitioning our objects into classes, and then individuals (as a result of being a member of a particular class) has access to the functions defined for that class. When these ideas of class and class membership are formalized and generalized, we enter the realm of Class Systems.

Such a Class System exists in TLC-LISP and is the subject of our next section.

Classes

We have mentioned that property-lists are not first-class objects in TLC-LISP (or in most other LISPs). That is, a property-list cannot be referenced without naming the symbol that "owns" the p-list. This impurity is an historical accident, since property-lists were initially built as part of the implementation structure of the LISP system. In the intervening period, property-lists have become useful in their own right as we saw in the last section.

The major distinctions between class-based systems and p-list based systems is the degree of structure and regularity that can be brought to bear on the situation. A class system encapsulates much that would have to be built up explicitly if we used the p-list approach. Thus (with some fear and loathing) we left p-lists in their un-washed state and, rather than making them first-class, added a (first-class) class system to TLC-LISP.

One application of a class system is an extension of the data-driven notions that we introduced in the property-list section. It builds upon both the data object and functional object perspectives that appear in TLC-LISP. Thus it is a blend of program and data objects. But let's begin at the beginning with the notion of class as an abstract idea.

The notion of "a class of objects" is an ancient one. Basically, the human penchant for categorization (or classification) drives us to find similarities among the objects of the real world. These similarities encapsulate the properties that all elements of that specific class share -- your typical dog has four legs, two eyes, hair, and barks. Yet each individual in a class has its own unique characteristics -- no two dogs are identical. They have different names, hair color, weight, and so forth. Furthermore, classes of objects share similarities just as individuals do -- there is a relationship between dogs and horses: both are mammals, have four legs, and hair. The class of dogs and the class of horses are both subclasses of the class of four-legged-mammals.

So within a classification we have several situations:

1. Individuals -- each is unique, but probably shares properties with other individuals in the classification

2. Classes -- a grouping of individuals according to some of the properties that they have in common.

3. Sub-classes -- a refinement of classes, allowing the description of some subset of individuals of a class as a class itself.

All these notions have been well-known in the arts and sciences for a very long time -- even before the invention of the IBM704. However, the application of these ideas to programming has been a reasonably recent event. Simula first, and later several AI languages and Smalltalk have employed class ideas to simplify the programming process.

Class systems lead to a very "descriptive" programming style. Basically, we can use such systems to define data objects and the operations that are to be performed on them. In Smalltalk and TLC-LISP parlance, these operations are named by "message selectors", the operations themselves are called "methods" and the act we perform to apply one of these methods is called "sending a message." A message consists of a sequence whose first element is a message selector and whose remaining elements are passed to the instance to be used as actual parameters to the method.

For example, if W is an instance of a class of display windows then we might expect that: (W ':NEXTLINE) would move W's cursor to the next line in the window, and (W ':GOTO 2 3) would move the cursor to the second column, third row of W. We might represent such a pair of message receivers thus:

```
:NEXTLINE (LAMBDA ()
                (SETQ YPOS (ADD1 YPOS)) ; after boundary check
                (SCREEN-GOTO XPOS YOPOS) )

:GOTO (LAMBDA (X Y)
            (SETQ XPOS X)
            (SETQ YPOS Y)
            (SCREEN-GOTO XPOS YPOS))
```

where the actual implementation of the methods :NEXTLINE and :GOTO are invisible to the user; and further the variables XPOS and YPOS may also be hidden from view. This hiding may be accomplished by making XPOS and YPOS "instance variables". Thus

```
(INST class instance-variables)
```

is a construct to create a unique object that is an instance of the specified class and that has several local variables that are its own private domain.

For example:

```
(INST WINDOW-CLASS '(XPOS 0 YPOS 0))
```

will make an instance of the class WINDOW-CLASS with XPOS and YPOS initialized to 0. (INST will "coerce" the list that appears as its second argument, into an object of type ENV.)

Besides being a good vehicle for hiding implementation details, the Class system can be used to share information. Sharing can occur in two ways. First, when we define a subclass of a class we are able to share the methods defined in the super class. Thus the simplest TLC class definition is:

```
(CLASS superclass local-methods NIL NIL)
```

Thus, when a message selector comes into an instance, the local method environment is examined for a matching method name. If found, that method is used; if none is found the search continues with the superclass.

Of course we are also free to over-ride any of the methods that we wish. The TLC Class concept also gives provisions for sharing information between instances of a class. These variables, called class variables, are available for communication between instances of a specific class; in contrast, instance variables hold values that are visible only to each specific instance. Thus the most general form of a class definition is:

```
(CLASS superclass
       local-methods
       class-variables
       instance-variables )
```

where the methods and variables are lists of alternative names and values. The instance variables are included here to allow specification of their initial values. We may override the initialization when we create an instance, however.

One important aid to understanding classes is to recognize the role played by closure objects in the intellectual history as well as the implementation of Class systems. In particular, the handling of instance variables is exactly the problem that is solved by closure variables: making values available in a local context, and hiding their values from the external world.

To help tie the notions of Class to the notions of Closure, we'll derive a class representation for dotted pairs, from a Closure-based implementation. That's pretty strange in itself!

```
(DE PAIR (X Y)
  (CLOSURE (LAMBDA (MSG)
               (SELECTQ MSG
                        (CAR X)
                        (CDR Y)))
           (ENV 'X X 'Y Y)))
```

Now to build a pair we say:

(SETQ XX (PAIR 1 2)), and to get the CAR or CDR part we say:

```
(XX 'CAR)  =>  1   or

(XX 'CDR)  =>  2
```

With this example we have completely reversed the roles of function and data--the dotted pair is now a functional object, and the selector functions have become message names. We could continue this reversal, showing that even operations like RPLACA and RPLACD could be represented in this scheme; in particular, they would become message names (like CAR and CDR), but their action would be to set the appropriate closure variable (X or Y) to the new value. Rather than continue this line, we want to apply these basic ideas to the construction of a Class-based version of PAIR.

Class Example of Dotted Pairs

We want to define a class representation for dotted pairs. Each pair will create an instance of the class. Such instances will have two instance variables: one to hold X (called HEAD), one to hold Y (called TAIL).

```
(SETQ DOTTED-PAIR (CLASS NIL
                         <methods for the class>
                         '(HEAD NIL TAIL NIL) ))
```

The important ideas are contained in the following dotted pair operations:

(PAIR X Y) -- construct a new object.

defined as:

```
(DE PAIR (X Y)
    (INST DOTTED-PAIR (ENV 'HEAD X 'TAIL Y))))
```

Coming "right out of the blue", such a definition might seem totally mysterious; but viewed from our experience with the CLosure-style definition, this PAIR is just a restatement of the previous PAIR.

Now what about the selector functions?

(CAR DTPR) -- select the first component of a dotted pair. Select the value associated with DTPR's HEAD instance variable.

We will do this by sending the message CAR to the instance. Thus:

```
(DE CAR (DTPR)
    (DTPR 'CAR) )
```

and so one message-method pair in the class DOTTED-PAIR should be:

```
CAR (LAMBDA () HEAD)
```

CDR would be done similarly. So we could describe our class by:

```
(SETQ DOTTED-PAIR (CLASS NIL
                         (ENV 'CAR (LAMBDA () HEAD)
                              'CDR (LAMBDA () TAIL)
                               . . . )
                         NIL
                         (ENV 'HEAD NIL 'TAIL NIL)) )
```

A more interesting operation is one to update components within a dotted-pair object. Consider:

(RPLACA DTPR Y) -- replace the first element of DTPR with Y.

How can we implement this? Within the instance representing DTPR we have two local variables, HEAD and TAIL. The RPLACA operation need simply replace the value of the local HEAD with Y. Thus:

```
(SETQ DOTTED-PAIR (CLASS NIL
                         (ENV 'CAR    (LAMBDA () HEAD)
                              'CDR    (LAMBDA () TAIL)
                              'RPLACA (LAMBDA (Y) (SETQ HEAD Y))
                               . . . )
                         NIL
                         (ENV 'HEAD NIL 'TAIL NIL)) )
```

and define RPLACA by:

```
(DE RPLACA (DTPR OBJ)
       (DPTR 'RPLACA OBJ) )
```

Notice that the locality of the values associated with HEAD and TAIL is absolutely crutial for the proper implementation of dotted pairs as class instances. If the values for HEAD or TAIL were shared between instances the implementation would not work.

Further food for thought: notice that we have introduced a rather dynamic quality to the creation of dotted pair objects. We introduced these objects as data structures --typically thought of as static storage quantities. With the class description, we have shown an alternative interpretation wherein the dotted pairs have become functional objects based on the notions of closures. The line between data objects and functional objects is indeed fine.

We now move on to another Class example that will show several alternatives of representation.

An Example of Turtle Graphics as a Class

A LISP dialect that is getting a lot of attention lately is Logo. In this section we show how to define a class of Logo-like turtles, based on the ideas in TLC-Logo.

Assume that we have a single dot-drawing, turning turtle, that has is contrtolled by the following graphics primitives:

(PENCOLOR <color>) Sets the color of the pen to the integer <color>.

(PENUP) and (PENDOWN) Raise and lower the turtle's pen.

(MOVE <dist>) Moves the turtle forward <dist> units. <dist> must be non-negative.

(MOVETO <x> <y>) Moves the turtle in a straightline to the point <x> <y>.

(TURN <angle>) Turns the turtle counter-clockwise <angle> degrees.

(TURNTO <angle>) Turns the turtle to <angle> degrees, where 0 is straight up.

We'll have to simulate the multi-turtle behavior using the single "hardware" turtle. Thus, every time we want to manipulate a turtle we'll have to make sure that the hardware turtle is mapped onto that turtle. The mapping is accomplished by setting the heading, position, color, and pen state of the hardware turtle to the desired values. We must recall that movement will leave a trail when the pen is down, and thus a function

```
(DE SETURT (XPOS YPOS COLOR HEAD PEN)
   (PENUP)
   (MOVETO XPOS YPOS)
   (PENCOLOR COLOR)
   (TURNTO HEAD)
   (IF (EQ PEN 1)
       (PENDOWN)) )
```

will set the turtle.

We can do the actual mapping in several ways:

1. Always map the hardware turtle to the desired turtle. This means that every operation on a turtle will be prefaced by a call on SETTURT. This will slow down every turtle operation.

2. Check to see if the last operation was on the same turtle. This would be a good place to use a class variable.

3. The final variety we'll consider is potentially the most efficient and in some ways the least satisfying of the bunch. Namely, always assume that the hardware turtle is in the right place. Therefore, require an operation that explicitly changes from one turtle to another, and that operation will be responsible for keeping multi-turtle information current.

In this scheme, there is only one turtle that we can address at any time. We don't need (TUR 'FD 10); (FD 10) will suffice where FD moves the hardware turtle. (TUR 'FD 10) can be factored into (ASK TUR) followed by (FD 10), and all turtle commands will be interpreted by TUR until another ASK is given. In the section How LISP Works we will see that this scheme is similar to an implementation technique called shallow-binding that always keeps a special cell (called the value cell) loaded with the current value of each variable.

We can implement a turtle as a vector of values, implement the hardware turtle's state as a vector, implement the turtle functions as operations on that distinguished vector, and implement ASK as a "vector swapping" function.

For example, let CURRENT-TURTLE be the name of the current turtle. Then:

```
(DE ASK (TUR &AUX (CUR CURRENT-TURTLE)) ; keep a reference
    (SETQ CURRENT-TURTLE TUR)
    CUR)
```

where we save the current value as we switch to the new turtle, returning it as value so that whoever called ASK can save that prior value.

Hatching a new turtle is like cloning, except we give the new turtle its own name. It retains all other properties of the current turtle.

```
(DE HATCH (TUR &AUX (X (COPY CURRENT-TURTLE)))
  (STORE X NAME TUR))
```

Finally, to demonstrate a turtle motion, FD (for ForwarD) should update the position of the current turtle and then exercise the graphics code to move the screen representation:

```
(DE FD (DIST)
  (STORE CURRENT-TURTLE XPOS
                        (ADD (CURRENT-TURTLE XPOS)
                             (* DIST (SIN (CURRENT-TURTLE HD)))))
  (STORE CURRENT-TURTLE YPOS
                        (ADD (CURRENT-TURTLE YPOS)
                             (* DIST (COS (CURRENT-TURTLE HD)))))
  (MOVE DIST) )
```

These examples are by no means complete or exhaustive; they only indicate the kind of abstract programming that Class Systems lets us explore. For further information see the Class System section of the Reference Manual, and for many more details see the TLC-Logo books and documentation.

This third case is the representation that we chose for TLC-Logo. The earlier representations are in keeping with the message-passing style of Smalltalk.

Classes are a powerful idea, supplying a descriptive and modular way of dealing with data objects. In fact, the type of system we have described here has been criticized for being too modular; each class has a single super-class, and thus can inherit methods only from classes above it in the hierarchy. At times this can be inconvenient, and extensions of the simple class system, allowing multiple inheritance have been proposed, designed, and implemented. Only experience will show how much of this generality is really effective.

Catch and Throw

Just as the kernel ideas of functional objects are being applied in wondrous new ways for building descriptive data programming, so too can functional ideas be used to build high-level control structures, like multi-processing, co-routines, and cooperating process descriptions.

The two most famous features of LISP are:

* Its parentheses

* Its use of data as programs.

We know all about parentheses, now -- LISP is a graphical language trapped in linear format.

We know some of the benefits of having executable data, or more accurately, the ability to represent programs as data.

The feature we wish to highlight here is the use of control information as data. Control information is the computational stuff that allows the LISP evaluator to find its way through the function-call/function-return jungle. The evaluator has to be conscious of several things while computing (f a1 a2 ... an):

* where to put values of the actual parameters as they are computed.

* when all values are present, combine them so that the function named f may get to them.

* remember how to get back from the evaluation of the body of f, for someone is waiting for that value.

On the surface, it seems that this detail is only of interest to the evaluator and the user has no business with any of it. But consider the role of a debugging program; we realize that it has use for the program-as-data facility so that the user may modify errant programs. And how does the user discover the problem? First, one must discover which function was active when the error occurred, what arguments we passed to it, and what function called the misfortunate function. All this is control information. Therefore, a choice must be made: write the debugger in a lower-level language that has access to this information, or lift control-information primitives into the surface language. LISP-like languages tend to opt for the latter course, since this is just another example of our representation problem: represent the objects, and define functions to operate on them.

In TLC-LISP, we have primitives to supply representations for the state of the internal "LISP machine"--what were the arguments, what were the functions, etc.--the kind of thing a debugger needs.

Now let's think about the functions that will control a debugger; since it is unrealistic to suppose that we can predict the occurrence of a bug, we need a different kind of control function--something other languages call an "exception condition". We need a construct that will say "go evaluate this expression and let me know if some specified anomaly occurs". If nothing untoward happens then the exception handler passes the result on wihtout comment; if the specified exception does occur then the handler is made aware of it, and can taken appropriate action. This notion of exception handling has been generalized in LISP and appears in several dialects (including TLC-LISP) under the function named CATCH.

A CATCH call appears as

```
(CATCH label expr ...  expr),
```

where the label is a symbol that names the particular "exception" that this expression will handle. The simplest invocation is of the form

```
(CATCH ERROR  expr ...  expr)
```

because the system will generate exceptions named ERROR whenever it encounters an error condition. So if any of the expressions in the body of this CATCH-expression generate an error, the evaluation will stop immediately, and the value returned for the CATCH will be some indication of the specific error.

This type of behavior has been in LISP systems since the early 1960s under the name of ERRSET. In fact, these early systems also allowed the programmer to explicitly cause errors by calling a function named ERR; This pair of functions, ERRSET/ERR, soon became overworked; too much time was spent in decoding what kind of "error" was occurring--was it a real one, or one generated by the programmer? Thus CATCH with a specifiable label was born; and in conjunction with it, the function THROW replaced ERR. So both CATCH and THROW expect a symbol as their first argument, and they work as a pair to loosen the function-call/function-return regime of the pure functional languages. In their most general form, they can be used to describe multi-processing operations in a very clean, high-level fashion. See the Abelson&Sussman book or Wand's paper for details. However, as with functional objects, most LISP implementations do not implement the full power of these control objects. TLC-LISP is no exception; it does, however, implement a very useful, and efficiently implemented subset of the ideas.

Now let's see what that implementation has to perform. As part of the invocation of CATCH, the system makes note of the occurrence of label before beginning the evaluation of the expr's in the body. If no THROW expression is encountered in that process, the value of the CATCH is the value of the last expr. However if we encounter a (THROW label expr-1 ... expr-n), then the system retreats to the CATCH, and returns with the value of last expr, exp-n.

This CATCH-THROW pair is much like a call-return pair, except that the returning THROW may bypass several intervening function calls. This is the level of CATCH-THROW that is supported in the current TLC-LISP.

Given the ability to manipulate control objects as first-class object, we can describe debuggers in this high-level notation. That's an immediate practical benefit, just like program-as-data makes the description of editors more manageable. At a higher level, such tools open the door to introspective systems--those that are able to analyze their own behavior, and perhaps modify it. Part of the analytic power is the ability to grasp the components of execution; the other part requires that the system have some model of its own behavior. A particular example of this is the InterLISP DWIM package.

DWIM--standing for "Do What I Mean"--comes into play when an InterLISP program has discovered an error. The explict control objects allow DWIM to examine the function-calling history, the current state of the evaluated arguments, and the expression under consideration when the error was discovered. Using this information, and some built-in knowledge about LISP programs and common LISP errors, DWIM proceeds to analyze the situation. For example, on most keyboards the parenthesis keys are upper-case; often, a LISPer will miss a shift and get the lower-case character instead, therby throwing off the nesting of the LISP expression. Often DWIM can detect this kind of situation, reconstitute the errant symbol after removing the parenthesis, and re-parse the expression.

Often, the computation can be backed up to a prior point and continued without having to lose valuable time and effort. This is a simple example of a DWIM-type correction. As the importance of control objects becomes more widely understood, this type of introspective system will flourish.

The important point to remember is that "control-as-data" is distinct from "program-as-data".

Evaluation

With the previous sections as background, we can present a more detailed description of the LISP evaluation process.

The family of LISP expressions consists of the following:

CONSTANTS: 1 T '(1 2 3) car (or CAR) "xyz" [1 2 3]

These are constants of the classes: number, truth-value, list, function, string, and vector respectively (in an implementation, "constants" like CAR may not really be "constants" -- they may actually be implemented as variables, and therefore subject to redefinition by the user. Of course such user actions are discouraged when attempted on very primitive LISP operations but, in keeping with the open nature of LISP, such actions are seldom explicitly prohibited.)

VARIABLES: X FACT

These are variables which might be found naming simple values and functions respectively (recall that variables are type-free, but objects are typed)

COMBINATIONS: (CONCAT 'A (FIRST L))

This combination represents the application of the function constant CONCAT to two arguments: a constant, and another combination.

CONDITIONALS: (IF X (CONS X L) NIL)

This conditional expression returns the value of combination (CONS X L) if the value of X is non-NIL. Otherwise NIL is the value of the expression.

Elegant simplicity!! As a result of LISP's simple syntax, the evaluation process is equally uncluttered. An even more pleasing property results from LISP's inclusion of program elements as data items: we can write the evaluation process in LISP itself. We won't carry out this last step here; it is an exercise which every LISP programmer should perform. Here we will only sketch the process and highlight the non-trivial spots.

1. The evaluation of constants: Any constant simply evaluates to itself. A certain amount of care needs to be taken: though strings, vectors, and numbers are recognizable as constants from their appearance, we also need to be able to differentiate between constant S-expressions and S-

expressions which are representing elements of the LISP language. This problem is the origin of the QUOTE operator.

Note that besides simple constants like S-expressions, numbers, vectors, and strings, LISP also has "functional constants" like CAR and COND. The term "constant" simply means predefined; all these predefined functions may be re-defined, though of course flagrant refedinition of LISP primitives will lead to obscure programs at best, and system destruction at worst. On the other hand, tasteful redefinition can be useful. For example,

```
(LET ((PRINT NEW-PRINT)) ...(PRINT ...) ...)))
```

will use NEW-PRINT instead of the system-defined PRINT within the body of the LET-expression. This could be helpful in redirecting output for other purposes. This redefinition of system-level functions is a special instance of "dynamic scoping" --LISP's default strategy for evaluation of variables.

2. The evaluation of a variable: LISP variables are "type-free" meaning that a variable is free to take on any type of value --number, string, list, vector, or even a class or function. It is the value which carries the type information; and it is the context in which a value is used which determines whether or not a "type restriction" is satisfied. For example, an error is signalled if one attempts to apply a string as a function. This means that the evaluation process for variables is reasonably straightforward: using the variable name, extract its value from the current environment.

Of course things are not quite all that simple: The conceptual issue raised by LISP is when to find the values; a few sections from now we will discuss the "how" of the programming techniques used in implementing LISP's variable binding, but here we restrict ourselves to conceptual questions. The issue is one of scoping rules. Scoping rules come into play when one adds function definitions to our system; in particular, the question involves free variables: variables which are not formal parameters of the definition.

Algol-like languages (including Pascal and ADA) use a static scoping rule: locate values of free variables at the time a function definition is installed in the system. This rule relates well to those languages with a penchant for compilation, since a compiler must be able to generate code from static text.

LISP defaults to a rule called dynamic scoping which says locate the values of free variables at the time the function is applied. This rule fits in well with LISP's interactive style of program development, since in LISP programming one frequently begins executing program fragments before all components are defined. This programming style is called "middle-out" as compared to "top-down" or "bottom-up".

Unfortunately, the issues of scoping rules are clouded. From a theoretical perspective, the correct rule is static scoping, and dynamic scoping is a bug; actually, in practical settings dynamic scoping is a bug as soon as we deal with functional objects in the context of arguments to, and values from, LISP functions, or in the implementation of Classes. Yet in the interactive development of programs --like the NEW-PRINT example-- dynamic scoping is definitely useful. The last word on scoping rules has not been said (or at least heard).

3. Combinations: A combination, also called a function application, is evaluated in a call-by-value fashion. That is, the function position is evaluated, assuring that a functional object is available there; then each of the actual parameters is evaluated in a left-to-right order before the function is applied. Note that this description of evaluation is recursive: the evaluation of a combination involves evaluation of all of the components of the combination. Typically, that process will terminate with values to continue the computation. If the called function is a primitive, then these values are passed to that function.

For example, consider: (CDR (CAR '((A . B) . C))) or its unabbreviated form (CDR (CAR (QUOTE ((A . B) . C)))).

The evaluator would come upon the form (CDR ...) first. Evaluation of CDR yields a functional object; however the operand of CDR requires further evaluation. It itself is a combination: (CAR ...). The evaluation of CAR yields a functional object. Now consider the evaluation of the argument to CAR; this time we encounter QUOTE. QUOTE is handled specially (see 4, below); QUOTE always returns its argument unevaluated; this time it is the constant ((A . B) . C). We apply CAR, getting (A . B). This value is finally passed to the outer CDR, resulting in B.

This example is typical of what happens in calling primitive functions. If the called function is a user-defined function, then added care must be taken.

A user-defined function has the following internal structure:

```
(LAMBDA (<param-1> ... <param-n>) <body>)
```

where (<param-1> ... <param-n>) are called formal parameters and the <body> is a sequence of LISP expressions. The complete unit is called a lambda expression. LAMBDA is a reserved word indicating that the material which follows it represents a function.

Once the values of the actual parameters are computed, the current values of the formal parameters of the called function are saved, and the evaluated parameters are then associated with the formal parameters; this process is called lambda binding. After the lambda binding is completed, the evaluation of <body> is performed. Upon completion of that evaluation the values of the formal parameters are restored to the values which were current when the function was entered. For example assume the variable X has value 5 and consider:

```
((LAMBDA (X Y) (CONCAT X Y)) 'A '(1 2)) => (A 1 2)

(ADD1 X) => 6
```

To evaluate the first line we save the values of X and Y; bind X to the atom A and Y to the list (1 2); note that besides getting a new value, X also gets a new type. We evaluate the CONCAT expression, returning (A 1 2), and we restore X and Y. The evaluating of the ADD1 expression yields 6.

Of course all of this description is highly oriented towards some mechanism to carry out the computation. There is a very non-process interpretation of functionality as well. This model is based on the notions in Church's lambda calculus, and hinges on the idea of substitution--replacing objects by objects so that meaning is preserved.

However as we've seen in the previous sections, a characterization substitution is difficult, and as we've seen in the closure and class sections, simulation (rather than explicit substitution) has distinct advantages.

4. Closures. Closures are a generalization of the previous case of combinations. There are two conditions to discuss here:

a. The evaluation of a

(CLOSURE function-text environment) object.

b. The evaluation of a combination that has a closure object in its function slot.

The first condition is reasonably straightforward; the system simply builds an internal structure to carry the text and the environment.

When a closure is to be applied as a functional object, the names and values of the closure environment take precedence within the evaluation of the body of the closure. For example,

```
((CLOSURE (LAMBDA (X) (ADD X Y)) '(Y 3)) 5)
```

will associate X with 5, then add the information that Y has value 3, and proceed, then, to evaluate (ADD X Y), getting 8. As we leave the closure combination, we restore whatever binding X and Y had.

For a more complex example, consider:

```
((CLOSURE (LAMBDA (X) (SETQ Y X)) '(Y 3)) 5).
```

This case proceeds as above until we execute (SETQ Y X). Here, we assign a new value to Y, and as we leave the combination we restore Y and X. But note here that the closure environment has been modified; the next time we apply the closure, Y will have value 5. This sense of locality that the closure environment grants us is at the heart of the notions that implement class systems. We will examine class-related evaluation a couple of paragraphs hence.

5. Special Forms. Special forms have the appearance of combinations: e.g., lists with a function-like object in the function-position. However, special forms are not combinations in the sense of 3. Combinations evaluate their arguments; whereas special forms pass their arguments as unevaluated data structures, and it is up to the special form to process the arguments. If FOO is defined as a special form, then the call

```
(FOO (CONS 2 (ADD1 4)))
```

would result in passing the list

```
((CONS 2 (ADD1 4)))
```

--not the value (2 . 4)-- to FOO for processing.

If evaluation is desired, then the LISP evaluator must be called explicitly.

There is a popular misconception that special forms are "call-by-name" functions. They are not the same. Primitive special forms of TLC-LISP include the COND, QUOTE, and IF constructs. IF and COND evaluate only a selected subset of their "arguments", while the purpose of QUOTE is to stop evaluation altogether.

Again, the description of IF and COND, given in the body of the TLC-LISP manual, will transform into simple LISP algorithms that we can add to the evaluation routine.

The above cases represent the basic evaluation algorithm of a LISP implementation.

It is most strongly recommended that the reader specify such an algorithm. The subtle point to contemplate in such an endeavor is LISP's treatment of functional objects. The interplay between such objects and the scoping rules is most interesting and worthy of a serious reader's time.

These LISP evaluators give an operational semantics, or meaning, to the programming language constructs. Put another way, the four steps compose the central processor of a simple LISP machine. There are two missing ingredients in the machine: first, the machine instructions; these include the data manipulating and testing instructions --CAR, CDR, CONS, ATOM, and EQ-- as well as the control instructions --QUOTE and COND. All other LISP operations can be defined in terms of these operations. The second missing component of the machine is the "microcode" to run the CPU: that is the business of the section "How LISP works".

Around this kernel called "pure LISP" is built a powerful, pragmatic programming tool. The next few sections, and the remainder of this section discuss some of those features.

The LISP we have discussed so far differs substantially from the traditional view of programming: there are no assignment statements or iterative constructs. More generally there is no concept of "state" or "side-effect".

Every "non-toy" LISP, including TLC-LISP, has included a healthy portion of traditional programming techniques. We will leave the details of these artifacts to the manual and will restrict our attention to some of the difficulties they cause in language design and implementation.

First, the concept of "state". The most common manifestation of "state" in programming languages involves the assignment statement. That construct views the world of variables as a collection of slots, each of which can contain a value (or if its a truly enlightened language, an object). We move through the computation, extracting values from the slots, modifying them, and placing them back in slots. This is a very "undisciplined" view of variables as compared with the "structured" access of variables present in pure LISP. The binding mechanism of LISP matches variable accesses with the control flow of function entry and exit; in contrast, assignments are often allowed to occur in a totally arbitrary way. This has detrimental effects at the theoretical end of the spectrum, in language implementation considerations (see "How LISP Works"), and even impacts on "sociological" issues of programming style.

The most well-known attribute of an assignment statement is its ability to cause a side-effect, meaning that it will affect the state of the computation outside of the current environment. For example, if a side-effect occurs, one cannot guarantee that two executions of the same piece of code will give the same result since the state has been modified. "Impure" LISP has both assignment statements to modify the state, and operations to modify data structures. These are related, but not identical ideas. For example, in a language like FORTRAN we can allocate an array such that the same array is referenced by two different variables, IX and IY, then changing an array element through IX changes a value in IY. This is a problem of sharing values called aliasing. Sharing of values is not problematic provided one cannot modify values.

Of course, we now know that one should not think about quantities that get modified as being values; it is better to think of them as objects--in fact mutable objects. In this way it becomes more natural to think of constructor operations as producing objects and later those objects may be mutated by applying an updater. Thus the CONS operation makes a new cell and copies the arguments into the CAR and CDR-parts (for more details see "HOW LISP works").

Modification operations introduce large impurities into LISP (or any language); but we realize that change and state must be considered in a "real world" language. Our concern takes two forms: first that the user understands the scope and power of operations that can change existing structures, and applies them in a "self-controlled" fashion. Second, we wish to explore language features that, while not imposing fascist regimentation, will encourage a localization of side-effects and state change. In this light we find the Class and Closure ideas most promising.

Classes, Closures, and applications contain an interesting combination of the functional view with the side-effect view. We have hinted at this combination in the evaluation of closures. We'll now be more explicit. A class definition simply builds a structure that contains all of the information about the super-class link, the local methods, the defaults for instance variables and a potpourri of the class variables that have been specified for this class or its super-classes. Instancing simply spins off a closure-like object, except that the function slot indicates the class, and therefore an implicit function, rather than the explicit function we would find in a closure object.

The real work occurs when a message is sent to an instance. This is handled like a combination: the parameters are evaluated in the current context; then the message name is used to index into the method table, resorting to super-classes if the method is not found locally. Assuming that a matching name is found, the class variables for that class are installed, and then the instance variables are overlayed. The method body is evaluated, and then the instance and class variables are saved away. This process is identical to that performed for closure invocation. That's not surprising since the message-passing metaphor can be described directly as an application of closures.

Macros and Backquote

We will close this section on a milder note, discussing some added styles of evaluation. Besides the two basic styles of application (call-by-value combinations, and special forms), many LISP's include a macro facility. Since we consider LISP an assembly-level language, it is only fitting that it have a macro capability similar to that enjoyed by many other assemblers. A traditional assembler utilizes macros as an abbreviational device such that the macro is "expanded" at the time the text is assembled. LISP doesn't really assemble, but interpretively executes the internal form of the list structure; therefore LISP macro expansion occurs at run-time. When a macro call is recognized, the instructions in the body of the macro are carried out; these instructions transform the call into another piece of LISP code, and then the evaluator executes this new code. LISP macros are a very powerful programming technique, passing programming details off to the machine.

For example, though in LISP we have the CONS operation to construct new S-expressions, we most usually wish to deal with lists. In that regard, we have a function named LIST whose purpose is to take an arbitrary number of objects and build a list from them. Though LIST is already defined in TLC-LISP, let's proceed as if it weren't. One solution we've already seen is to utilize our extended parameter syntax, and define LIST as:

```
(de LIST (&REST l) l)
```

But assuming we'd like a different (or more explicit) solution, we might recall that a list can be constructed by a sequence of CONSes. Thus:

```
(CONS 1 (CONS 2 (CONS 3 () ))) = (LIST 1 2 3)
```

This kind of "textual equivalence" has long been exploited at the assembly language level. In the early 1960's T. Hart introduced this powerful macro facility to LISP. The essential idea involves LISP's program/data duality: the data-structure representation of the actual function call is passed to the function as its parameter.

In the above example, the call (LIST 1 2 3) would pass the list (LIST 1 2 3) to the LIST macro. The list structure will be decomposed, reconstituted into (CONS 1 (LIST 2 3)) and returned for further evaluation. The evaluator can process ((CONS 1 ...) but will call the LIST macro again for (LIST 2 3), resulting in (CONS 1 (LIST 3)).

Finally (LIST 3) will decompose into (CONS 3 ()), and the process will terminate after evaluating

```
(CONS 1 (CONS 2 (CONS 3 ())))
```

Notice that the macro expansion process involves substantial use of the program/data duality and it is all carried out without user intervention.

Of course the critical ingredient is missing: what does a macro definition look like, and how does the evaluator process it? First, the definition: we've already said that what gets passed to the macro is the actual call in list-form, so the definition should not look too foreign.

```
(dm LIST (l)
  (if (null l)
      ()
      (concat 'cons (concat (second l)
                            (concat (concat 'list (rest l 2))
                                    () )))))
```

The LISP evaluator also has to cooperate, recognizing that the value returned by a macro must be evaluated again. This is a straightforward expansion of the repertoire of function-types that the evaluator handles.

Since the body of a macro has a tendency to be messy, in tearing down one expression and building up an expanded form, an abbreviated syntax has been developed. This notation, called "back-quote" (‘) works as an "anti-quote" so that unquoted structures within its scope are assumed to be constants, and when we want expressions within its scope evaluated, we have to indicate that desire; one particular way is to decorate an expression with a comma prefix (,). So compare the following definition of LIST with the non-backquote version.

```
(dm LIST (l)
  (if (null l)
      nil
      ‘(cons ,(second l) ,(concat 'list ,(rest l 2))) ))
```

That looks better, but still there's some explicit list-construction that clutters up the appearance. In particular the application of CONCAT within the comma-ed expression is only there to indicate that we want to combine the elements of the REST-expression with the symbol LIST. We couldn't get this effect by writing

```
(list ,(rest l 2))
```

since that would give list of two elements--first one LIST and the second, the result of REST. For example we'd get

```
(LIST (1 2 3)) not  (LIST 1 2 3)
```

So we extend the macro notion a bit further, adding an at-sign (@) which is used in conjunction with comma to mean "splice in" rather than "cons in". So we can finally write LIST as:

```
     (dm LIST (l)
       (if (null l)
           nil
           '(cons ,(second l) (list ,@(rest l 2))) ))
```

The actual definition of backquote and its auxiliary functions are located in the system file "LISP.SYS".

A related idea is called read macros. The read macro is applied at the input phase of LISP programming. A function can be associated with a character; when this character is recognized in the input stream, the function is activated. That function may perform arbitrary LISP computations, including further reading of the input. The result of the read macro is passed to the input stream as if it were the original input. For example the single-quote, ', is a read macro.

```
(DMC-' ( ) (LIST (QUOTE QUOTE) (READ)))
```

is effectively the definition of the read-macro. Note that we can't write

```
(DMC-' ( ) (LIST 'QUOTE (READ)))
```

because that would invoke the read macro before it was defined.

How LISP Works

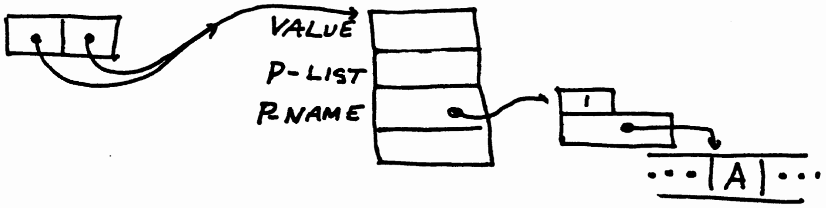
This section is not a description of the implementation of any particular LISP. Rather, it is an overview of several techniques that occur in LISP implementations. Since much of this information is both useful and somewhat difficult to obtain in a cohesive form, it is included here. Its assimilation will improve one's understanding both of LISP and the relationships between the practical techniques of systems implementation and language design.

A LISP machine is best thought of as a calculator: one prepares an input expression, presents it for evaluation, and receives an answer. That input may have a side effect --for example, the definition of a function--, but one always receives an answer. This "top level" of LISP is called the read-eval-print loop, because READ, EVAL, and PRINT are the names of the functions that accept input, evaluate expressions, and prepare output respectively. In the following three paragraphs we will discuss some of the more interesting features of these algorithms.

READ: The LISP reader (also called a parser) has the overall responsibility to transform the external linear list notation into the internal tree-structured representation. Of course the TLC-LISP reader has more to do --numbers must be internalized to a form compatible with the arithmetic unit of the machine; strings are stored in a more efficient non-list form-- but we restrict attention to the primeval reader. Functionally, there are two components to the reader; the most primitive piece is the LISP scanner called SCAN. This routine will recognize the characters special to LISP: for example, space, (, and). SCAN also is responsible for building the internal form of an atom, be it number or symbol.

LISP symbols play a role similar to that of words in a natural language dictionary; in fact since property lists are most usually associated with symbols, the analogy is exact. The property name is a "part of speech"; the property value is the corresponding meaning. A dictionary entry contains all the information about that particular entry, including pointers to other words in the dictionary. The organization of the dictionary is such that we need only look in one place for the meaning of a particular word; without such assurance a dictionary would be useless.

To insure similar organizational benefits in LISP, we require that SCAN make every reference to a particular symbol point to the same dictionary entry. This is accomplished at the time a symbol is created; that is, when the character sequence is to be transformed into a symbol. Given a sequence od characters that represents the name of a symbol, the LISP system will compare this string with strings it has already converted into symbols. This comparison employs some efficient search technique so that not every symbol is compared. Typically, LISP systems seem to prefere a "hash algorithm", though history has as much to to with this decision as anything else. Regardless of the technique, a unique representation for each symbol reference is guaranteed. Thus, for (A . A) we'd have something like:



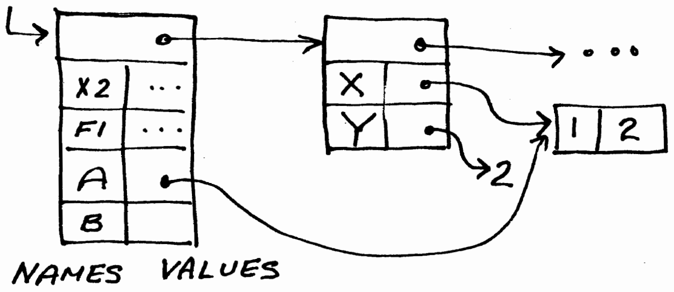
A Representation of (A . A)

where the "print name" structure is a string that contains the actual characters that make up the symbol name. It is called the print-name (or P-name) since it's what get printed for the symbol. (Advanced point: This scheme is a bit more complex when packages are involved, but not much.)

EVAL: The previous section on evaluation discusses the "what" of evaluation; this note describes some of the "how."

There are metaphysical issues and implementation issues that must be addressed in the implementation of a LISP dialect. The deeper concern is that of scoping: traditonal LISP uses dymanic scoping which, as we've seen, can severly damage the power of functional objects. The alternative of lexical (or static) scoping requires a more delicate hand so that power can be dispensed in a fashion that will not damage efficiency. Regardless of the choice of scoping rules, similar decisions face the implementor when we come to the intricacies of variable binding and access.

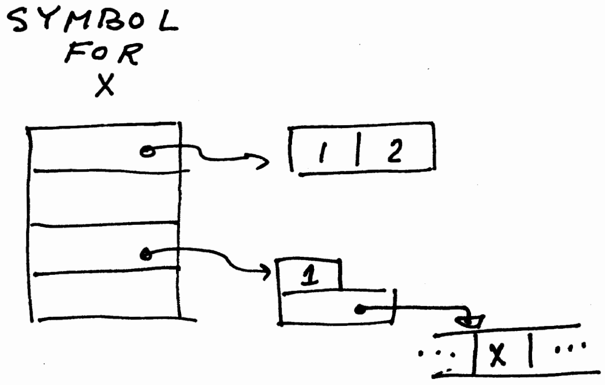
There are two common strategies: deep binding and shallow binding; they correspond closely to the distinctions between standard programming and data-driven programming. In a deep binding-implementation the search algorithm is given a variable name and a table of names and values; it will search for a match in the name column and return the corresponding value as the value of the variable.



Deep Binding #1

In this scheme, each block corresponds to binding of a set of formal parameters ot a set of actual parameters. If the values are not found in the current block, the previous block is searched. Note: this search strategy will work for either lexical or dynamic scoping; the difference occurs only in the way the blocks are chained together.

With shallow binding, we position the value of the variable with the symbol that represents the variable. In this case the search routine need only examine the designated slot in the symbol. The "value property" is always found in the value cell of the variable; no search is required.



Shallow Binding

As with most things, there is "no free lunch". The simplicity of the shallow-bound search is offset by corresponding complexity in the maintenance of the bindings. As one might suppose, the maintenance problem in deep binding is simpler. Recall our discussion of LAMBDA and the binding properties (called "shadowing") that made old values of the formal parameters invisible. The straightforward implementation of deep binding can accomplish this behavior by structuring the table as a list, and encoding the binding rule to add pairs to the front of the list.

The implementation of shallow binding involves a destructive store into the appropriate value cell after saving the old value. The corresponding "unbinding" operations are of comparable complexity. For a complete discussion of LISP implementations see Anatomy of LISP.

Regardless of the binding strategy, a major concern in the evaluator involves what to do with the value that finally gets extracted. The problem is particularly involved in the case of a combination (or function application). First, the function position is examined; if that object represents a call-by-value function, then the arguments (if any) are evaluated in left-to-right order; if the function object is a special form, then no argument evaluation is involved. The next phase involves the parameter passing operation; in most LISP implementations (including this version of TLC-LISP), this involves simple stack, or push-down list, operations. However, the most general LISP must be prepared to do more. LISP's unrestrained use of functions as data objects can force a tree-like, rather than stack-like, behavior on the parameter passing implementation. The quest for adequate implementations of this general difficulty is called the "funarg problem", or "functional argument problem". In TLC-LISP we have avoided the most general situation and implemented the functional subset called closures. See Anatomy of LISP for details of the implementation of functional objects.

A final note related to binding should be discussed here: regardless of the scoping rules or binding strategy, the implementation is such that when we leave a scope the appropriately saved bindings are restored. That is, these bindings follow function entry/exit protocols. Thus, these are distinguished from the bindings which we encounter with assignment statements. These later bindings --called "destructive bindings"-- cut through program structure as surely as the beleaguered "goto" cuts through control regimes.

An assignment-like binding, called SETQ, exists in LISP. Both assignments and gotos are useful programming constructs, but should be used in moderation. Contemporary programming has two legs: the applicative limb, containing recursive programming and the related non-destructive binding and the imperative limb containing iteration and destructive binding. To program effectively we need both legs.

PRINT: Print is the least complex of this trio, converting an internal form to a readable external form. Some of the more interesting print routines do "pretty-printing". That is, they format the output using conventions based on the structural nesting of the expressions.

Memory Management: The final topic of this section is the LISP memory management system. LISP views data as a very dynamic and volatile commodity. Objects are created and destroyed freely and constantly in a LISP program. The major mechanism for creation is the CONS function, that creates a new node in a list structure. The memory management system maintains a data structure called a free-space list. Requests from CONS extract pristine nodes from this list. When that list is exhausted, a storage reclaimer or garbage collector, is called to recover nodes that have been discarded. These recyclable nodes are discovered by scrutinizing the current state of the computation, marking all the data items which are still being used. This process is called the mark phase. It follows the topology of the LISP list structure. The next phase, the sweep phase, follows the topology of memory, visiting every node --both marked, and unmarked. It collects the unmarked nodes into a new free list, being assured that any unmarked node was inaccessible and therefore "garbage". Armed with this new supply of nodes, the manager can now fill the CONS request. For more complete discussions of garbage collection see Anatomy of LISP or Knuth's volume.

LISP as a Systems Language

The traditional vehicle for systems implementation has been assembly language. Given our perspective of LISP as an assembly language (including macros), it is natural to investigate the viability of LISP as a systems development tool. The compulsion becomes stronger when we consider that artificial intelligence programming tends to be among the most complex of tasks and LISP is that field's primary programming language.

What does LISP provide for a systems designer? There is a built-in collection of primitive data structures along with appropriate functions to manipulate those items and build complex objects from components. In a modern LISP, these data objects include: numbers, strings, identifiers, and arrays. Arbitrary precision numbers (bignums) are not included in this version of TLC-LISP. These primitive notions are augmented by operations for constructing new data objects. One may construct new strings and arrays at run-time, combine existing structures into new objects using CONS, and construct record-like structures using the property-list operations.

The details of creation and management of LISP objects is the province of the language and not the concern of the program designer. The creation of objects is totally dynamic. One does not have to declare space allocations for strings, records, or arrays before beginning to program. Storage management is handled by the system using a "garbage collector" and is totally transparent to the user.

* LISP is interactive. There is an evaluator which will execute expressions and produce the result without complex conventions and declarations. This calculator-like behavior allows one to design, program, and debug in an incremental fashion. Small subcomponents can be designed and tested, then set aside, later to be composed with other small pieces to make a larger component. One does not write large monolithic LISP programs very often.

* LISP is a debugging language. A major problem in designing a complex system is the debugging and modification of ideas. One does not begin such a project with a precisely sepcified algorithm. One begins with ideas, and uses the machine to test those ideas. Therefore, a major mode of operation is "modification and testing". Modification in LISP is easy. The whole of LISP's environment is open to change. We will say more about this below under "extensibility". Testing in LISP is also simplified. LISP is a machine language, and as such, the debugging devices present and receive their information in LISP. One debugs LISP programs in LISP. There are built-in functions to handle errors, suspending the computation and allowing the user

to examine or modify the suspended state. These functions, of course, can be replaced by the user, and much more complex monitoring programs can be built --all in LISP.

* LISP is a tool box. There are built-in "tools" --parsers, scanners, output formatters, and table maintenance programs-- which relieve the designer of many lower level implementation details.

* LISP is extensible. The implementation is open to modification. Few decisions in the implementation are irreversible. One can change the LISP library, the evaluator, the parser, and the scanner to the extent of even defining a new language.

This last point, extensibility, is worth expanding upon. Every function name in the LISP environment has a piece of program associated with it. That association can be broken, either temporarily using a lambda binding, or permanently using an assignment. This will allow us to redefine the LISP library. Extensibility requires more: we must be able to define new control structures. This means we must be able to modify the evaluation process. This can be done in LISP in at least two ways. We can install a new version of the LISP evaluator. This is simple because the evaluator is expressible in LISP. An alternative is to introduce new control operations by adding a new special form and carrying out the evaluation ourselves.

These techniques allow modification of the semantics of the language. What about syntax? Suppose we wish to define an Algol-like language --a language with substantially different syntax. Here we need do more than just replace the parser. We need to modify LISP's conception of what is a well-formed expression. Most LISP input systems (including TLC-LISP) are implemented in a table-driven fashion. By this we mean that all of the information about what is a legal construct is stored in a table, rather than being "hard-wired" into an algorithm. To change the the language one changes the table. For example, in TLC-LISP each character has an associated attribute, describing how it can participate in the input: it's a digit, it's a letter, it's a delimiter, it's a comment character, etc. That table is user-modifiable. To design a new input syntax one changes that table and supplies a new routine to collect the input tokens. The new routine will build a LISP-representation of the input. That representation can be executed by LISP's evaluator and the results can be displayed.

A production-quality version of LISP is a fluid collection of tools which can be used to build as varied a collection of applications as any other language. Therefore arguments that LISP is "special purpose" do not hold. Arguments that LISP need be inefficient are also fallacious. It has been demonstrated that one may construct a LISP compiler which is as efficient as a FORTRAN compiler when dealing in the numerical domain. Clearly FORTRAN cannot begin to compete with LISP in the non-numerical domain.

It's interesting to note that the arguments against LISP have now been shifted over to the Logic Programming Languages--they're inefficient, ... The more things change, the more they remain the same.

The power of LISP is truly astounding. There is not one single feature which is the source of this power. It is a blend of several aspects. In combination, these ingredients give a most powerful, but controllable programming language.

Bibliography

Abelson, H, & Sussman, G, _The Structure and Interpretation of Computer Programs_, McGraw-Hill Book Co, New York, 1984.

Allen, J. _Anatomy of LISP_, McGraw-Hill Book Co., New York, 1978.

Allen, J. Don't Overlook LISP, Guest Editorial, BYTE, March 1979, p.6 ff.

Allen. J., Davis, R., Johnson, J, _Thinking About TLC-Logo_, Holt Rinehart Winston, New York, 1983.

Aiello, L. et. al., Adding Classes to LISP, Instituto di Elaborazione Della Informazione, B76-13, Pisa, 1976.

BYTE Magazine, Special Issue on LISP, August 1979.

BYTE Magazine, Special Issue on Smalltalk, August 1981.

BYTE Magazine, Special Issue on Logo, August 1982.

Charniak, E., Riesbeck, C., & McDermott, D., _Artificial Intelligence Programming_, Lawrence Erlbaum Associates, Publishers, Hillsdale, New Jersey, 1979.

Kowlaski, R., Algorithm=Logic+Control, Communications of the ACM, Vol 22, No 7, pp. 424-436.

Knuth, D., _The Art of Computer Programming_, Vol. 1, Addison Wesley, 1968.

Pirsig, R., _Zen and the Art of Motorcycle Maintenance_, Bantam Books, New York, 1974.

Sandewall, E., Programming in an Interactive Environment: The LISP Experience, Computing Surveys, Vol 10, No.1, March 1978, pp 33-71.

Steele, G, and Sussman G., The Art of the Interpreter, or, the Modularity Complex, MIT AI Memo No.453, Cambridge, May 1978.

Teitelman, W., A Display-Oriented Programmer's Assistant, Xerox Palo Alto Research Center, CSL-77-3, 1977.

Winograd, T., Beyond Programming Languages, Communications of the ACM, Vol 22, No 7, pp391-401.

1980 LISP Conference Proceedings
 Particularly:
  Wand, M. Continuation-based Multi-processing pp. 19-28

T L C - L I S P   D O C U M E N T A T I O N

P A R T   I I

Getting Started with TLC-LISP

How To Get Started with TLC-LISP

In the following section we include a sample session with the system that will give you an indication of how we use the system ourselves.

First, to the preliminaries. Before doing anything, make copies of the disk and store the original disk in a safe place. Once that's done, you might want to examine the directory of the disk:

The executable TLC-LISP86 interpreter is named LISP.EXE or LISP.CMD, depending on your operating system's expectations. The executable forms of the LISP utilities (editor, system extensions, file utilities, ...) have an extension of "P", meaning that they contain P-code, the byte-level instructions of a stack-like LISP pseudo machine. These files are not printable, being a self-contained compact form of binary op-codes and symbol tables. Most of these files are loaded automatically at system initialization time. The file LISP.SYS controls the initialization.

The file LISP.SYS is printable, and the curious user may wish to look at it. You will see a sequence of commands to load the P-files and initialize various system variables. The system always performs a (load "LISP.SYS") when it begins. So as you develop your applications you might want to add specific initialization rituals to this file.

The files of the form ??HELP.LSP are used by the editor when help is requested. They are loaded automatically during a help dialog.

The file TOURETZKY.LSP is a package of functions to help make TLC-LISP act like the LISP practiced in "LISP: A Gentle Introduction to Symbolic Computation", written by David S. Touretzky, and published by Harper&Row. The Section titled "Tutorial" that appears later in this Part (Part II) utilizes this book.

```
****  Again, we strongly suggest that you take no actions   ****
****        until  you copy the TLC-supplied disks.         ****
```

How To Load TLC-LISP86

Once the copies have been made, place a copy of your disk in the currently selected drive, and type

```
lisp <return>
```

where <return> means the return/enter key on your keyboard. In the future we will write RET to indicate that key. Meanwhile, back to the loading of LISP.

The operating system will load the executable LISP file from the disk and pass control to the LISP interpreter. The interpreter will allocate memory for the various LISP spaces.

There are two major spaces: Lisp space and Byte space. Lisp space contains the true LISP objects (lists, symbols, numbers, and descriptions of strings, vectors, and P-code, for example), while Byte space contains portions of the internal representations of these objects (internal structures of strings, vectors, and P-code, for example)

You will see an indication of LISP's allocation flash by on the screen. For example, on one particular day we saw:

```
     Lisp space at segment(s) 1D07, 29D3, 369F length CC00
     Byte space at 436B:07CB length F820
     Byte space at 536B:04B1 length 94E0
     Byte space at 087C:0070 length 0CC0
     Byte space at 130C:0113 length 2120

     TLC-LISP V1.46
     Copyright (c) 1982, 1983, 1984 The LISP Company
```

Of course, that actual numbers will vary from machine to machine.

This information may be useful in advanced applications. The section, "Command Line Options" at the end of this Part (II) of the TLC-LISP documentation describes how to modify the default settings, either to redistribute the LISP spaces or to reserve portions of memory for other tasks.

After the initial allocation is completed, TLC-LISP loads several support files. This system load is controlled by the file LISP.SYS. That file contains directives to load the actual code. Later you may find it useful to modify or replace LISP.SYS with you own initialization file.

Once the loading is completed, you will see the prompt:

```
>>>
```
and now TLC-LISP is ready to receive input.

Examples of TLC-LISP

The information in the remainder this section can be skimmed by the knowledgeable LISP afficionados to familarize themselves with TLC-LISP, or these individuals may prefer to skip to the next section on the TLC-LISP editor.

Those new to LISP can use the examples in the next paragraphs in conjunction with the catalog of functions in Part III and then perhaps use the system with Tourtezky's "Gentle Introduction" in hand and our Touretzky tutorial tto develop a more through understanding of LISP.

Like learning to drive, the best way to learn a programming language is to do it -- experiment. Of course competency in driving does not qualify one for either the design or the repair of an automobile. Similarly, the most sterling of programming skills need not imply competency in computer science.

LISP is a calculator. The default listen loop will invoke the evaluator on each well-formed expression it is supplied (of course this behavior may be changed by the user (see TOPLEV). If the expression is atomic, then its value is returned. If the expression is a function call (an application), then the activation will recursively call the evaluator on the components. This description is getting pretty abstract, so let's bring it back to reality with a few simple examples.

You may now use the system as a LISP calculator, typing expressions, striking the return key, and receiving answers.

So for example:

```
>>> t           ; remember to press "return" or "enter"
t
>>> nil         ; ditto
nil


>>> (+ 2 3)   RET          ; recall RET indicates return/enter
5

>>> (cons 1 2) RET
(1 . 2)

>>> (reverse '(a b c)) RET
(b c a)
```

Since every request to the system must be followed by a return-key, we will refrain from RET decoration unless an explicit reference would add clarification.

Continuing with the calculator mode, we can get more ambitious and define new functions. For example:

```
>>> (de add5 (x) (add x 5))
add5

>>> (add5 17)
22
>>> (de add6 (y) (add 6 y))
add6
>>> (add6 (add5 4))
15
>>>      ...
```

Besides defining new functions, we also have a traditional assignment operator. This operation is called SETQ. For example, assume we wish to assign the value of the sum of 5 and 6 to X, and then perform (CONS X NIL), assigning that value to Y:

```
>>> (cons nil t)
(nil . t)
>>> (setq x (add 5 6))
11                         ; the value of the assignment is 11
>>> x                      ; let's check it
11
>>> (setq y (cons x nil))
(11)                       ; note this is (11 . nil)
>>> (CAR Y)                ; note "case" is ignored
11
>>> (cdr y)
nil
```

Let's move to some more complex examples.

We will illustrate several styles of function definition using the factorial function. This is usually written "n!" and is defined as follows

```
n! = 1            if n=0
n! = n*(n-1)!     if n greater than 0
```

The first LISP version of factorial is:

```
>>> (de FACT1 (n)
      (cond
        ( (zerop n) 1)
        ( t (mul n (fact1 (sub1 n))))))
fact1
```

FACT1 corresponds closely with the mathematical description of "n!". We first test if N is zero; if so, we exit with value 1.

Otherwise we perform the multiplication using the value of N and the result of computing FACT1 with the value of N minus one.

We use COND to represent the two cases. The first list within COND covers the situation when n is zero. The second list within COND covers the "otherwise case" --when n is non-zero. The use of "t" as the first member of that list represents the truth-value, true.

Just to assure yourself that fact1 is accurate (at least for one value), we can try:

```
>>> (fact1 3)
6
```

Since factorial is really a case analysis, we can also write it that way:

```
>>> (de FACT2 (n)
      (selectq n
        (0 1)
        (otherwise (mul n (fact2 (sub1 n))))))
fact2
```

and

```
>>> (fact2 3)
6
```

One might consider FACT2 somewhat closer to the mathematical ideal since it is a simple "case"-expression, comparing the value of N against 0 or OTHERWISE, where OTHERWISE is guaranteed to match. Both FACT1 and FACT2 are straightforward recursive computations, based on the complexity of the argument, N.

Now here's a more complex factorial definition:

```
>>> (de FACT3 (n)
      (fact3* n 1))
fact3
```

where:

```
>>> (de FACT3* (n m)
      (if (zerop n)
          m
          (fact3* (sub1 n) (mul n m)))))
fact3*
```

but still:

```
>>> (fact3  3)
6
```

Definition FACT3 is a bit more involved, relying of an "auxiliary" function FACT3* to carry the burden of the computation. FACT3 is used only to initialize the variables which FACT3* needs. FACT3* operates by counting the first argument down to zero as it builds up the factorial value in its second argument. Though FACT3* is recursive, calling itself if N is non-zero, it has a somewhat different behavior than that of FACT1 or FACT2. In particular, when FACT3* has counted N down to zero, it is all ready to return the desired value, M. However when either FACT1 or FACT2 have counted their argument down, there is still a nest of (MUL N (MUL (SUB1 N) ...1) to be computed before the value of the factorial is available. Somehow FACT3 is more "iterative" than "recursive"; this idea can be made precise if necessary. For our purposes, however, we simply note the difference is recursive style; for some problems the FACT3-style is more natural; for some the FACT1-FACT2-style is most applicable.

The aggravating feature of FACT3-FACT3* is not recursion-versus-iteration, but that FACT3* is really a sub-function of FACT, existing only to serve the needs of FACT3. So why clutter up our name space with these unwanted symbols? Well we can solve the problem a couple of ways. First, we may use an extended parameter description syntax to dispense with the subsidiary function altogether:

```
>>> (de FACT3! (n &OPT (m 1))
      (if (zerop n)
          m
          (fact3! (sub1 n) (mul n m)) )
    )
fact3!
>>> (fact3! 3)
6
```

where FACT3! will supply a value of 1 for M when FACT3! is called initially. In this case we use the "optional parameter" m, where the syntax &OPT (m 1) means that FACT3 may be called with either one or two parameters and, if only one paramter is supplied, then the second paramter defaults to 1. Optional paramters are an elegant feature of modern LISP systems.

We cannot always expect such auxiliary functions to be so docile when we attempt to obliterate them. But we can limit the scope of their names by using yet another parameter mechanism called "auxiliary" (or local) variables. Thus:

```
     >>> (de FACT3 (n &AUX (fact3*
                             (lambda (n m)
                               (if (zerop n)
                                       m
                                   (fact3* (subl n) (mul n m))))))


                      (fact3* n 1))
     fact3
```

where in this case the name FACT3* is local to FACT3, meaning the name FACT3* only has that function definition during the execution of FACT3. But let's pull back from the brink of obfuscation to a more common style of programming -- iteration.

One could also recognize an iterative representation for factorial and write:

```
     >>> (de FACT4 (n)
           (do ( (m 1 (mul n m))
                 (n n (subl n)) )
               ( ((zerop n) m)) ))
     fact4
```

and still:

```
     >>> (fact4 3)
     6
```

Definition FACT4 exploits the iterative DO-expression. The first list argument in the DO is a description of how to maintain the local variables N and M. The notation means "initialize M to 1 and on every iteration of the loop set M to the product of the current value of M and the current value of N." Similarly for N, we initialize a new variable N to the value associated with the original N and, on every iteration of the loop, decrement N's value.

There are several important facts to note about these DO-variables. First, these names M and N are introduced as lambda-bindings, receiving the values 1 and the external value of N. Second, in the iterate phase M and N are used as traditional variables for assignments; one simply replaces the old values with those computed by the iterator expressions. Third, these iterator assignments must be done simultaneously. If, for example we reversed the order, performing N's computation before M's, we would not get the appropriate factorial computation.

Rather than insisting that an order be imposed, DO is defined such that parallel assignments are the rule. Similarly the DO is defined so that the initializations are also done in parallel; it makes no difference in FACT4, but may in general.

To continue our discussion of FACT4, we pass to the next list in the DO. This list contains the "exit clauses". In this case there is only one: "if N is zero, exit the DO with the value of M." In the general case there can be several exit tests and several computations to perform if a test is satisfied. If none of the tests are satisfied, the "body" of the DO is executed. In this case the body is empty, so we pass immediately to iterate M and N. In its general formulation, the DO is a most expressive programming construct.

TLC-LISP also offers more restrictive forms of iteration. In particular the factorial function surrenders its secrets to FOR rather nicely:

```
>>> (de FACT5 (n &AUX (m 1))
        (for I (1 n) (setq m (* m i)))
        m))
fact5
>>>
```

and we'll leave it for you to discover the value of (fact5 3).

Several of these factorial definitiona are sufficiently complex that typing errors (particularly parentheses errors)) will arise, and one soon tires of this calculator mode, desiring to change or save function definitions. This brings us to the editor.

Using The TLC-LISP86 Editor

The editor is configured in the style of WordStar. The basic text editing operations have been augmented by a suite of LISP-oriented commands.See page 126 and 127, Part III, of the TLC-LISP documentation for a summary of commands. You might wish to glance at that material now.

To begin the session, let's assume we want to save the two definitions, add5 and add6 on a disk file. First, we load the editor, initializing it with the definition of add5 by:

```
>>> (edit add5)
```

If the editor is not in memory, the system will display

```
Loading  Editor ...
```

while it loads the program.

Since we've called the editor with a symbol name, the editor will also invoke the pretty-printing program to present the definition of add5 in a fashion that illuminates the structure of the definition. When all this is finished, you will see the screen set up with a banner on the first line.

Since we also want add6's definition in the editor, we must load it using ^KA. The notation ^K indicates that the K-key should be struck while the control-key is depressed. In response to ^KA, the editor will request a symbol name. Type

```
add6 RET
```

and you will see the definition of ADD6 appear on the screen.

If there are other functions that you want to place on the initial file, continue to use ^KA until they are all in the edit buffer. Now we're ready to save the buffer.

Notice first that the banner line says "editing NO file" because we entered the editor by loading it with a symbol's value, not a file.

To save the buffer as a file, the simplest option is to type ^KX. This editor command pair will update the currently selected file and exit back to LISP. Since there was no currently selected file, the system will prompt for a file name.

Type a file name (or file name and extension) For example, try:

```
example RET
```

The banner will be updated, and a copy of the editor buffer will be copied to the file EXAMPLE.LSP.

A name without an extension is suffixed with LSP, but if a file extension is given it would be used. A file without an extension would be indicated by a trailing dot (.).

If the file already existed, the old file contents would be saved with a BAK extension.

After writing the file, the editor will display:

```
tlc-lisp
>>>
```

and you're back in LISP.

The definitions are still active in TLC-LISP's memory, so:

```
>>> (add5 7)
12
```

If we decide to edit the functions, or add new functions to this session, it now makes more sense to go back to the editor. Thus:

```
>>> (edit "example")
```

where the string reference indicates that a file is desired. If there is no such file, the editor would create a new one. However in this case, EXAMPLE.LSP exists and the editor will reactivate using the saved file.

At this time we can add more definitions or modify existing ones. The questions then are how to get the new information saved on the disk and how to get the new information installed in the LISP memory.

There are two options to save the file.

* We can overwrite the previous version by typing ^KS

* We can also save the new version under a different name. In this case we must use a sequence of commands. First we must mark the whole buffer.

1. Go to the beginning of the buffer with ^QR
2. Place the beginning mark with ^KB
3. Now go to the end of the buffer with ^QC, and
4. Mark the end of the buffer with ^KK

Now type ^KW. As ^KW is executed you will see the banner prompt for a file name. Type one of your choice. The edit buffer will now be written out into the new file, and you will be left in the original file name.

To summarize, a typical way to use the system involves building new functions in the editor buffer. We can do this by calling EDIT with an unused file name and then begin writing functions. Once we have a sufficiently interesting collection in the buffer, we use ^KS to save our work and then read the text into the TLC-LISP workspace. That's the next topic.

How To Get The Text Into The TLC-LISP Workspace

Given that we can save text on files, we must also be able to read that text into LISP's workspace. Several options are available here too.

* We may read the whole edit buffer into memory. This is accomplished by ^KJ. As this command is executed, you will see the results displayed in the banner. If an error is discovered during this operation, the system will "beep" and the cursor will be positioned at the location of the error. After identifying and correcting the error, you may wish to execute another ^KJ, or you might wish to execute a sequence of our next operation, ^JJ.

* We may read in an expression-at-a-time by ^JJ, ^JJ will evaluate the expression following the cursor, and move the cursor past the end of that expression. In particular, if the cursor is at the beginning of an expression like:

```
(de foo (x) ...

     ...)
```

then ^JJ will install foo's definition and move the cursor past the right parenthesis that completes foo's definition. This means that a sequence of ^JJ's will incrementally execute the contents of the edit buffer.

Some Tricks For Finding Your Way Around In The Editor.

* The ^JP command will find a matching right-parenthesis. Put the cursor ata left paren and type ^JP.

* The right-parenthesis fence trick. One common syntax error is unbalanced parentheses. We tend to place a collection of right parentheses at the end of each definition. This we we're guantanteed at least that the definition will parse to something--it may not be correct, but at least the parser will stop at the fence and not continue gobbling text.

* Once the text buffer has been read successfully, then ^JP can be used to analyze the finer parenthesis structure of definitions that don't seem to be acting as expected.

What To Do Now That The Definitions Are Installed.

Now that the definitions have been turned into TLC-LISP code, we'd like to exercise them. We could do this by writing expression-like text in the buffer and then use ^JJ. The results of the evaluation will appear in the banner line. This strategy is only useful for small test cases. More usually, we will leave the editor, passing control to the interpreter. As with other editor operations, we have several options here too. The most usually scheme is to press the "escape" key (in the future, we will write "type ESC" to mean "press the escape key"). Type ESC to suspend the editor (rather than exit from it) and return to the interpreter's screen format.

Whenever you wish to return to the editor press the "escape" key--this time followed by a carriage-return, and you'll return to the editor (in the furute, we will refer to the "escape, carriage-return" sequence as simply ESC-RET). Try this entry-exit protocol a few times, ending up in the editor.

The ESC/ESC-RET toggle is an effective method for program development. It can be used with ^KS and ^KW to keep the external files consistent with the internal working code.

It is also possible to terminate (rather than suspend) a session with the editor. This can be done by a series of ^K-options:

^KX leaves editor after writing the latest version of the file.

^KQ leaves the editor, after giving the user a chance to update a file if the buffer has been modified.

An Advanced Feature--The EDIT Stack.

The editor may be called recursively, allowing you to suspend one edit while editing another file. If you suspend an editing session with the ESC option, then a subsequent call on EDIT will invoke a new copy of the editor and it in turn can be suspended. The result is a stack of suspended edits. To pop elements off the stack, exit with ^KX or ^KQX and the previous edit will be uncovered.

Some Examples Of The Editor In Action.

Below is a reproduction of a file START.LSP. It contains several errors--some syntactic, some semantic.

```
; The file START.LSP

(de foo (x)
   (if (zerpo x) 1 (* x (foo subl x)))))))


(de bar (x y) (cons x y)


(de baz (x) (car x ))))))))))


;   end of the file
```

Assume we read this file into the TLC-LISP system by:

```
(edit "start")
```

Note that we'll get start.lsp because of the implied .lsp file extension.

Once in the editor we can execute the file with ^KJ.

To check that the definitions are installed, type ESC to suspend the editor and return to the TLC-LISP interpreter.

Typing foo (followed by a return) displays the definition of foo. But typing bar gets us something we may not have expected. We see that the definition of baz got included within bar. A moment's reflection reveals the problem--an insufficient collection of right parentheses at the end of bar's definition.

Type ESC-RET to return to the editor. Position the cursor at the end of bar's definition and add some right parentheses. In this case, we can see that one will suffice, but the parenthesis fence trick is a good habit to develop. Now go to the beginning of the bar definition and type two ^JJ sequences. The first one will install bar; the second ^JJ will install baz.

Just to be safe, we can do a ^KS to save the file, and then type ESC to return to the interpreter.

Now, at least the low-level syntax problems have been licked. Let's try to run some of these examples.

Debugging: Finding and Fixing Semantic Errors.

If we tried

(baz '(1 2))   or

(bar 2 3)

we'd get the results we should. However

(foo 0) gives an error:

```
unbound-symbol zerpo
```

The problem, of course, is a misspelling--we meant to write zerop. The simplest solution in this case is to return to the editor and change the text (in situations in which a substantial computational investment has been made, such a restart would not be appropriate).

To reenter the editor type ESC-RET and move to the offending zerpo symbol and correct it. If this were a large file, we could use ^QF and ^L to find the symbol.

Now go to the beginning of the definition and execute a ^JJ sequence, and then ESC to the interpreter. We'll now find that:

```
>>> (foo 0)
1
```

For a more comprehensive test we'll try

(foo 5)

Alas, this supplies us with another error message:

```
too-many-arguments
```

To discover the offender, we use the back-trace function, bt.

```
>>> (bt)
```

You will see a sequence of frames on your screen. These frames contain indications of the dynamic state of the machine at the time of the error. The first frame indicates the latest state--the error frame, and the next one (frame 2) indicates the context in which the error occurred.

That frame looks like:

```
+--- FRAME 2
Fcn: foo
Arg: (subl x)
```

The Arg-slot indicates the expressions that were to be evaluated as arguments to the function described in the Fcn-slot. Since Arg is a list of two elements, it says we passed to arguments to foo. Therein lies our problem: foo expects a single argument, but we've given it two arguments--subl and x. The problem is a set of missing parentheses, so we return to the editor using ESC-RET.

So we edit foo, replacing

```
(foo subl x)   with

(foo (subl x))
```

Now move to the beginning of the definition; do a ^JJ; then a ^KS; then ESC. Finally we can try

```
>>> (foo 5)
 120
```

This is a simple introduction to TLC-LISP, to the editor, and to debugging. But it is a productive path through the complexity. As you gain confidence with your command of the language or as you gain frustration with the limits of these techniques, examine the documentation for other options.

How to Use TLC-LISP86 with "LISP: A Gentle Introduction to Symbolic Computation" by David S. Touretzky

As with any language, one's competence and skill increases with exposure and practice. The TLC system comes with several examples of LISP source code that show how to use the language in practice. These examples should prove valuable later. But first we need a bridge between them and the one-line code of the previous sections. That bridge must serve two purposes. First, it should expose the beginner to the mechanics of the language--its primitive words and the techniques for combining those words to make meaningful phrases. Equally important, the bridge must introduce the components of the language in a way that develops a sense of style and elegance in the new user.

It is particularly important for a LISP novice to understand issues of style--LISP is a sharp tool, a tool that contains few built-in stylistic restrictions. As such, LISP is succeptible to misuse. Furthermore LISP is a twenty-year old tool. As such, it still contains some artifacts of ancient coding style (progs and gos). LISP shares these horrific features with its general purpose bretheren of that age.

LISP also still contains some rather muddled notions in its own right. It was the first practical functional programming language and though it tried mightily, it contains several confused notions. In particular traditional LISPs (including Common LISP) have confused the notions of name versus object. These confusions in LISP make it difficult to build a clean, modern treatment of functional programming techniques in a traditional LISP. We believe that TLC-LISP86 offers a reasonable compromize between historical LISP (LISP1.5) and future LISP-like languages (Scheme). We believe that Touretzky's book offers a good vehicle for understanding LISP in general and, provided that a few precautions are taken, provides a good tutorial for TLC-LISP. We therefore have developed a package of TLC-LISP code that will emulate the LISP as described in Touretzky's book except in those places where we believe a compromise cannot be made.

In the poem entitled "i sing of olaf", e. e. cummings deals with the question of compromise. Before moving into the tutorial mode, we will outline a few areas we find indigestible and beyond compromise.

Irreconcilable Differences

Though all the differences between TLC-LISP86 and Touretzky's book could be washed away with a compatibility file, we believe that a few differences are so important that they cannot be compromised. Specifically:

* Function name versus function object *

In traditional LISP a functional parameter will be quoted, as in

```
(APPLY 'PLUS '(2 3))   ; Touretzky, page 92.
```

but 'PLUS references a symbol--the name of a function, not a function. Similarly, in:

```
(APPLY '(LAMBDA (X) (CONS X NIL)) '(A))
```

'(LAMBDA (X) (CONS X NIL)) references a list, not a function.

The resultant confusion between name and object has muddled LISP for decades. LISP got away with the confusion because the majority of LISP programming activity has steered away from functional parameters--for good reason. However as an appreciation for the power of functional programming techniques grew, the problems with LISP's confused notion of function became undeniable. In the long run, the rule will be lexically scoped languages with first-class functional objects. In the interim, we build a compromise in TLC-LISP.:

TLC-LISP enforces the distinction between a function and its name. Thus, in TLC-LISP APPLY expects its first argument to be a functional object, not to be something that names a functional object. So in TLC-LISP we write

```
(APPLY PLUS '(2 3)))     rather than   (APPLY 'PLUS '(2 3))
```

Likewise '(LAMBDA (X)(CONS X NIL))) is only a list;

while (LAMBDA (X)(CONS X NIL))) a functional reference.

Viewed in terms of Touretzky's functional "box notation", a function that expects a functional argument would have to have a "box", not a list on one of its inputs.

We believe that this distinciton between object and representation is important. That is why TLC-LISP is built the way it is. That distinction is sufficiently important that we did not include code to make TLC-LISP compatible with Touretzky's LISP regarding "functional" parameters.

*** LISP 1.5 PROGs versus &AUX, DO, REP, FOR, CATCH, and THROW ***

The early LISP implementations introduced iteration in a fashion consistent with those times--1958. The resulting structure, called the PROG-feature, remains even in modern LISPs though many cogent arguments have been given for its demise. Well, TLC-LISP killed the PROG and its assorted paraphernalia many years ago and is not about ready to resurrect it.

TLC-LISP offers several options in place of the PROG-feature:

-- For local names (PROG variables), we supply &AUX variables.

```
So replace (DE FOO (U)
              (PROG (X Y Z)
                   (SETQ X x-init) (SETQ Y y-init)
                    .  .  .   )

with  (DE FOO (U &AUX (X x-init) (Y y-init) Z) .  .  .)
```

-- For controlled iteration use DO, FOR, or REP.

```
Replace  LOOP
          (AND (ZEROP N) (RETURN X))
          (SETQ X (CDR X)) (SETQ N (SUB1 N))
              .  .  .
          (GO LOOP)))))

with   (DO ((X X (CDR X))
            (N N (SUB1 N)))
           (((ZEROP N) X))
              .  .  .  )
```

-- For a more simple iteration, FOR is appropriate:

```
(FOR (I 1 20) (PRINT I))
```

-- For the simplest of loops, REP may suffice:

```
(DE DELAY (N) (REP N ()))
```

-- For non-structured control operations, use CATCH, THROW, and UNWIND-PROTECT. These are sufficiently advanced that we'll pass on their discussion for now.

With these few caveats in mind, we feel comfortable in offering Touretzky's book as a "Gentle Introduction to Symbolic Computation."

You may find it rewarding to examine TOURETZKY.LSP. Type

(edit "touretzky")   and browse around.

How to Use the Tutorial

First, don't turn on the machine. Read the first two chapters of "Gentle" just to get a feel for the ideas. Then turn on the machine, load TLC-LISP and after the initialization ritual is finished, type:

```
(load "touretzky") RTN
```

When the prompt appears, type

```
(enter-dt) RTN
```

and you will be transported to the Touretzky package named dt. You may verify this by examining the value of the symbol named package:

```
>>> package RTN

dt:
```

When you wish to leave the dt-package, type

```
(leave-dt) RTN
```

Now, however, we want to explore the book. At this point you may begin working through Chapter three of the Touretzky book. We suspect that you'll find it worthwhile to bring the editor into the loop early.

For example, simple things like

```
(PLUS 2 2)
```

can be done within the editor by:

* Load the editor
* Type the expression
* Move the cursor in from of it.
* Type ^JJ.
* See the result in the banner window.

As computations get more complex, you'll soon find need for the editor. Perhaps try:

```
(edit "session")
```

to load up the editor with a new file, and then use the ^KS-^KJ-ESC/ESC-RTN sequence to move between the editor file and the interpreter.

Remarks by Section

This section is a running commentary to be read in conjunction with "A Gentle Introduction". It contains comments that came to mind in reading the book and comparing it with TLC's ideas.

Chapter 1.

The "box notation" for functions is a good idea to cultivate, both intellectually and in terms of programming style. Though LISP will support the arbitrary use of global variables and ill-structured code, such behavior is not recommended. So if you think about programs as implementations of such boxes, you'll develop a functional style of LISP programming.

Chapter 2: Lists

Section 5. (page 38) The introduction of list implementation seems premature. Addresses, numbers, and pointers are irrelevant to a discussion of lists and list operations. Think of lists as abstract objects, not addresses in a machine.

Chapter 3: Eval Notation

Introduction. The comment "In Lisp, functions are data" is not quite accurate in Touretzky's LISP (and many others). These LISP's confuse the notion of function with their representation. See the section "Irreconcilable Differences", particularly the comments about functional "box notation". You might consider how to extend his box notion to support REAL functional objects.

Section 5. (page 70) The "quoting" issue is another instance of object-versus-reference or use-versus-mention. You might think about a LISP-like language (whatever that means) that has only numbers and strings for data objects. Quoting would not come up here, because we would not be able to refer to components of the language (symbols or identifiers) as data objects. This is typically the case in other general purpose languages, and such languages do not need quotes. Furthermore, if we took that same LISP-like language and only added functional arguments and values (REAL functional objects, that is) we would still not need quoting. We'd be able to refer to functional objects by their names as we do in TLC-LISP:

```
  (foo 3 bar)
```

For example, the occurrence of foo means function application, and the occurrence of bar indicates a functional parameter.

Quoting only becomes an issue when we allow components of the language to become objects of the language; that is, when we can talk about the language within that same language. That's a separate issue from that of functional objects.

Section 8. (page 74) Touretzky follows traditional MacLISP in using DEFUN as the single mechanism for function definitions. In the early chapters, DEFUN is used in the form

```
(DEFUN <name> <formal parameters> <body>)
```

which is equivalent to TLC-LISP's

```
(DE <name> <formal parameters> <body>)
```

We have included this version of DEFUN in the compatibility package.

However DEFUN a'la Touretzky can also be used to define other function-types, specifically special forms as described on page 260 of "Gentle". There he writes:

```
(DEFUN GARBLE FEXPR (X)
  (CONS 'SAY (REVERSE X)))
```

The compatibility package does not support this extended DEFUN, believing that overloading the operation is not productive. Rather, we support DF for special forms, DM for macros, and DMC for character (read) macros. For example,

```
(DF GARBLE (X)
   (CONS 'SAY (REVERSE X))))
```

A major point to notice about special forms like DEFUN is that they do not follow the "function-followed-by-arguments-to-be-evaluated" ritual. When we write

```
(DEFUN SUB2 (X) (SUB1 (SUB1 X)))
```

we do not expect each component of the expression to be evaluates as we would if presented with following list of similar syntactic appearance:

```
(LIST  SUB1   (LIST) (SUB1 (SUB1 X)))
```

Namely, LIST will evaluate all three of its arguments, finding a primitive routine as the value of SUB1, the empty list for the val;ue of (LIST), and will complain if it cannot find a value for X.

However, DEFUN will not evaluate any of its arguments, but rather act like an assignment, associating its second and third arguments with the symbol that it finds in its first argument slot.

Operators like DEFUN are callled Special Forms because the handle their arguments in special idiosyncratic fashions. Other special forms we'll see are IF, COND, FOR, and CATCH, for example.

Section 10. (page 77) The question of variable bindings is a very important one--one which LISP has tended to muddle. Specifically, consider the following example from "Gentle":

```
(DEFUN SQUARE (N) (TIMES N N))
```

Traditional LISP treats SQUARE and TIMES differently from N, not just that the former are functional references and the latter are numeric (in this case); but it is possible for the three symbols to have both functional and simple values simultaneously. This leads to the following kinds of scummy code:

```
(DEFUN SQUARE (TIMES) (TIMES TIMES TIMES))
```

and the LISP interpreter is required to discover which value of TIMES is meant to be used in each context. Unfortunately, such LISP interpreteres are not always able to guess the correct binding.

In contrast, we believe that SQUARE, TIMES, and N are all of the same general category--symbols that play the role of variables. TIMES and SQUARE are names of functions, while N names a numeric value. We believe that such role-playing symbols can carry but a single value at any one time; sometimes the value can be functional (TIMES and SQUARE) and sometimes the value can be a simple object (N). But at most one such value, please. That seems self-evident from a mathematical perspective, and is the way TLC-LISP handles variables and their bindings.

We believe that the "double value" hack is without integrity; it only maintains its usefulness because of LISP's deeper problem with dynamic scoping. For consider the last SQUARE example. If TIMES can only take on a single value, all three occurrences of TIMES in expression

```
(TIMES TIMES TIMES)
```

reference the same value. Such pathologies can be spotted easily, but they are indicative of a deeper problem. Namely that in a dynamically scoped LISP, any formal parameter has the potential of hiding (or shadowing) any name that will be accessed within the dynamic scope of that formal parameter.

So if we write

```
(DE FOO (N X LIST)   <function body>)
```

then any attempt within the execution of <function body> to reference the system's LIST function will fail; such a reference will get the value of FOO's third parameter. (In fact, we can get around this problem by using the package system, placing FOO's LIST symbol in a different package, and referencing the system's LIST as :LIST -- ugh).

While the dual function-value hack will stop many of these problems, it doesn't solve the problem. The solution is lexical scoping. Modern LISP's are moving toward that position. In the interim, TLC-LISP would rather anticipate the future than perpetuate the past.

Meet the Computer (page 80). Experimentation and observation is appropriate here. There are some differences between TLC and MacLisp error messages. Our prompt indicator is >>> rather than asterisk (*), and our "grinder" is called PP, not GRINDEF. Rather than sugar-coat the differences, we have left them. You should learn to accommodate slight differences in systems.

Advanced Topics 3

1. A Note on Lambda Notation. (page 86) Touretzky remarks that "LAMBDA is not a function, it is a marker treated specially by EVAL." Ugly. In TLC-LISP, LAMBDA is not treated specially by EVAL, rather it is a Special Form that manufactures a function-type object from the components found in the body of the LAMBDA.

2. Functions of No Arguments. (page 87) A simple comment: typing

```
TEST
```

will retrieve the function definition in TLC-LISP.

3. Dynamic Scoping. (page 87). This section indicates the roots of many of LISP's problems. The dual function/value hack allowed LISP to prosper over its first twenty years of activity because people tended to write rather traditional code, staying away from the problematic areas of scoping--namely functional objects. As the power of functional programming became more widely appreciated, the weaknesses in LISP's implementation of scoping became more obvious.

5. EVAL and APPLY (page 92). With the appearance of APPLY, the distinction between the name of a function (Touretzky) and a function (TLC) is made explicit. Where Tourtezky writes:

```
(APPLY 'PLYS '(2 3))
```

we write

```
(APPLY PLUS '(2 3))
```

Chapter 4: Conditionals

General Comment: TLC-LISP also supports a type of CASE-expression, named SELECTQ. See the TLC-LISP documentation.

Chapter 5: Global Variables and Side-Effects

General Comment: Why introduce this here! If you want to be conservative, introduce iteration here; if you want to be liberal, the applicative operators or recursion would be appropriate; but this is down-right reactionary.

Specific comment: Notice the two SETQ's on the bottom of page 120, giving values to FIRST and REST. These assignments could destroy the functionsof the same name in TLC-LISP. If you examine TOURETZKY.LSP you'll se how we solved that problem.

Chapter 6: List Data Structures

Section 4. Lists As Sets. You might want to compare our implementation of the set functions (in touretzky.lsp) with those on page 286 of "Gentle". We have combined a lot of common code to highlight the similarities in the set algorithms.

Advanced Topics 6. (page 155) EQ versus EQUAL. Compare this with the discussion in the TLC-LISP documentation on identity versus indistinguishability, and object versus value (pages 22-24 of Section I). There's a lot more to EQ and EQUAL than pointers and addresses.

Another facet of equality involves a practical issue. When are two numbers equal? An integer 1 and a floating-point 1 are equal in some measure, but what about comparison of two floating point numbers that differ in a barely perceptible way?

Chapter 7: Applicative Operators

This is a very nice touch. It starts to bring the functional power of LISP-like languages into play. We would rather have seen Chapters 7 and 8 (recursion) where Chapter 5 (side effects) was, but ...

Section 2. The APPLY-TO-ALL Operator. (page 165) The version we supply in touretzky.lsp expects a function, not a function name as argument. Thus we write

```
(APPLY-TO-ALL SQUARE '(1 2 3 4 5))
```

where Touretzky writes

```
(APPLY-TO-ALL 'SQUARE '(1 2 3 4 5))
```

We will insist on maintaining this difference throughout:

```
 Section 3 Lambda Expressions
            (lambda (x) (times x x)) is a function
           '(lambda (x) (times x x)) is a list.

 Section 4 The FIND-IF Operator
            oddp is a function
            'oddp is a symbol
```

Similarly for Section 5 and 6.

Section 7. The REDUCE Operator (page 176). As far as we know, REDUCE-like operators were introduced into LISP-like languages with the birth of the LIT operator (standing for LIst Iterator) by Strachey and Barron (1966). See Anatomy of LISP page 196.

The LIT operator took three arguments: the function the argument list and the terminating value. Thus for example APPEND could be defined as:

```
(DE APPEND (X Y) (LIT CONS X Y))    ; note CONS, not 'CONS
```

Now what about REDUCE? It makes more sense to us to have the terminating value manifest in the notation than hidden on the property list. Thus our version of REDUCE expects three arguments. Furthermore the "Gentle" REDUCE contains too many unrelated special cases. As a result, we've supplied LIT-style reduction operators for REDUCE, LEFT-REDUCE and RIGHT-reduce. You will have to modify Tourtezky's examples accordingly.

Chapter 8: Recursion.

Chapter 8? A strange place for a key facet of functional programming. This should have come earlier. Be that as it may, recursion is an elegant way to build complex computations with a few lines.

Tail recursion is subject of importance both for the style of programming that it develops, but from a practical efficiency-driven, perspective. Specifically, the interpreter can take liberties with the (recursive) execution model and execute many instances of tail recursive computations without consuming stack space.

A technical remark: when a tail-recursive instance is recognized, the interpreter can reuse the existing stack frame, replacing the values in the frame and re-executing the code body, effectively translating something like (Touretzky, page 223.):

```
(DEFUN TR-REV1 (X Y)
    (COND ((NULL X) Y)
          (T (TR-REV1 (CDR X) (CONS (CAR X) Y)))))
```

into:

```
(DEFUN TR-REV1 (X Y)
  (PROG NIL
       LABEL (COND ((NULL X) (RETURN Y)))
             (SETQ Y (CONS (CAR X) Y))
             (SETQ X (CDR X))
             (GO LABEL))))))
```

Two points:

* Minor note: the ancient LISP compilers (circa 1965) used to do this kind of transformation before interpreters were constructed to do it (circa 1975, Greussay).

* Major point: if compilers and interpreters can unwind such recursions into efficient iterative code, why persist with the PROG-GO hack?

Coming of Age in LISP

By this time you should be reasonably comfortable with Touretzky's book, with TLC-LISP, and with LISP in general. You should also become reasonably immune to local distinctions and quirks of specific LISP dialects. So now we'll change gears and accentuate the differences rather than the similarities. A good place to start is to load touretsky.lsp into the editor, and just compare it with Appendix C (page 279).

Chapter 9: Elementary Input/Output

Elementary Input and Output is never "elementary". This area is always the most idiosyncratic portion of a language. Rather than attempt to reconcile Touretzky to TLC we'll highlight the advanced features of TLC-LISP input/output.

All of the primitive I/O operations advertized in Touretzky's book are supported in TLC-LISP. The function MSG is also supplied in the compatibility file. The interesting portion of the TLC I/O system involves the "source" and "sink" possibilities for input and output respectively.

All TLC I/O functions will accept an optional final argument that specifies the origin of the input character stream (source) or the target for output characters (sink). At system initialization the I/O objects are set up to interact with the console and keyboard. A LOAD operation will redirect input to specified file, for example. Or we could define a stream to drive a printer by;

```
(setq lpt-stream (stream lpr))     and then
```

(print <expression> lpt-stream) will print the expression on the device.

As this last example indicates, in the most general setting, the sink/source object is a stream that contains a piece of LISP code that specifies what to do with, or where to get, the characters.

File I/O in TLC-LISP is again as robust as that expected by Touretzky (page 241), but again it differs in specifics.

This is all the bad news about I/O. The good news is that for most application you can ignore it. The default reader and printer supplied by the system offers clean communication paths. The saving and restoring of files is well-handled by editor interaction. And the initialization of applications packages is made painless by the varieties of LOAD.

Chapter 10: Iteration.

Section 2, "The PROG Special Form" should not be read. PROG is an artifact of 1960 LISP and, as such, we believe that there is no reason for it to persist. The effect of PROG variables is much better served by &AUX-variables. Gos and their associated labels are better served either by DO or by CATCH-THROW pairs. So go directly to Section 3, "The LET Special Form".

One remark should be made about Section 4, "The DO Special Form". Specifically, the TLC version of this construct is more general than that supported by Zeta LISP (the DO described in "Gentle"). In their DO there is a single end-test condition; in TLC-LISP there are multiple tests. Thus where page 252 says

```
(condition action-1 ... action-n)
```

TLC-LISP expects

```
((condition-1 action1-1 ... action1-n)
     ... )
```

Advanced Topics 10.

Section 3: Defining Macros (page 260). Remember that TLC-LISP uses DM to define a macro.

Section 4: Functions with Arbitrary Numbers of Inputs. (page 261) TLC-LISP uses the &REST-mechanism rather than LEXPRs. So we'd rather write POLY-PLUS as:

```
(de poly-plus (&rest l) (reduce + l 0)))
```

Why be more obscure than necessary?

Chapter 11: Property Lists

Before getting into the pros and cons of property-lists, we make one religious comment about the book's functions:

```
(GET symbol propname)
```

```
(PUTPROP symbol propvalue propname)
```

The irregularity in the first two argument positions of these functiions has always annoyed us.

So in TLC-LISP we write:

```
(GETPROP symbol propname)

(PUTPROP symbol propname propvalue)
```

We certainly believe that property-lists are useful, otherweise we would not have implemented them in TLC-LISP. Historically, p-lists were a breeding ground for data-driven programming techniques--a precursor of classes, instances, and message-passing, made famous by Smalltalk.

One annoying feature of the usual implementation of property-lists is their dependence on a specific symbol name. There is no notion of an anonymous p-list. This is unfortunate since it spoils the notion of p-lists as first-class objects. If property-lists were to hold a more prominent position in future Lisp programming techniques, then we'd consider extending them. However, the notions embodied in the TLC-LISP86 class system appear to offer a superset of the p-list features, and the class concepts are first-class in TLC-LISP86. Thus we leave p-lists in their historical state. For a more detailed discussion of the applications of p-lists, classes, and data-driven programming techniques see the TLC-LISP documentation.

Advanced Topics 11

This section of the book is out of date. MacLisp has not used the property-list technique for functions and values for at least ten years. The preferred technique stores functions and values in special slots within the structure that implements a symbol. In fact, the treatment attributed to Franz LISP (on page 274) was taken from MacLisp. TLC-LISP also follows this tradition, though as we noted before there is a single Value Cell here. Properties other than the function/simple value of a symbol are stored on the p-list and accessed via the PLIST function.

Summary

As you've seen, we have some problems with both the LISP and the programming style advertized in Touretzky's book, but in general it is a good introduction to the topic.

Parser Examples

When learning a new language, it is always useful to examine a reasonably large program written in that language. This is particularly useful when learning a language whose power and scope is as broad as that of LISP.

One complaint about LISP is its syntax; while other languages expend a great deal of effort on complex notation, LISP uses simple variations on the single theme --(<operator> <operand-1> ... <operand-n>). The simplified notation has several benefits, as we have seen. A benefit that we wish to exploit in this section is the simplicity of the parser; the parser is the algorithm to translate the external list notation into the internal tree representation. In a moment we will write a LISP parser in about a half-dozen lines of LISP. Through a series of simple tansformations, we will use the power of LISP and its notational simplicity to write a parser that will camouflage the LISP syntax under an Algol-like notational blanket. The final parser will be user-modifiable and table-driven; it will exploit LISP's property lists to maintain the tables. Those tables will contain both data and parsing programs, exploiting the program/data duality to give us a flexible, compact and understandable parser. It is ironic: to quiet the complaints of the non-LISP community who believe LISP's syntax and the above-mentioned programming features are obscure and difficult, we depend on those very attributes to develop a flexible and highly readable parser for those people. It would be a non-trivial exercise to encode this parsing scheme in another language without sacrificing flexibility or clarity.

By the time we have constructed the last Algol-like parser you may feel that the power of the undecorated LISP is sufficiently seductive that the notational "convenience" which we constucted will go unused.

The example of this section requires some concentration; the problem is non-trivial and LISP may be new to you. However, the major difficulty is unlearning old programming habits and restriction, and learning how to use the power of LISP to describe complex problems which could not be succinctly described and designed with other tools. Let us begin.

We discuss a sequence of parsers, leading from a simple algorithm that mirrors LISP's list-structure reader, to a generalized parser that is capable of supporting an Algol-like pre-processor for LISP. All these algorithms will use TLC-LISP's basic scanner named SCAN. SCAN processes an input stream, looking for basic objects --symbols, numbers, and strings-- and delimiters; it will construct the basic objects, returning their representations as values, and will return a character representation of the delimiter; in TLC-LISP, a character constant is represented as \<char>. The scanner is also able to recognize comment strings and strip them out of the input. All of SCAN's knowledge about what is a symbol, number, string, delimiter, or comment, is stored in a user-modifiable table; see TYPECH for a description of the tabular information. Initially, we will use the default LISP settings; later parsers will modify that table, allowing us to describe a totally new syntax.

Our first parser is a simple version of TLC-LISP's READ; it only recognizes list-notation, not dotted pairs.

```
(de READER (&AUX obj)
  (selectq (setq obj (scan))
    (\( (read-rest))
    (ow obj) ))

(de READ-REST (&AUX obj)
  (selectq (setq obj (scan))
    (\( (concat (read-rest) (read-rest)))
    (\) nil)
    (ow (concat obj (read-rest)))))
```

The actual parser in TLC-LISP is more complex. It performs error checking and in fact is a non-recursive implementation based on an algorithm described in Anatomy of LISP; however, the conceptual essence of a LISP parser is cogently and concisely described in READER and READ-REST.

Clearly, this READER will understand nothing but LISP; our search for generality must begin by removing this unilateral view. The key is to note that READ-REST terminates when it sees a \); that is, READ-REST is a special instance of an algorithm we might call READ-UNTIL, which reads the input stream until is sees a designated character; in the case of READ-REST, the designated character is a right parenthesis. That is:

```
(de READ-REST ()
  (read-until \) ) )
```

Our intention here is to move all of the language-specific information out of the parsing technique, and install that knowledge in tables which a general parser can refer to. We have seen something like this already: read macros are table-driven procedures which are invoked when a special character is seen in the input stream; this is the second notion we need for effective generalization.

The general scheme that we are about to elaborate --Top Down Operator Precedence-- is due to Vaughan Pratt (see the Parser Bibliography at the end of this section). The essential problem in parsing is to discover the structure of the text being input to the system. To discover structure in a string of input means to determine the entities of the language, and to determine the interrelationships between them. A scanner finds the entities; the parser detemines the interrelationships. We were all probably introduced to the formal notion of parsing through the same problem: "how do you group (or parse) x+y*z?" The solution was to associate the "y" with the "z", effectively giving x+(y*z) instead of (x+y)*z. We say that the operator * "takes precedence over", or "binds more tightly than" +. This idea of "operator precedence" was formalized by R. Floyd (see the Bibliography). The Pratt parsers use an extended precedence relation, which associates "left and right binding powers" with operators. For example, given operators O1 and O2, and a segment of text:

```
        ...O1 ... O2 ...
```

if the right binding power of O1 is greater than the left binding power of O2, then the (parsed) text between O1 and O2 is associated with O1.

In the implementation, adapted from one written by Martin Griss, the left- and right- binding power of an operator is stored as a dotted pair of numbers on the property list of the operator under an indicator named INFIX. For example:

```
     (putprop '+ 'infix '(10 . 10))
     (putprop '- 'infix '(10 . 10))
     (putprop '* 'infix '(12 . 12))
     (putprop '= 'infix '(5 . 5))
     (putprop '? 'infix '(-2 . -2))
```

where = will be used for an assignment operator, and ? will be used to indicate the end of an expression.

The parser is given a binding power and an initial token, and parses from left-to-right until it finds an operator with left binding power greater than the given binding power. When it comes upon an operator with a lower left binding power it applies the parse algorithm recursively. For example, the phrase:

```
      z = x + y * z ?     would parse as    (= z (+ x (* y z)))
```

Given this internalized form of the input, we can further translate it into a list which can be evaluated by LISP. The definition of PARSE follows:

```
(de PARSE (rbp exp &AUX (ex2 (getprop obj 'prefix)))
  (if ex2
      (setq exp (list exp (parse ex2 (scanit))))
      ; else
      (scanit) )
  (do ( (ex2 (getprop obj 'infix) (getprop obj 'infix)) )
      ( ((or (null ex2)
             (ge rbp (car ex2)) ) exp) )
      (setq exp (list obj
                      ex2
                      (parse (cdr ex2) (scanit)) ))))

(de SCANIT ()
  (setq obj (scan)) )
```

This is all there is to the parser! The parse behavior is controlled by the information stored on the property list of the operators. Operators have INFIX or PREFIX properties; all other atoms are operands.

The next embellishment would be to allow an operator to control the parse locally. For that, we could store a program on the property list. This program could contain arbitrary computations, including code to parse more of the input stream.

Parser Bibliography

Floyd, R., Syntactic Analysis and Operator Precedence, Journal of the ACM, Vol. 10, pp 316-333, 1963.

Pratt, V., Top Down Operator Precedence, Proceedings of the ACM Symposium on Principles of Programming, pp.41-51, 1973.

Pratt, V., CGOL - An Alternative External Representation for LISP Users, MIT AI Lab, Working Paper No. 89, 1976.

Command Line Options

The following options may appear in the command line that you type to the operating system to invoke LISP.

@<filename> instructs LISP to use the file <filename> instead of the file LISP.SYS for automatic loading after startup. Recall that the default extension is ".LSP" if none is supplied.

P<num> instructs LISP to use <num> percent of allocated memory for Lisp space, the rest for byte space. Lisp space is that area containing CONS-nodes; byte space is the area containing descriptor-based object, like strings, vectors, and P-code. The default is sixty resulting in a 60/40 split.

M<hexadecimal number> sets the maximum number of paragraphs (one paragraph is sixteen bytes) that Lisp will attempt to allocate from the operating system. The default is 0FFFFH (one megabyte) This option is useful for Concurrent CP/M systems to prevent Lisp from grabbing all available memory. It is also useful for MSDOS version 2 to allow use of the EXEC function to invoke a second copy of the MSDOS command interpreter.

The defaults are equivalent to a command line like:

```
    LISP @LISP.SYS P60 MFFFF
```

T L C - L I S P   D O C U M E N T A T I O N

P A R T   I I I

The TLC-LISP Reference Manual

[illegible]

[illegible]

[illegible]

Part III -- TLC-LISP Manual

This section is a complete catalog of the primitives, library functions, and constants in TLC-LISP. Each function and constant is listed in the index at the end of this manual. All functions include a short description and an example of their application.

Conventions

In the next sections we use the following conventions:

1. A word surrounded by angle brackets represents an element in the category named by the word. For example <object> represents the category of objects. This category contains all of the data objects that LISP may manipulate.

2. {<object>} represents zero or more instances (not necessarily identical) of elements in <object>.

3. Frequently we will wish to specify that an <object> be a member of a specific class of syntactic LISP objects.

   <atom> is anything that is not a dotted-pair (or a list). That includes symbols, vectors, classes, and numbers for example.

   <symbol> is comparable to an "identifier" in other languages; the first character is alphabetic and succeeding characters are either alphabetic, numeric, or selected special characters. Examples: A, A123, AGA-MEM-NON, but not 1DERFUL.

   <num> is expected to be numeric (fix, integer, or float) For example, 123 (fix), 32456 (integer, base 10), #[2]1011 (integer, base 2).

   <flt> is expected to be a floating point number. For example, 1.23 and 2.718E-4 but not 1 or "1" or A.

   <str> is expected to be a string. For example, "abcABC" and "123ASD" but not A or \A or 100.

   <char> is expected to be a character object. For example, \A and \1 but not A or "A".

   <sexpr> is any well-formed LISP symbolic expression, atomic or composite. For example, T, (A . B), and (A B C D) but not (A .).

> <list> is expected to be an empty or non-empty list object. For example, (A B C D) but not (A . B) or A. Empty matching parentheses (), and the atom NIL both represent the empty list.
>
> <vector> represents a vector object. For example [A B C D] is a non-empty vector. [] is the empty vector.

Special characters (like [,], ., and ") and particular symbols (like numerals) are used to identify occurrences of constant objects. A vector named READ-TABLE contains the information that defines how characters are interpreted by TLC-Lisp--which characters are delimiters; which can appear in an atom, number, or string, for example. For some advanced applications it may be useful to change this table. See the section on Input and Output for more details.

4. It is also convenient to specify that an <object> be a member of a semantic LISP class.

> <var> is a symbol that can be used as a variable. Therefore numbers are disallowed as are the LISP reserved words: T and NIL. This means, for example that the names of TLC-LISP's built-in functions are available as variable names. Of course, the redefinition of built-in functions must be done with great care. Appropriate use of packages offers a better solution than blatant redefinition.
>
> <env> is an object of type environment, and is a collection of alternating <var>s and <object>s. Such objects are used in closures to represent the local state information, and are used to represent methods, instance variables, and class variables in class-related objects.
>
> <fcn> is a LISP function; for example, <fcn> may be a primitive function object, a LAMBDA expression, or a closure.
>
> <pred> is a LISP form that is expected to be used as a predicate; that is, its evaluation yields a LISP truth-value, NIL for false, or non-NIL for true.
>
> <form> is a LISP expression that meets LISP's syntactic requirements for being an executable element. For example, (A B) is a <form> since it represents the application of a function named A to the actual parameter B; however (A . B) does not represent any application. <form> makes no claims about the evaluation; it could produce a value, cause an error, or fail to terminate.
>
> <stream> is an object that can be used as a sink or source for LISP input and output. These are either constructed from files or from LISP functions.

5. => is to be read "evaluates to". This notation is used in conjunction with many of the examples in the following sections.

6. Finally some general notes. The typical pattern we use for a definitional description will be:

(<name> {<arguments>}) <type>

where <name> is the name of the function being discussed. <Arguments> are the components expected in an application of <name>, and <type> describes the "calling style" of <name>. The most common instances of <type> are SUBR --a built-in, call-by-value function, and FSUBR --a special form. See Part I for a complete discussion of calling styles.

A few built-in functions are of type LSUBR, meaning they are call-by-value, but will take an arbitrary number of arguments. Some functions are defined in LISP --EXPR, FEXPR (special form) and MACRO are interpreted forms, PCODE, FPCODE (special form) and MPCODE (macro) are compiled forms.

With these calling-style considerations, **{<arguments>}** will be interpreted in two basically different ways:

a. As the types of the values passed to <name> with a specific number required for SUBRs and a variable number allowed for LSUBRs.

For example given:

(FOO <atom> <number> <sexpr>) SUBR or EXPR or PCODE

then a call (FOO (FIRST X) (ADD1 22) 'A) would fit the constraints provided that (FIRST X) evaluated to an atomic object since the value of (ADD1 22) is a number and the value of 'A is a symbolic expression; the body of FOO would receive these three values.

b. In the case of FSUBRs, as a pattern to be matched against the textual form of the argument, since the actual parameter is treated as a list, rather than an expression. For example,

(BAR <atom> <number> <sexpr>) FSUBR or FEXPR or FPCODE

could be called like (BAR X 22 'A). The body of BAR would see a single argument (X 22 (QUOTE A)) Note: X is an atom, 22 is a number, and the list (QUOTE A) is a symbolic expression.

Object Types in TLC-LISP

The following is a brief description of the types supplied in TLC-LISP. More detailed descriptions appear in the following sections.

SYMBOL

Symbols are similar to identifiers in other programming languages. They have three attributes: the print name, the value and the property list. The print name is a string of characters. The value can be any LISP object. The property list is a list of properties (or attributes) and values.

FLOAT

An IEEE standard format single precision floating point number. The 8087 numerics co-processor is supported.

INTEGER

A thirty-two bit signed binary integer.

FIX

A ten bit binary integer that (unlike integer objects) consumes no storage.

DOTTED PAIRS

The traditional LISP object, consisting of two other LISP objects. The constituent objects are historically referred to as the CAR and the CDR. Lists are a special case of dotted-pairs.

STRING

A vector of eight-bit characters; up to 63520 characters may appear.

CHAR

A single eight bit ASCII character, stored more efficiently than a one character string.

VECTOR

A one dimensional array of arbitrary LISP objects, up to 31760 long.

STREAM

The object for input and output, consisting of source or sink (which can be an arbitrary function) for characters and storage for a lookahead (next) character.

FILE

An object representing a disk file consisting of a name string, an access mode, a buffer and some operating system specific data structures. Files are not usually accessed directly, but are found in the source/sink field of a stream.

SUBR

A built-in call-by-value function.

FSUBR

A built-in function that does not evaluate its arguments.

LSUBR

A built-in call-by-value function with an arbitrary number of arguments.

EXPR

A user defined call-by-value function.

FEXPR

A user defined function that does not evaluate its arguments.

MACRO

A user defined function that constructs a list that is then evaluated a second time.

CLOSURE

An object consisting of a functional object and an environment of variables and associated values. When the closure is applied, the closure's environment overrides the current environment for references to those specific variables; it is of particular importance to note that assigment-type operations can affect the environment, and thus any changes to the closure's environment will be saved when the closure completes its computation.

ENV

An environment; a collection of variables and associated values. It can be used as "local state" for closures, or can be applied as a finite function.

PKG

A package; a collection of symbols and accessing rules, useful for preventing name conflicts.

PCODE

User defined pseudo-machine code.

FPCODE

User defined pseudo-machine code that does not evaluate its arguments.

MPCODE

User defined pseudo-machine code that functions as a macro.

TURTLE

A graphics-oriented object possessing properties like position, shape, color, pen state, and heading.

CLASS

A class object has four attributes: the superclass, the class variables (shared storage), the collection of messages and their associated methods, and the collection of instance variables that become part of any instance of this class.

INST

An instance of a class consisting of the instance variables (local storage) and the superclass.

UNBOUND

A special object indicating that an atom has no value.

NONE

A special type used internally.

ILLEGAL

A special type used internally.

ALOAD

The type for "virtual objects"; this type contains a string representing a file name, and a representation of a position in that file where we can find the Lisp text that defines that object. See the section on Autoloading for more details.

Defining Functions

There are three fundamental types of functions in TLC-LISP: call-by-value functions, special forms, and macros. In the Evaluation section we discussed these basic strategies. Here we introduce techniques for adding new functions to the LISP library. Such user-defined call-by-value functions are called EXPRs; new special forms are called FEXPRs.

The following built-in functions are used to add new definitions to the LISP library.

(DE <var> <parameters> {<form>}) FSUBR

creates a call-by-value function from the parameters and the forms, and then it installs that definition as the value of the <var>. The <parameters> may contain the special indicators &OPT (or &OPTIONAL), &AUX and &REST. The value returned by DE is the symbol <var>. When <var> is invoked, <parameters> are bound to the appropriate actual parameters; then the <form>s are evaluated sequentially, from left to right.

The makeup of <parameters> is sufficiently involved to demand its own discussion. For a detailed treatment, see the section, Introduction to TLC-LISP in Part I. However, a few simple examples follow:

```
     (de FACT (x &OPT (n 1))
       (if (zerop x)
           n
           (fact (subl x)(mul x n)) ))
```

```
(FACT 5) => 120
```

This is a definition of the venerable factorial function. While the next is just a toy.

```
     (de WHIZ (x &OPT (y (cons 5 x)))
        y)

     (whiz 2 7) => 7, and (whiz "ab") => (5 . "ab")
```

(DF <var> ((<param> {&AUX {<params>}}) {<form>}) FSUBR

DF is similar to DE, but for special forms. Note the single <param>. When the special form <var> is applied, it looks just like an ordinary call (as in (fl 1 (addl 2) 3)), but an FEXPR binds a list of the unevaluated parameters (i.e. (1 (addl 2) 3)) to <param>.

For example assume the definition for fl is:

```
(df fl (x) (first x))
```

then

```
(fl (addl 42) 4 3) => (addl 42)
```

since the list ((addl 42) 4 3) is bound to x. Be clear that the value is the list (addl 42) and not the value 43.

Though a DF must have exactly one required parameter, and no &OPT or &REST parameters, it may specify a set of local (&AUX) variables to be allocated on entry to the special form.

For further information see the discussion in Part I.

The usual LISP definition is a "DE", with special forms invoked only if the user wishes to control the parameter evaluation in a special way. Such evaluation will involve explicit calls on the evaluator using EVAL to execute pieces of the text. Many applications that have traditionally been done with such DF's are, in fact, better handled by macros. So we suggest that careful consideration be given to situations that appear to demand new Special Forms. There may be better ways to address the problem.

The final member of the function-defining trio is used to introduce macro definitions. LISP macros exploit the program-data duality of LISP even more than special forms do.

A LISP macro definition has the appearance of a definition with only one parameter.

(DM <var> (<param> {&AUX {<params>}}) {<form>}) FSUBR

Associates the macro definition, represented in

```
(<param> {&AUX {<params>}}) {<form>}),
```

with the name <var>. As with DF, a DM has only one formal parameter, and may also specify auxiliary parameters. As with other styles of invocation, a macro call looks like an ordinary function call. In contrast to FEXPRS, the entire call is bound to that single parameter. For example: given a macro definition of the form:

```
     (dm test (l) ...)
```

the call

```
     (test (first x) 4 'now)
```

will bind variable l to the list

```
     (test (first x) 4 (quote now)).
```

The body of the macro definition is free to manipulate that text with all the power of LISP. So far, the effect is similar to that of a special form. However, the value computed within the macro is expected to be a new expression; then, as we leave the macro call, that expression is evaluated by the interpreter (a second evaluation) and the resulting value is the final value of the macro call. Before we give an example, we summarize the transformations: the original call (program) is passed to the macro (data) where it is manipulated (data) and finally reevaluated (program). Let's examine an example now:

```
     (DM NCONCAT (L) (LIST 'CONCAT (CADR L) NIL))
```

Consider a call (NCONCAT 6).

The list (NCONCAT 6) gets bound to L; then the evaluation of the body gives a list (CONCAT 6 NIL). Finally, that list gets evaluated and (NCONCAT 6) returns (6) as value.

One of the traditional applications of special forms that can be better handled by macros is the description of functions with an arbitrary number of arguments (like +) by writing them as macro-expansions of a function with a fixed number of arguments.

```
(+ 1 2 (subl 3))) =macro-expander=> (add 1 (add 2 (subl 3)))

(dm + (l)
   (if (eq (length l) 3)
        '(add ,(second l) ,(third l))
        '(add ,(second l) (+ ,@(rest l 2)))))
```

a somewhat cryptic, but fool-proof way of translating the form using macros and backquoting.

The alternative of defining such functions as Special Forms leaves us the job of explicitly evaluating the parameters ourselves, as in:

```
(df + (l) (+expand (l))

(de +expand (l)
 (if (null l)
     0
     (add (eval (first l)) (+expand (rest l)))))
```

But such explicit calls on the evaluator open the door to scoping problems--what happens when the expression (FIRST L) involves the name L? Macros dispense with a lot of this grief. See the _Evaluation_ section in Part I for a discussion of macros, scoping, and "backquote" (a technique to simplify the syntax of macro definitions).

Macros are able to express a complex behavior in terms of simple transformations that can be carried out on the program text; thus macros can be used to obscure many implementation details. They are an exceptionally powerful technique for "information hiding".

Displacing Macros

If the value of the atom SMASH is non-NIL then all macros are 'self-destructive' or 'displacing' macros. If the value of SMASH is NIL then macros are evaluated each time they occur in an expression.

For example, if we wanted to define (IS-DOG X) to be equivalent to (EQ (FIRST X) 'DOG), we could write:

```
     (de IS-DOG (x)
       (eq (first x) 'dog))
```

We would rather define IS-DOG as a macro:

```
     (dm IS-DOG (x &AUX (arg (second x)))
       (list 'eq (list 'first arg) ''dog))
```

or equivalently using the back-quote facility:

```
     (dm IS-DOG (x &AUX (arg (second x)))
       `(eq (first ,arg) 'dog))
```

Notice the similarity in style between the DE-form and the backquote-form.

Regardless of how we define IS-DOG, we can use it (in the same way) as in the following definition:

```
     (de TEST (n m)
       (if (and (is-dog n)
                (is-dog m) )
           'compatible
           'no-way))
```

Assume IS-DOG is not a displacing-macro. Then each time we execute TEST, the IS-DOG macro creates a new list which gets evaluated and thrown away. However if we execute:

```
     (SETQ SMASH T)
```

and execute TEST once, then the definition of TEST will change to reflect the expansion of the IS-DOG macro, thus:

```
     TEST => (lambda (n m)
               (if (and (eq (first n) 'dog)
                        (eq (first m) 'dog) )
                   'compatible
                   'no-way)
```

The advantage of 'displacing' macros (value of SMASH non-NIL) is execution speed. The (eq ...) inside TEST executes faster than the equivalent expr IS-DOG and faster still than the non-displacing macro IS-DOG. The disadvantage is poor readability. If during debugging you come across the displaced version of TEST you may not recognize (or understand) it. The (eq (first ...)) construct appearing inside TEST obscures the purpose of TEST with unnecessary implementation details -- the original definition with its self-documenting IS-DOG construct is much more readable.

Note that since SMASH is an atom it may be used as an &AUX variable, allowing macros to be displacing inside debugged functions and non-displacing otherwise. The editor EDIT and the pretty printer PP make use of this feature.

Function Constructors

The functions DE, DF, and DM are used typically at the "top-level" of LISP to make permanent definitions. They destroy the current contents of the value cell associated with the function name. However there are also two operations, LAMBDA and FLAMBDA, that are used to make more temporary function definitions.

(LAMBDA <parameters> {<form>}) FSUBR

Creates a functional object (expr) whose formal parameters are <parameters> and whose body is the sequence {<form>}.

This functional object, called a lambda expression, can be used anywhere a call-by-value function is expected. This means that functions need not be associated with a name before they can be used; such lambda expressions are therefore often called anonymous functions. For example:

```
((LAMBDA (X Y) (ADD  X Y)) 3 5) will evaluate to 8.
```

We bind X to 3 and Y to 5, and then evaluate (ADD X Y).

These functional objects can be passed around freely in TLC-LISP, even to the point of using them as arguments to functions and returning them as values of functions.

Currently, TLC-LISP supports only a subset of the full power of functional objects. Complete functional objects would be able to remember the entire state of the system (values of all existing atoms) in effect at the time of their creation. We supply CLOSURES which require the user to specify the subset of the system state desired. See the section Function Manipulating Functions.

(FLAMBDA (<var> {&AUX {<params>}}) {<form>}) FSUBR

FLAMBDA is similar to LAMBDA, but constructs an anonymous special form (fexpr).

The system uses lambda expressions within the implementation of DE. This operation has two purposes: to define a functional object, and to associate that object with a name.

Since we expect the name association to be rather permanent we use a destructive binder named SET--a form of the assignment statement. Then we can define DE as:

```
     (dm DE (l)
       (list 'set (second l) (concat 'lambda (rest l 2))))
```
or

```
     (dm DE (l)
       `(set ,(second l) (lambda ,@(rest l 2))))
```

Of course, DF and DM can be defined in a similar fashion, and in that context, the system also supplies an **MLAMBDA** construct. However, MLAMBDA may not be used anonymously, but only in the context of defining a named macro. Recall that part of the macro call is the name of the macro.

Tail Recursion Elimination

The TLC-LISP interpreter eliminates many cases of tail recursion. In particular, it is eliminated in exprs whenever the last expression in the expr body is an application of the expr itself or when the last executable expression is one of the subrs IF, COND, SELECTQ, PROGN and the last expression in the subr expression is an application of the expr. Thus the following functions will never run out of stack space regardless of the value of the argument:

```
        (de TEST1 (n)
          (prin0 n)
          (test1 (add1 n)) )

        (de TEST2 (n)
          (if (not (zerop n))
              (test2 (sub1 n))
              (print "end of the line") ))
```

However, the following function is not tail-recursive:

```
        (de FACT (n)
          (if (zerop n)
              1
              (* n (fact (sub1 n)))))
```

since the interior call on fact is not the last expression.

Simply because one formulation is not tail-recursive it does not mean that the function has no tail-recursive formulations. For example:

```
(de factl (n &opt (m 1))
  (if (zerop n)
      m
      (factl (subl n) (* n m))))
```

Note that either clause of an IF expression may be tail-recursive. The same situation holds for COND:

```
(de TEST3 (n)
  (cond
    ( (eq n 100000)
      'big )
    ( (eq n 300000)
      'bigger )
    ( t
      (prin0 n)
      (test3 (addl n)) ) )
```

and the tail-recursion on TEST3 will be recognized and reomved.

Evaluation

The interpretation process supplies (and imposes) a default evaluation for the constituents of LISP expressions. The "top-level" of LISP is a "calculator mode" in which the user types an expression, LISP evaluates it, prints the result and prompts for another expression. This top-level loop is called the READ-EVAL-PRINT loop. This gratuitous evaluation often suffices, but sometimes it is convenient to impose other evaluation regimes.

One also needs to evaluate lists that have the appearance of expressions, and thus exploit the program-data duality of LISP. This is accomplished with EVAL, which explicitly calls the evaluator, allowing the dynamic evaluation of expressions which have been constructed by the data manipulating operations of the language.

(EVAL <form>) SUBR

This is the call on the LISP evaluator. The argument is a data structure that is expected to conform to the syntactic rules for LISP programs. The value computed by EVAL is the value of <form>. Note that EVAL is a SUBR, and therefore the argument to EVAL will be evaluated before EVAL is called.

```
(EVAL 3)  =>  3

(SETQ X 'A)
(SETQ A 4)
(EVAL 'X) => A
```

since the actual parameter passed to EVAL is the atom X.

```
(EVAL X) => 4
```

since the actual parameter passed to EVAL is the atom A.

```
(EVAL '(FIRST '(1 2 3))) => 1

(EVAL (LIST 'CAR (LIST 'CONS X 'X))) => 4
```

since the value passed to EVAL is the list (CAR (CONS A X)).

(EVLIS ({<form>})) SUBR

EVLIS creates a list of the evaluated <form>'s. Its effective definition is:

```
     (DE EVLIS (L)
       (IF (NULL L)
           ;then; ()
           ;else; (CONCAT (EVAL (FIRST  L))
                          (EVLIS (REST L)) )))
```

Note: we have used our comment conventions to emphasize the structure of the IF control primitive.

```
     (EVLIS (LIST 3 '(ADD1 2) '(FIRST  (LIST '(ADD1 2) 3))))
          => (3 3 (ADD1 2))
```

since EVLIS will be passed the list

```
     (3 (ADD1 2) (FIRST (LIST (QUOTE (ADD1 2)) 3)))
```

Or using the bindings of X and A given above with EVAL,

```
     (EVLIS (LIST X 'X A)) => (4 A 4).
```

(PROG1 {<form>}) FSUBR

Performs left-to-right evaluation of the <form>'s, returning the value of the first <form>.

```
     (PROG1 (CONS 1 3) 4) => (1 . 3)

     (PROG1) => NIL
```

(PROGN {<form>}) FSUBR

Similar to PROG1, but returns the value of the LAST <form>.

```
     (PROGN (CONS 1 3) 4) => 4

     (PROGN 1 2 (ADD1 1) (CAR '(A . B))) => A
```

(QUOTE <sexpr>) FSUBR

QUOTE is the LISP primitive to stop evaluation. It is most commonly abbreviated by the read-macro single-quote ('). The effective definition is:

```
      (DF QUOTE (L) (FIRST L))
```

(TOPLEV) SUBR

TOPLEV is the name of the function that controls the user interface. It is initially defined to be approximately:

```
(de TOPLEV ()
  (do () (nil)                ; forever
      (prin0 '>>>)
      (print (eval (read)))))
```

The user may supply a different TOPLEV --simply redefine TOPLEV. A certain amount of caution should be exercised, however; bugs in a new TOPLEV may destroy the system. (see TAPPLY and RESTART in the section on Errors and Debugging.).

Interpreter Modifiers

TLC-LISP supplies two functions to modify the behavior of the interpreter.

(AP <type> &OPT <fcn> or NIL) SUBR

Returns or sets the apply behavior of objects of type <type>. The apply behavior is invoked when an expression like (APPLY <obj> <arglist>) is evaluated. The <fcn> would receive the two arguments <obj> and <arglist>. If <fcn> is NIL, the AP-property is reset. See also EAP.

As an example of AP, the default apply-behavior of vectors could be defined as follows:

```
(de APVECTOR (vect l &AUX (index (first l)))
  (vref vect index))

(ap 'vector apvector)
```

Thus:

```
(setq vl [a b c])
(apply vl '(2)) => b
```

(EAP <type> &OPT <fcn> or NIL) SUBR

Returns or sets the eval-apply behavior of objects of type <type>. The eval-apply behavior is invoked when an expression like (<type> {<args>}) is evaluated. The <fcn> would receive the two arguments, the evaluated function position <type> and the unevaluated complete expression (<type> {<args>...}). If <fcn> is NIL, the EAP property is effectively removed. See also AP.

For example, the eval-apply behavior of packages could be defined as follows:

```
     (de EAPPKG (p l &AUX (package p))
       (apply progn (rest l)))

     (eap 'pkg eappkg)
```

Then if we evaluate:

```
     (pp: (load "pp.lsp"))
```

EAPPKG is invoked with the package pp: bound to P and the complete expression (pp: (load "pp.lsp")) bound to L. EAPPKG locally binds PACKAGE to pp: then evaluates

```
(progn  (load "pp.lsp"))
```

thus insuring that the file is read into the desired package.

Functions to Manipulate Functions

The functions in this section operate with one or more parameters that are expected to be functional objects.

(GETFN <fcn>) SUBR

(PUTFN <fcn> <list>) SUBR

These functions allow us to manipulate the text of a user defined interpreted function. GETFN extracts a list representing the body of the function <fcn> if it is a user-defined function. PUTFN is used to re-install <list> as a function definition of <fcn>. <list> must be of the correct form to represent a functional object.

```
      (de FOO (x y)
        (cons x y) )

     (GETFN FOO) => ((x y) (cons x y))
```

Note that:

```
     (TYPE FOO) => expr
```

but

```
     (TYPE (GETFN FOO)) => list
```

GETFN can be useful when debugging a macro. We can use it to examine the value returned by a macro before the second evaluation, thus:

```
     (de MAC (l &aux (macro (eval (first l))))
     ; returns result of macro before second evaluation
       (apply (apply lambda (getfn macro))
              (list l) ))


     (MAC '(FOR (I 1 10) (PRINO I)))

       =>     (do ( (i 1 (addl i)) )
                  ( ((gt i 10) nil) )
                  (prin0 i) )
```

(APPLY <fcn> <list>) SUBR

Apply the function <fcn> to the list of arguments represented in <list>. The arguments in <list> are not evaluated.

```
(APPLY ADD (LIST (ADD1 5) (MUL 4 5))) => 26
```

Since APPLY is a call-by-value function, its parameters are evaluated; therefore it gets passed the (primitive) functional object for ADD and the list (6 20).

```
(APPLY CONS (LIST 'A 'B)) => (A . B)
```

since APPLY gets the functional object associated with CONS and the list (A B).

```
(APPLY (LAMBDA (X Y) (LIST X "is"  Y)) '(LISP NEAT))

      =>   (LISP "is" NEAT)

(SETQ X 4)

(APPLY CAR (LIST (CONS X 'X))) => 4
```

APPLY, like EVAL, seldom needs to be explicitly applied. APPLY can be used with SUBRs and EXPRs, but may not be used with a special form or macro in the <fcn> position.

(FUNCALL <fcn> {<arg>}) LSUBR

FUNCALL is like APPLY except that the arguments are not in list form. That is, FUNCALL applies its first argument to the rest of its arguments.

```
(FUNCALL ADD 1 2 3) => 6

(FUNCALL CONCAT (LIST 1 2) '(3 4 5)) => ((1 2) 3 4 5)
```

(MAP <fcn> <list>) SUBR

Apply the function <fcn> successively to <list> and its tails. The value returned is NIL. MAP is equivalent to:

```
(DE MAP (FN L)
  (IF (NULL L)
      ()
      (FN L)
      (MAP FN (REST L)) )))
```

Note the implicit application of FN to L.

Here's a simple example:

```
(MAP PRINT '(A (B C) D))
  => (A (B C) D)
     ((B C) D)
     (D)
     NIL
```

where the final NIL is the value returned.

(MAPLIST <fcn> <list>) SUBR

Apply the function <fcn> successively to <list> and its tails. MAPLIST returns the list of these results. Its definition can be given as:

```
(DE MAPLIST (FN L)
       (IF (NULL L)
           ()
           (CONCAT (FN L) (MAPLIST FN (REST L))) ))
```

We could define EVLIS as:

```
(DE EVLIS (L)
  (MAPLIST (LAMBDA (X) (EVAL (FIRST X))) L))
```

(MAPVEC <fcn> <vector> &OPT <vector1>) SUBR

Apply the function <fcn> to each element of <vector> or, if <fcn> is binary and <vector1> is present, to consecutive elements of both vectors, building a new vector.

```
(MAPVEC ADD1 [1 2 3]) => [2 3 4]

(MAPVEC ADD [1 2 3] [3 2 1]) => [4 4 4]
```

(CLOSURE <fcn> <list or env>) SUBR

If the second argument of the closure is a list, it must be of the form of alternating <var>s and <object>s. In this case, the system creates an object of type <env> from these elements. In either case, the <env> is associated with the functional object <fcn> in such a way that the <vars> and their associated values will be established as the current bindings whenever the closure object is applied as a function. Also, changes made to these <var>s while the closure is being applied will be "remembered" for the next application of the closure. Closures are implemented using the equivalent of UNWIND-PROTECT so that throwing out of a closure will guarantee that the closure's local variables are updated.

```
     (DE YLIST (X)
       (LIST X Y))

     (SETQ F (CLOSURE YLIST '(Y 2))

     (SETQ Y 'A)

     (F Y) => (A 2)
```

F executes YLIST in an environment where Y has the value 2, whereas:

```
     (YLIST Y) => (A A)
```

YLIST is executed in the global environment where Y has the value A.

A more realistic example is the LINE-EDITED-STREAM function from the file SYS.LSP:

```
(de LINE-EDITED-STREAM (source echo)
  (stream (closure linebuffer
                   (env 'source source
                        'echo echo
                        'buffer (newstr 100 *eof*)
                        'index 0 ))))))
```

where the second argument to the closure becomes an environment that contains the four variables and their values. We want them to be local variables because we want to have several streams in the system, and each must have its own buffers, indices, and functions. Within the context of a closure we're assured that this will be the case, and are thus able to specify the following definition for linebuffer:

```
(de LINEBUFFER (&aux (c (nth buffer (setq index (addl index)))) )
; if buffer not empty, return next char else refill buffer
  (cond
    ( (eq c *eof*)
      (setq index 0)
      (getline)
      (linebuffer) )
    ( t  c ) ))))

(de GETLINE (&aux (c (source)) )
; at end of buffer, refill, handle carriage return, ctl-x and backspace
  (cond
    ( (eq c *cr*)
      (echo *cr*)
      (putchar buffer (addl index) *cr*)
      (putchar buffer (add 2 index) *eof*)
      (setq index 0) )
    ( (eq c ^x)
      (rep index (backone))
      (setq index 0)
      (getline) )
    ( (or (eq c *backspace*)
          (eq c *rub*) )
      (if (zerop index)
          (getline)
          ; else
          (backone)
          (setq index (subl index))
          (getline) ) )
    ( (lt (ascii c) 32)  ; ignore other control characters
      (getline) )
    (t (echo c)
       (putchar buffer (setq index (addl index)) c)
       (getline) ) ))))

(de BACKONE ()
  (echo *backspace*)
  (echo *space*)
  (echo *backspace*))))
```

Flow of Control

Call-by-value, recursion, and the parameter evaluation mechanism impose an order of execution on LISP computations. These are examples of implicit control. A traditional programming language also contains explicit control structures in the form notations that specify which of a set of alternative computations are to be executed.

Control structures depend on the existence of predicates: LISP functions whose values are interpreted as the truth values "true" and "false". In LISP we take NIL as the representation of false, and any non-NIL value is taken as truth. See the discussion preceding MEMQ for more details.

The primary explicit control structure in any LISP is the conditional expression. TLC-LISP supplies two forms:

```
(IF <pred>
      <form1>
      {<form2>}) FSUBR
```

The expression <pred> is evaluated first; if it returns a value other than NIL then <pred> is considered true and the value of the IF-expression is the value of <form1>. Otherwise the sequence {<form2>} is evaluated and the value of the IF is the value of the last <form2>.

```
(IF (FIRST X) 1 2)
```

gives value 1 if (FIRST X) is non-NIL, and gives 2 otherwise.

Think of the IF as reading "if <pred> then <form1> else {<form2>}".

Note that there is exactly one <form1>, but there can be a sequence of actions specified as {<form2>}.

The second form of conditional expression is the COND:

```
(COND
    (<pred1> {<form1>})
     ...
    (<predn> {<formn>}) ) FSUBR
```

Each construct of the form (<predi> {<formi>}) is called a clause. The evaluation of a COND-expression is as follows. The predicate, <pred1>, of the first clause is evaluated. If it yields a non-NIL value then the elements of {<form1>} are evaluated and the value of the COND is the value of the last element in {<form1>}. If NIL was returned by the <pred1>, then the <form1>s are not evaluated, but the process continues by looking at the next clause and repeating the above process.

If none of the <predi>s give non-NIL, then the value of the COND is NIL. However, it is good programming practice to make the last predicate, <predn> be the constant predicate T. This way the <formn>'s are able to handle all exception cases. Such usage acts as an "otherwise" clause.

A useful degenerate case occurs when a clause is a single expression, (<pred>); that is, the collection {<form>} is empty. In this case, if <pred> evaluates to a non-NIL quantity then the value of the conditional expression is just that value. Used with the NIL/non-NIL truth-values of LISP, this abbreviation can be particularly convenient. For example, if the value of <pred> is either expensive to compute or causes a side-effect, then a conditional like:

```
(COND
  (<pred> <pred>)
  ... )
```

is inappropriate since <pred> will be evaluated twice. The effect is better described by:

```
(COND
  (<pred>)
  ... )
```

(OR {<form>}) FSUBR

Evaluate the sequence of <form>s from left-to-right, terminating that process if one returns a non-NIL value. That value is the value of the OR-expression. If no <form> gives a non-NIL value, then the value of the OR is NIL.

```
(OR (ATOM '(A B))
    (CONS 1 2)
    (CAR 1) ) => (1 . 2)
```

Note that the value of (CONS 1 2) is an acceptable representation for "true" (being non-NIL). Further note that the expression

```
(CAR 1)
```

which would yield an error never gets evaluated.

```
     (OR {<form>}) is equivalent to:

     (COND
       (<form-1>)
       (<form-2>)
       ...
       (<form-n>) )
```

(AND {<form>}) FSUBR

AND evaluate the <form>s from left-to-right, stopping the evaluation and returning NIL as soon as one of the <form>s gives a NIL value. If no <form> gives NIL, return the value of the last <form> as the value of the AND-expression.

```
     (AND (CONS 1 2)
          NIL
          (CAR 1) ) => NIL

     (AND (CONS 1 2)
           T
           4
           (ADD1 2) ) => 3

     (AND {<form>}) is equivalent to:

     (COND
       ( (NOT <form-1>) NIL)
       ( (NOT <form-2>) NIL)
       ...
       ( <form-n> ) )
```

(NOT <form>) SUBR

Returns NIL if <form> is non-NIL, and T otherwise.

```
     (NOT T) => NIL

     (NOT NIL) => T

     (NOT 1) => NIL
```

(SELECTQ <form> {(<objecti> {<formi>})}) FSUBR

The value of <form> is compared successively against each <objecti>; the <objecti>s are not evaluated (the Q in SELECTQ stands for QUOTE). The type of match is determined by the structure of <object>. If the <objecti> is a list, then the match uses MEMQ on <objecti>. If <objecti> is not the symbol T, the match uses the predicate EQ; if the <objecti> is one of the atoms T, OTHERWISE, or OW then the match succeeds automatically.

If a comparison is successful the match process halts and the corresponding <formi>s are evaluated. The value of the SELECTQ is the last <formi>. If no comparison is successful, then the value of the SELECTQ is NIL.

```
(SELECTQ (SENSE X)
  (LOOK ...)
  ((SMELL TOUCH HEAR) ...)
  (OW (LOSE X)))
```

is equivalent to:

```
(LET ( (TEMP (SENSE X)) )
  (COND
    ( (EQ TEMP 'LOOK) ...)
    ( (MEMQ TEMP '(SMELL TOUCH HEAR)) ...)
    ( T (LOSE X)) ))
```

where we have to assign the value of (SENSE X) to a temporary variable to keep from computing (SENSE X) more than once.

Variations of SELECTQ appear in Pascal-like languages under the guise of CASE statements. Unfortunately since Pascal treats them as statements, much of their power is wasted.

(LABEL {<form>}) LSUBR

LABEL evaluates {<form>} in the context of the last (dynamically) surrounding lambda expression. This is a generalization of the LISP1.5 LABEL operator, that allowed recursive definitions without explicit naming. For example:

```
(LAMBDA (N) (IF (ZEROP N)
                1
                (MUL N
                     (LABEL (SUB1 N)))))
```

is a definition of the factorial function.

(CATCH <symbol> {<form>}) FSUBR

(THROW <symbol> {<form>}) FSUBR

This pair of functions operates together to supply a non-structured type of function exit. When a CATCH expression is entered, the <symbol> is noted and the body, {<form>}, is evaluated as a sequence of expressions. If, during that evaluation, an expression of the form (THROW <symbol> {<formi>}) is encountered, then the {<formi>} are evaluated and the value of the last <formi> is returned as the value of the CATCH expression. If no such form is encountered, the value of the CATCH expression is the value of the last <form> in the body of the CATCH.

```
(CATCH EXIT
       (MAP (LAMBDA (X)
              (AND (NUMBERP (FIRST X))
                   (THROW EXIT (LIST 'YES (FIRST X)))))
             '(A B 2 C) )
       'NO ) => (YES 2)
```

In order for a program to determine whether a CATCH terminates with or without throwing, the following technique is useful. Make the last expression in the body of the CATCH be NIL. Insure that any THROW always throws a non-NIL value. you can then test the result of the CATCH expression, if NIL then no THROW occurred otherwise the value is the value of the THROW. Thus:

```
(if (catch label
           ...
           (if something (throw label something-non-NIL))
           ...
           nil )
    (print "threw")
    (print "Did not throw") )
```

Only in the case where the catch body returns nil do we know that no matching throws have occurred since any throw will cause the catch to return a non-nil value.

If a THROW expression is encountered which does not have a dynamically surrounding CATCH expression with a matching <symbol>, then a CATCH-NOT-FOUND error is generated.

The CATCH-THROW pair is particularly useful for effecting an immediate return from a sub-computation without requiring the program to explicitly exit through all the intervening levels of functions that have been called but not yet exited. This strategy would require all functions involved to include explicit tests for exit conditions and corresponding function-exit clauses.

(UNWIND-PROTECT <form1> {<form2>}) FSUBR

UNWIND-PROTECT is a "super" catcher to guard against THROWs that might otherwise throw too far, too fast.

<Form1> is evaluated then {<form2>} is evaluated. The value returned is the value of the last expression in {<form2>}. This effect is equivalent to:

```
(progn <form1> {<form2>})
```

However, if during the evaluation of <form1> we attempt to throw to a catch label outside the unwind-protect then {<form2>} is guaranteed to be evaluated before the catch is completed. For example:

```
(de TEST ()
  (catch error (try this))
  (try that))

(de TRY (a)
  (turn-on-faucet)
  (unwind-protect (play-around)
                  (turn-off-faucet) )
  (print-results))

(de PLAY-AROUND ()
  ...
  (if (bad-things) (throw error 'bad))
  ... )
```

The UNWIND-PROTECT in TRY guarantees that the faucet will be turned off before the (try that) form is evaluated even if bad-things happen in play-around.

Unwind-protect will also work when using RETFRAME.

```
(DO ( {(<var> <init> <iter>)} )
    ( {(<exitp> {<exitval>})} )
    {<form>} )                    FSUBR
```

DO is the TLC-LISP iteration function. We will discuss the most general form of DO first, and follow that with an analysis of several useful subcases. There are four basic parts to the semantics of the DO expression:

1. The initialize phase. When the DO is entered, the <init> forms are evaluated and lambda-bound in parallel to their corresponding <var>s. This means: a) that the <var>s act as local variables within the scope of the DO, and b) that all of the initializations are performed in the environment that surrounds the DO.

2. The exit tests. Next, we test the <exitp>s in a fashion analogous to the semantics of a conditional expression. If we find a true exit-condition, we evaluate the associated <exitval>s and exit the DO, unbinding any local DO-variables. The value of the DO is the value of the last <exitval>. If none of the exit-conditions is true we move to phase 3, entering the body phase.

3. The body phase. The body of the DO, consisting of the <form>s, is evaluated in left-to-right order.

4. The iterate phase. Following the body phase, we evaluate the <iter> forms; again, this is done in parallel. Only now, we assign the new values to their corresponding <var> rather than lambda-bind them (this is analogous to the way tail-recursion elimination is implemented). After all the iterators are evaluated, we loop to phase 2 and check the end conditions.

This constitutes the basic loop of the DO. Here are some useful special cases:

```
(DO () ...)
```

If there are no var-init-iter triples, we have no local variables. The execution of the DO involves only the exit-tests and the body.

```
(DO ((var1) (var2 init) ...) ...)
```

If a var has neither an initial value nor an iterator, then it is initialized to UNBOUND. If a variable is followed by only one form, that form is taken to be an initialization value; that value is lambda-bound to the variable, but the variable is ignored in the iterate phase (of course the value can be modified within the DO by a SETQ).

```
     (DO  (...)  (NIL) ...)
```

In this case the predicate will never be true, and so the DO will continue without end (unless it contains a THROW form.)

```
     (DO (...) () ...)
```

In this case the body is executed only once.

```
     (DO  (...)  (...))
```

If no body is present then we pass directly to the iterate phase.

Here are several other control structures expressed as equivalent DO formulations:

```
     (LET ({(var init)}) body) is (DO ({(var init)}) () body)

     (WHILE pred body) could be defined as (DO () (((NOT pred))) body)
```

We could define a membership predicate as:

```
     (DE MEMBER (X L)
       (DO ( (L L (REST L)) )
       ( ((NULL L) NIL)
         ((EQUAL (FIRST L) X) T)) )))
```

where the body segment is empty. Here are some useful macros that could be defined using DO:

```
     (REPEAT <count> {<forms>}) is:

     (dm REPEAT (l &aux (body (rest l 2))
                        (count (second l))
                        (var (gensym)) )
       '(do ( (,var 0 (addl ,var)) )
            ( ((ge ,var ,count) nil) )
         ,@body ))
```

(FOR (<var> <initial> <final> &OPT (<increment> 1))
{<form>}) is:

```
(dm FOR (l &aux (var-list (second l))
                (body (rest l 2))
                (var (first var-list))
                (init (second var-list))
                (final (third var-list))
                (incr (selectq (length var-list)
                        (3 '(add1 ,var))  ; default
                        (4 '(add ,(fourth var-list) ,var))
                        (ow (error 1)) )) )
'(do ( (,var ,init ,incr) )
     ( ((gt ,var ,final) nil))
  ,@body ))
```

(FOREVER {<form>}) is:

```
(dm FOREVER (l)
  '(do () (()) ,@(rest l)))
```

(LET ({(<var> <form-1>)}) {<form-2>}) FSUBR

This function is equivalent to:

((LAMBDA ({<var>}) {<form-2>}) {<form-1>})

The "LET-style" is attractive since it places the <var>s in closer proximity to their binding forms, <form-1>s, thereby increasing readability.

(REP <num> {<forms>}) FSUBR

Evaluates {<forms>} <num> times. REP's advantage over DO is that it will allocate only one integer object per invocation whereas the equivalent DO could allocate one integer per iteration if <num> is sufficiently large.

Recognizers and Predicates

A recognizer is a special predicate which tests the 'type' of its argument. Though LISP variables are type-free, meaning that a variable can contain any legal LISP object, each LISP object has a distinguishable type. The LISP recognizers are predicates that the programmer can use to determine the type of a value.

(ATOM <object>) SUBR

ATOM returns T if <object> is not a list or dotted pair. It returns NIL otherwise. Symbols, strings, and numbers are atomic quantities, for example.

```
(ATOM 3) => T

(ATOM "AB") => T

(ATOM (ATOM '(3 . "ABC"))) => T

(ATOM 'CONS) => T

(ATOM CONS) => T  ; The value of CONS is a SUBR

(ATOM [1 2 3]) => T
```

Compare the behavior of ATOM with that of SYMBOLP.

(LISTP <object>) SUBR

This recognizer returns T if its argument is a composite object. Composite objects are lists and dotted pairs.

```
(LISTP 4) => NIL

(LISTP (CONS 1 'A)) =>T

(LISTP (LIST 1 'A)) => T

(LISTP NIL) => T    ; Since NIL represents the empty list.
```

(SYMBOLP <object>) SUBR

(NUMBERP <object>) SUBR

(FIXP <object>) SUBR

(INTEGERP <object>) SUBR

(FLOATP <object>) SUBR

(CHARP <object>) SUBR

(VECTORP <object>) SUBR

(STRINGP <object>) SUBR

(PKGP <object>) SUBR

(ENVP <object>) SUBR

(CLASSP <object>) SUBR

(INSTP <object>) SUBR

These recognizers check for the occurrence of a specific type.

```
(SYMBOLP 4) => NIL
(SYMBOLP "BAC") => NIL
(SYMBOLP 'A) => T

(NUMBERP 4) => T
(NUMBERP 3.3E4) => T
(NUMBERP 'A) => NIL

(FIXP 3) => T
(FIXP 1.2) => NIL

(CHARP \A) =>T
(CHARP "A") => NIL
(CHARP 'A) => NIL

(STRINGP \A) => NIL
(STRINGP "ABC") => T
(STRINGP 'ABC) => NIL
```

(PROCP <object>) SUBR

This recognizer returns T if <object> is a functional object. Functional objects are of type SUBR, LSUBR, FSUBR, EXPR, FEXPR, CLOSURE, MACRO, PCODE, FPCODE or MPCODE. If you need to have more detailed type information, TYPE will provide it. If <object> is not a functional object, NIL is returned.

```
(PROCP PROCP) => T

(PROCP COND) => T
```

(BOUNDP <symbol>) SUBR

Returns T if <symbol> has a value other than UNBOUND otherwise returns NIL.

```
(BOUNDP 'CONS) => T

(UNBIND 'A)
(BOUNDP 'A) => NIL
```

(NULL <object>) SUBR

NULL returns T just in the case that <object> is the empty list.

```
(NULL '(A)) => NIL

(NULL (REST '(A))) => T

(NULL (NULL '(A))) => T

(NULL 3) => NIL
```

(EMPTY <object>) SUBR

EMPTY returns T in the case that <object> is the empty string, vector or list.

```
(EMPTY "ABC") => NIL

(EMPTY '(TRASH . CAN)) => NIL

(EMPTY "") => T
```

(TYPE <object>) SUBR

This is a general type-extraction function, returning an atom that describes the type of the argument <object>.

```
(TYPE 'TYPE) => SYMBOL

(TYPE TYPE) => SUBR

(TYPE (CONS 1 2)) => LIST  ; the type of dotted pairs

(TYPE (CONCAT 1 '(2))) => LIST ; and the type of lists

(TYPE (LAMBDA (X) 1)) => EXPR

(TYPE '(LAMBDA (X) 1)) => LIST
```

Also see NUMTYPE and TYPENUM in the Advanced Section.

(EQ <object1> <object2>) SUBR

If the <object> are not numbers then EQ tests <object1> and <object2> to see if they are the same object. Since atomic objects are stored uniquely, (EQ A A) is always T. If the <object>s are numbers then EQ returns T if they have the same value AND the same type. Sometimes two floating point numbers can have the same printed representation but not be EQ. This is due to conversion errors that occur when a binary mantissa is printed in decimal notation. Composite objects satisfy EQ if <object1> and <object2> are references to the same object.

```
(EQ 1 1.0) => NIL

(EQ 'A 'A) => T

(EQ 'A 'B) => NIL

(EQ "AB" "AB") => NIL

(EQ [A B] [A B]) => NIL
```

Note that

```
(EQ '(A B) '(A B)) => NIL
```

because these are different objects, but

```
(SETQ L '(A B)) (SETQ M L)

(EQ M L) => T
```

since M and L are references to the same object.

(EQUAL <object1> <object2>) SUBR

This is the general equality predicate in TLC-LISP, returning T just in the case that:

1. The objects are <sexpr>s, and <object1> and <object2> have the same tree-structure, and the tip-nodes of those trees contain identical atoms in identical places.

2. The objects are vectors and all components are equal

3. The objects are strings, and they are identical in each character position.

The definition of EQUAL for S-exprs is equivalent to:

```
(DE EQUAL (X Y)
  (OR (EQ X Y)
      (AND (EQUAL (CAR X) (CAR Y))
           (EQUAL (CDR X) (CDR Y)))))
```

```
(EQUAL 'A 'A) => T

(EQUAL '(A B) '(A B)) => T

(EQUAL "ABC" "ABC") => T

(EQUAL [A B "AB"] [A B "AB"]) => T
```

Arithmetic

TLC-LISP supports both fixed point and floating point arithmetic. The arithmetic functions use the convention that if any argument is a floating point number, then the result will be floating point number.

The representation for floating point numbers is in accordance with IEEE K-C-S single precision floating point standard. Floating point numbers may range from 1.2E-38 to 3.4E38 (positive or negative) with 24 bit mantissa precision. Accuracy is within one least significant bit for arithmetic functions and within the two least significant bits for transcendental functions (except for tangent near its discontinuous points and logarithms near 1)

Overflow, underflow and invalid operations are detected and generate the appropriate error.

8087 Support

(FLOAT87 &OPT <flag>) SUBR

If <flag> is absent then the current FLOAT87 state is returned. If <flag> is NIL then the floating point routines will use software to compute floating point functions. If <flag> is non-NIL then 8087 instructions are used to compute floating point functions. The default is equivalent to (FLOAT87 NIL).

Arithmetic Functions

(ADD1 <num>) SUBR

Returns <num> plus 1.

```
(ADD1 4) => 5

(ADD1 -1) => 0
```

(SUB1 <num>) SUBR

Returns <num> minus 1.

```
(SUB1 4) => 3

(SUB1 0) => -1
```

(ABS <num>) SUBR

Returns the absolute value of <num>.

```
(ABS -1) => 1

(ABS 3.4) => 3.4
```

(ADD {<num>}) LSUBR

Return the sum of the arguments.

```
(ADD 3 4) => 7

(ADD 1.2 4 4) => 9.2

(ADD) => 0
```

(SUB {<num>}) LSUBR

With one argument, this function returns the number's negation. With more than one argument, it returns the first argument minus the rest of the arguments.

```
(SUB 4) => -4

(SUB 1 2) => -1

(SUB 1 2 3) => -4
```

(MUL {<num>}) LSUBR

Returns the product of the arguments.

```
(MUL 2.0 3 4) => 24.0

(MUL 2 (ADD1 5)) => 12
```

(DIV {<num>}) LSUBR

DIV returns its first argument successively divided by the rest of its arguments. If only one argument is given, the reciprocal is returned.

(DIV 4.0 2) => 2.0

(DIV 4 2) => 2

(DIV 5.0) => 0.2

(DIV 24 4 6) => 1

(REM <num1> <num2>) SUBR

Form the remainder upon division of <num1> by <num2>.

(REM -5 2) => -1

(REM 64 8) => 0

Two type conversion functions are provided:

(FIX <num>) SUBR

(FLOAT <num>) SUBR

(FLOAT 4) => 4.0

(FIX (ADD1 7.4)) => 8

A collection of arithmetic predicates is also included in TLC-LISP. These predicates return NIL if the test fails, and return a non-NIL value otherwise.

(ZEROP <num>) SUBR

Returns NIL if <num> is non-zero otherwise returns <num>.

(GE <num1> <num2>) SUBR

(GT <num1> <num2>) SUBR

(LE <num1> <num2>) SUBR

(LT <num1> <num2>) SUBR

Returns <num1> if the comparison is true, returns NIL otherwise. These functions also work for strings.

```
(GE 1 2) => NIL

(LT 2.2 300) => 2.2
```

(MINUSP <num>) SUBR

Returns <num> if <num> is a negative number otherwise returns NIL.

```
(MINUSP -1) => -1

(MINUSP 1.1) => NIL
```

A collection of trigonometric and transcendental functions is provided. Each of these functions returns a floating point number.

(RAD <num>) SUBR

<Num> is converted from degrees to radians.

(DEG <num>) SUBR

<Num> is converted from radians to degrees.

(LN <num>) SUBR

The natural (base e) logarithm.

```
(LN 2) => 0.6931472

(LN 1.1) => 9.531022e-02

(LN -1) => error    ; invalid floating point operation
```

(EXP <num>) SUBR

The transcental number e to the power <num>.

```
(EXP 1) => 2.718282

(EXP -1.1) => 0.3328711
```

(LOG10 <num>) SUBR

The common (base 10) logarithm.

```
(LOG10 100) => 2.

(LOG10 -1.1) => error    ; invalid floating point operation

(LOG10 0) => error       ; floating point overflow
```

(SIN <num>) SUBR

The sine of the angle <num> in radians.

```
(SIN PI) => 0.

(SIN (RAD 90)) => 1.
```

(COS <num>) SUBR

The cosine of the angle <num> in radians.

```
(COS PI) => -1.

(COS (RAD 90)) => 0.
```

(TAN <num>) SUBR

The tangent of the angle <num> in radians.

```
(TAN (RAD 45)) => 1.

(TAN 0.5) => 0.5463024
```

(ATAN <num>) SUBR

The angle in radians whose tangent is <num>.

```
(ATAN 0.5) => 0.4636476

(ATAN 0) => 0.
```

(X2Y <numx> <numy>) SUBR

<numx> raised to the power <numy>. <numy> is first converted to an integer if necessary.

```
(X2Y 2 10) => 1024.
```

(RN <num>) SUBR

Create a random number between 0 and <num>. If <num> is a floating-point number, it must be convertible into the range of fixed-point numbers.

```
(RN 33)  => 26

(RN 34.5) => 22
```

Boolean Functions

The functions in this section perform bit-wise logical operations. Their <num> parameters are restricted to be <integer> or <fix> quantities.

(LOGAND <num1> <num2>) SUBR

Perform the logical 'and' between <num1> and <num2>

(LOGAND 6 5) => 4

(LOGOR <num1> <num2>) SUBR

Perform the inclusive or between <num1> and <num2>

(LOGOR 6 5) => 7

(LOGXOR <num1> <num2>) SUBR

LOGXOR gives the exclusive or between <num1> and <num2>.

(LOGXOR 6 5) => 3

(LOGNOT <num>) SUBR

Form the complement of <num>.

(LOGNOT -1) => 0

Lists and Dotted Pairs

Selector Functions for Dotted Pairs

As the name suggests, selector functions select components. It is good style to preface a selection operation with an appropriate type test, assuring that the object meets the requirements of the selector. Such tests are built into TLC-LISP for the primitive data types -- for example CAR and CDR of strings is disallowed-- however, consistent with LISP's open nature, it is generally the programmer's responsibility to control the tool.

(CAR <sexpr>) SUBR

This function selects the first component of the dotted pair represented in <sexpr>.

```
(CAR NIL) => NIL

(CAR '(A . B)) => A

(CAR '(A B)) => A
```

Although the representation of (A B) is (A . (B . NIL)), it is better style to use the list selector FIRST when manipulating lists.

(CDR <sexpr>) SUBR

This function selects the second component of the dotted pair represented in <sexpr>.

```
(CDR NIL) => NIL

(CDR '(A . (B . C))) => (B . C)

(CDR '(A)) => NIL
```

(C...R <sexpr>) SUBR

These (twelve) functions give the usual CAR-CDR chains of LISP selection operations. Thus (CADDR <sexpr>) means the cAr of the cDr of the cDr of <sexpr>.

```
     (CADR '((1 . 2) . (3 . 4))) => 3    ; CAR of the CDR

     (CDAR '((1 . 2) . (3 . 4))) => 2    ; CDR of the CAR

     (CDDR '((1 . 2) . (3 . 4))) => 4    ; CDR of the CDR

     (CAAR '((1 . 2) . (3 . 4))) => 1    ; CAR of the CAR

     (CAAR NIL) => NIL                   ; NIL is special
```

and so on. Note that:

```
     (CDAR '(NIL . 3)) => NIL
```

because it is equivalent to

```
     (CDR (CAR '(NIL . 3))) => (CDR NIL) => NIL
```

but that:

```
     (CDAR '(1 . 3)) => error
```

because it is equivalent to:

```
     (CDR (CAR '(1 . 2))) => (CDR 1) => error
```

Selector Functions for Lists

To help reinforce the conceptual distinction between dotted pairs and lists, we have included selector functions that are supposed to be applied only to lists.

As a programming convenience, selector functions applied to NIL will return NIL. Also selecting the <num>-th element of a list of length less than <num> will return NIL.

(FIRST <list>) SUBR

(SECOND <list>) SUBR

(THIRD <list>) SUBR

(FOURTH <list>) SUBR

(FIFTH <list>) SUBR

(SIXTH <list>) SUBR

These functions select the appropriate element from <list>.

```
(FIRST '(1 2 3)) => 1

(FIFTH '(A B C D E F)) => E

(SECOND NIL) => NIL

(FOURTH '(1 2)) => NIL
```

(REST <list> &OPT ((<num> 1)) SUBR

<List> is a non-empty list. <Num> is greater than or equal to one. REST returns the remainder of <list> after the <num>-th element. If <num> is greater than or equal to the length of <list> then NIL is returned.

```
(REST '(A B C D)) => (B C D)

(REST '(A B C D) 3) => (D)

(REST '(A)) => NIL

(REST NIL) => NIL        ; Note

(REST '(1 2 3) 8) => NIL
```

(NTH <list> <num>) SUBR

NTH returns <num>-th element of <list>. If there are less than <num> elements in the list then NIL is returned. NTH also works on strings and vectors.

```
(NTH '(A B C D) 2) => B

(NTH '(A B C) 9) => NIL

(NTH NIL 2) => NIL

(NTH '(A B C) -1) => error
```

NTH for lists is equivalent to:

```
(de NTH (l n)
  (cond
    ((or (minusp n)(zerop n)) (error))
    ((eq n 1) (first l))
    (t (nth (rest l) (sub1 n)) ) ))
```

Constructors for Dotted Pairs

Besides being able to test the type of an object and select components of a composite structure, we must be able to construct new objects of specified types. The generic name for such a function is a constructor.

(CONS <object1> <object2>) SUBR

This constructor makes a new dotted pair whose CAR-branch is <object1> and whose CDR-branch is <object2>.

```
     (CONS 'A 'B) => (A . B)

     (CONS "A" '(A .  B)) => ("A" A . B) which is ("A" . (A . B))


     (CONS [1 2 3 4] 1) => ([1 2 3 4] . 1)
```

Recall that the printer attempts to print any <object> in list notation. Thus (A . (B . NIL)) prints as (A B).

```
     (CONS (ATOM 'A) (ATOM '(A))) => (T) which is (T . NIL)
```

(SUBST <object1> <object2> <sexpr>) SUBR

SUBST substitutes <object1> for every occurrence of <object2> in a copy of <sexpr>.

```
     (SUBST 'C 'A '((1 . A) (A B) C)) => ((1 . C) (C B) C)
```

SUBST uses EQUAL internally, thus:

```
     (SUBST "def" "abc"  '(A B "abc")) => (A B "def")
```

even though the two strings "abc" are not EQ. SUBST is equivalent to:

```
     (de SUBST (x y z)
       (if (atom z)
           (if (equal y z)
               x
               z )
           (cons (subst x y (car z))
                 (subst x y (cdr z)))))
```

(COPY <sexpr>) SUBR

This function returns a copy of <sexpr>.

```
(EQ  X  (COPY X))  =>  T    if X is a number or symbol
                   =>  NIL  otherwise
```

but (EQUAL X (COPY X)) => T, always.

```
(COPY '((A . B) . C)) => ((A . B) . C)

(SETQ L '(A B C))
(EQ L L) => T              ; identical
(EQ L (COPY L)) => NIL     ; not identical
(EQUAL L (COPY L)) => T    ; but of identical form

(EQ 'A (COPY 'A)) => T     ; atomic
```

COPY for lists is equivalent to:

```
(de COPY (l)
  (if (atom l)
      l
      (cons (copy (car l))
            (copy (cdr l)) )))
```

COPY also works for other types of objects. The COPY of a vector is a new vector that shares all the components of the original vector. The COPY of a string is a new descriptor (see Part I of the documentation) that shares the characters. See the Sections on constructors of vectors and strings for details.

Constructors for Lists

(CONCAT <object> <list>) SUBR

This constructor expects a list in its second argument position. It makes a new list object with <object> as its FIRST element, and has <list> as its REST-component. In terms of the traditional implementation of LISP, CONCAT and CONS are equivalent. However, good programming style dictates that one use CONS when constructing dotted pairs and use CONCAT when constructing lists.

```
(CONCAT 'A '(S D F)) => (A S D F)

(CONCAT 'A NIL) => (A)

(CONCAT '(A S D) '(A B C)) => ((A S D) A B C)
```

(LIST {<object>}) LSUBR

This constructor makes a list out of its arguments.

```
(LIST (CONS 1 2) (CAR '(A . B)) (REST '(A B)))
  => ((1 . 2) A (B))
```

(APPEND <list1> <list2>) SUBR

Creates a new list whose initial segment consists of the elements of <list1> and whose final segment is the list <list2>. APPEND will copy the elements of <list1>. Thus

(APPEND <list> NIL) has the effect of copying <list>.

```
(DE APPEND (L1 L2)
  (IF (NULL L1)
      L2
      (CONCAT (FIRST L1) (APPEND (REST L1) L2))))

(APPEND '(1 2 3) (REST '(A B C))) => (1 2 3 B C)
```

(REVERSE <list>) SUBR

REVERSE makes a new list whose elements are the elements of <list> in reverse order:

```
(DE REVERSE (L)
 (REV1 L NIL) )

(DE REV1 (L1 L2)
  (IF (NULL L1)
      L2
      (REV1 (REST L1) (CONCAT (FIRST L1) L2)) ))

 (REVERSE '(A B C D E)) => (E D C B A)
```

List and Dotted Pair Modifiers

The LISP functions of the preceding section perform their computations by constructing new objects. The functions of this section allow the programmer to modify existing objects. These operations are powerful and therefore must be used with great care.

For example, the list-modifying operations can create circular list-structure, which can cause difficulty for a simple list-printer.

A more subtle difficulty can arise in the "alias problem" wherein lists, strings and vectors that share objects, can all be effected when one of the shared components is modified.

Modifiers are not always to be thought of as pernicious predators, though. In particular, vector objects are historically manipulated by such modification operations.

(RPLACA <sexpr> <object>) SUBR

RPLACA, from 'RePLAce the CAr of', expects <sexpr> to be a dotted-pair or a non-empty list. It replaces the CAR part of <sexpr> with <object>. The value returned is the modified <sexpr>.

```
(RPLACA  '(A B) 'C)  =>  (C B)

(SETQ X '(A B)) => (A B)
(SETQ  Y X) => (A B)
(RPLACA X 'C) => (C B)
```

Now X => (C B) as expected, but note also Y => (C B) which may not have been anticipated.

(RPLACD <sexpr> <object>) SUBR

Replaces the CDR-part of <sexpr> with <object>. As with RPLACA, RPLACD expects <sexpr> to be a dotted pair or non-empty list.

```
     (RPLACD '(A . B) 'C) => (A . C)

     (RPLACD '(A B C) 1) =>  (A . 1)
```

since (A B C) is represented as (A . (B . (C . NIL)))

(RPLACB <sexpr1> <sexpr2>) SUBR

Replaces the CAR-part of <sexpr1> with the CAR-part of <sexpr2>, and the CDR-part of <sexpr1> is replaced with the CDR-part of <sexpr2>. <sexpr1> and <sexpr2> must both be non-empty lists or dotted pairs.

```
     (DE RPLACB (X Y)
       (RPLACA X (CAR Y))
       (RPLACD X (CDR Y)) )
```

(NCONC <list1> <list2>) SUBR

This function has an effect similar to that of APPEND, except NCONC does not copy its first argument; rather, it replaces the NIL which terminates the list <list1> with <list2>. The value returned by NCONC is the value of the modified list.

```
     (DE NCONC (L1 L2)
       (IF (NULL L1)
           L2
           (IF (REST L1)
               ; then
               (NCONC (REST L1) L2)
               ; else
               (RPLACD L1 L2)
               L1 )))

(NCONC '(A B C) '(D E F)) => (A B C D E F)

     (SETQ X '(A B C)) => (A B C)
     (SETQ Y '(D E F)) => (D E F)
     (NCONC X Y) => (A B C D E F)
```

and Y => (D E F), but beware, X => (A B C D E F)

(FREVERSE <list>) SUBR

This is a destructive version of REVERSE, using no CONSes.

```
(DE FREVERSE (L1 &OPTIONAL (L2 ()))
  (IF (NULL L1)
       L2
       (FREVERSE (REST L1) (RPLACD L1 L2)) ))
```

again, application of FREVERSE must be done carefully:

```
(SETQ X '(A B C))  => (A B C)
(SETQ Y (REST X))  => (B C)
```

now (FREVERSE Y) => (C B) and Y => (B), but X => (A B)

General List Functions

As we mentioned in the Control section, all LISP functions can be used as predicates; TLC-LISP (and most other LISP implementations) map non-NIL and NIL to true and false, respectively. This is more than 'just a programming trick'; it is a very useful programming technique. For example, we often need to compute an expression like 'find the first element which satisfies a condition, if one exists'. Instead of using a predicate to test for existence, followed by a selection function to extract the value if one exists, we use a 'pseudo predicate' which will return NIL (false) if none is found, but will return some representation of the element (testable as 'true') if one is found. In fact, since the search usually involves the traversal of a list, it is good practice to return the list-segment whose first element satisfies the test; then, if that element fails to satisfy other criteria, we can continue the search with the remainder of the list. A good example of this programming style is ASSOC.

(ASSOC <atom> ({(<atomi> . <objecti>)})) SUBR

ASSOC searches the list ({<atomi> . <objecti>}) for a match of <atom>. If one is found, the remainder of the list beginning with the pair containing the match is returned. If no match is found, NIL is the value of the ASSOC. (see the note after MEMQ).

```
(DE ASSOC (X L)
  (COND
    ( (NULL L) NIL)
    ( (EQ X (CAR (FIRST L))) L)
    ( T (ASSOC X (REST L))) ))

(ASSOC 'TLC '((FOO . LOSE) (TLC . WIN) (NERD . LOSE)))
  => ((TLC . WIN) (NERD . LOSE))
```

(MEMQ <atom1> ({<atom2>})) SUBR

MEMQ is another 'pseudo predicate', returning either NIL if the first argument, <atom1>, is not found in the list ({<atom2>}). MEMQ returns the remainder of the list beginning at the match if a match is found. MEMQ's definition follows:

```
(DE MEMQ (A L)
  (IF (OR (NULL L) (EQ (FIRST L) A))
      L
      (MEMQ A (REST L))))
```

```
(MEMQ 'A '(1 2 3 A B C)) => (A B C)
```

Note: Though ASSOC and MEMQ are defined in terms of <atom>s, they may be applied with expressions in those positions. Both functions use EQ. Care must be exercised since (EQ '(A) '(A)) will give NIL; if you don't understand this, don't use expressions in the <atom> positions.

(LENGTH <object>) SUBR

Returns the length of the <object>, which may be a string, vector, or list.

```
(LENGTH '(1 2 3 4)) => 4

(LENGTH '("xx" [1 2] 4)) => 3

(LENGTH NIL) => 0
```

LENGTH for lists could be defined as:

```
(de LENGTH (l &OPT (n 0) )
  (if (null l)
      n
      (length (rest l) (add1 n))))
```

or as:

```
(de LENGTH (l)
  (do ( (n 0 (add1 n))
        (l l (rest l)) )
      ( ((null l) n) ) ))
```

Strings and Characters

Strings are ordered collections of eight bit characters. A string may be 63520 characters long. Strings print with surrounding double quotes (") and literal strings may be entered directly by enclosing them in double quotes, i.e "A string".

As a convenience for editor programs, the character ^Z (ASCII 26) will be treated as the end of the string by the printer. Thus the string created by (STRING "abc" ^Z "def") will print as "abc".

Characters will be treated by many functions as if they were strings of length 1. Thus (LENGTH \a) => 1, (NTH \a 1) => \a.

Selector Functions for Strings

Though strings can be thought of, indeed implemented as lists of characters, there are some inherent distinctions between the data types, string and list. These distinctions are reinforced in the actions of the string selector function.

(SUBSTRING <str> &OPT <num1> <num2>) SUBR

Creates a new string EQUAL to the substring of <str> beginning with the <num1>-th character and containing the next <num2>-th characters. If <num1> and <num2> are missing, all of <str> is returned; if <num2> is missing it defaults to the length required to specify the tail of the string.

```
(SUBSTRING "abcdef" 4) => "def"

(SUBSTRING "abcdefg" 4 2) => "de"

(SUBSTRING "abc" 2 0) => ""          ; the empty string

(SUBSTRING "abc" 0 2) => error       ; index zero is illegal
```

Note that:

```
(SUBSTRING "abc" 4 0) => error
```

even though the length requested is zero, the position specified is illegal.

Note that the string returned by SUBSTRING actually shares all or part of the body of <str>. Thus altering the body of one string will have the side effect of altering the body of the other string. If you want a copy then use (STRING (SUBSTRING ...)).

```
     (SETQ S1 "abcdefghi")

     (SETQ S2 (SUBSTRING S1 3 4)) => "cdef"

     (STRING-REPLACE-CHAR S1 5 \-) => "abcd-fghi"
```

but now:

```
     S2 => "cd-f"
```

Although the body of a string and its substring can be at the same memory location the string descriptors are not the same, thus:

```
     (SETQ S1 "abc" S2 (SUBSTRING S1))
     (EQUAL S1 S2) => T
     (EQ S1 S2) => NIL
     (EQ (POINTER S1) (POINTER S2)) => T
```

(NTH <str> <num>) SUBR

Selects the <num>-th character from <str>. NTH also works on lists and vectors.

```
     (NTH "abc" 2) => \b

     (NTH "abc" 0) => error   ; index zero is illegal

     (NTH "abc" 4) => error   ; index out of range
```

(STRING-SEARCH <str> <char or string> &OPT ((<num> 1)) SUBR

Return the index of the first occurrence of <char or str> in <str>, beginning the search at index <num>. If the <char or str> does not occur in <str> then return NIL.

```
     (STRING-SEARCH "abcdefg" "cd") => 3

     (STRING-SEARCH "abcdefg" "ABC") => NIL

     (STRING-SEARCH "abcdefg" \b) => 2

     (STRING-SEARCH "abcabcabc" \b 6) => 8

     (STRING-SEARCH "abc" "ab" 6) => error   ; index out of range
```

Constructors for Strings

(STRING {<str> or <char>}) LSUBR

STRING takes an arbitrary number of strings and characters as arguments and builds a new string.

```
(STRING "ABC" \D \E) => "ABCDE"

(STRING "AB" (SUBSTRING "ABCDEF" 4)) => "ABDEF"
```

(COPY <str>) SUBR

COPY is equivalent to (STRING <str>). It also works on lists and vectors.

(NEWSTRING <num> {<form>}) FSUBR

Returns a string of length <num> whose characters are the result of evaluating <form>. <Num> is evaluated once. <Form> is evaluated once for each character of the new string. The evaluation of <form> must return a character or an error is generated.

```
(NEWSTRING 5 \a) => "aaaaa"

(SETQ X (ASCII \a))
(NEWSTRING 5 (ASCII (SETQ X (ADD1 X)))) => "bcdef"

(NEWSTRING 10 0) => error     ; char expected
```

String Modifiers

The following functions alter their string argument as opposed to manufacturing a copy. They should be used with care. If you do not desire this side effect on the original string, then use (COPY <str>).

(STRING-INSERT-CHAR <str> <num> <char>) SUBR

Insert <char> at position <num> in <str>, moving the character currently at position <num> (and those following) one position towards the end of the string. Returns the character that falls off the end. Note that this function operates on <str> and not on a copy of <str>.

```
(SETQ FOO "abcdef")
(STRING-INSERT-CHAR FOO 3 \Q) => \f
FOO => "abQcde"

(STRING-INSERT-CHAR "abc" 4 \Q) => error      ; index out
                                              ; of range
```

(STRING-DELETE-CHAR <str> <num> <char>) SUBR

Delete the character at position <num> of <str>, moving the characters following it towards the beginning of the string. Insert <char> at the end of the string. Return the character deleted. Note that this function operates on <str> and not on a copy of <str>.

```
(SETQ FOO "abcdef")
(STRING-DELETE-CHAR FOO 3 \Q) => \c
FOO => "abdefQ"

(STRING-DELETE-CHAR "abc" 4 \Q) => error      ; index out
                                              ; of range
```

(STRING-REPLACE-CHAR <str> <num> <char>) SUBR

The <num>-th position of <str> is set to <char>.

Replace the selected character with the new character. This function operates directly on <str>, not a copy of it.

```
(STRING-REPLACE-CHAR "abc" 2 \B) => "aBc"
```

(UPPER <str or char>) SUBR

For characters, UPPER returns an uppercase version of the character. For strings, converts all lowercase letters to uppercase. Note that UPPER operates on the string argument itself and not on a copy.

```
    (SETQ S "abc123")
    (UPPER S) => "ABC123"
    S => "ABC123"

    (UPPER "123") => "123"

    (UPPER "") => ""
```

(LOWER <str or char>) SUBR

For characters returns an lowercase version of the character. For strings, converts all uppercase letters to lowercase. Note that LOWER operates on the string argument itself and not on a copy. If you do not desire this side effect then use (LOWER (COPY <str>)).

```
    (SETQ S "ABC123")
    (LOWER S) => "abc123"
    S => "abc123"

    (LOWER "123") => "123"

    (LOWER "") => ""
```

Miscellaneous String and Character Functions

(ASCII <arg>) SUBR

If <arg> is number, ASCII returns the character whose ascii code is that number. If <arg> is a character or a single-character string, then the ascii code for that character is returned. Any other type of argument is an error.

```
    (ASCII \C) => 67

    (ASCII 67) => \C

    (ASCII 'A) => error      ; number expected

    (ASCII "a") => 97
```

(GT <str1> <str2>) SUBR

(GE <str1> <str2>) SUBR

(LT <str1> <str2>) SUBR

(LE <str1> <str2>) SUBR

These functions allow lexicographical comparison of the two strings, returning <str1> if the comparison is true, NIL if the comparison is false. This quartet also works on numbers.

```
(GT "AB" "A") => "AB"

(LT "A" "B") => "A"

(LT "a" "B") => NIL      since all lowercase letters are
                         greater than all uppercase ones.
```

(HASH <str>) SUBR

Returns a number based on characters of <str>. Identical to the hash function used by the system when INSERTing atoms into packages.

```
(HASH "abcd") => 19380
```

(LENGTH <str>) SUBR

Returns the number of characters in <str>. LENGTH also works on lists and vectors.

```
(LENGTH "ABCD") => 4

(LENGTH (SUBSTRING "abcdef" 4)) => 3

(LENGTH "") => 0
```

Vectors

Vectors may be thought of as finite functions. Therefore, for a vector V, we may refer to the I-th component by (V I). The identical result is returned by (VREF V I). Literal vectors are designated by surrounding the elements with square brackets, as in:

```
[1 2 3 4]
```

The elements in a vector may be arbitrary LISP objects, including vectors themselves. In this way we can generate arbitrary (even "ragged") arrays.

Selector Functions for Vectors

(VREF <vector> <num>) SUBR

Returns the <num>-th element of <vector>.

```
(VREF [a b c] 2) => b

(VREF [a b c] 0) => error     ; index zero illegal

(VREF [a b c] 4) => error     ; index out of range
```

Constructors for Vectors

(VECTOR {<object> or <vector>}) LSUBR

VECTOR is similar in function to STRING. Given <object>s, VECTOR will make them elements of a new vector. Given <vector>s, VECTOR will "flatten" them into elements for the new vector.

```
(VECTOR 1 2 (ADD1 3)) => [1 2 4]

(VECTOR 1 2 [3 4]) => [1 2 3 4]
```

arguments that are vectors will only be "flattened" one level, thus:

```
(VECTOR [ [1 2] 3] 4 5) => [ [1 2] 3 4 5]
```

The first argument [[1 2] 3] is a vector of two elements and thus contributes two elements to the new vector.

(COPY <vector>) SUBR

Is equivalent to (VECTOR <vector>). COPY also works for strings and lists.

COPY builds a new "receptacle" for the vector elements, but shares rather than copies those elements. Thus:

```
(SETQ XX [1 2 (A B) 3]) => [1 2 (A B) 3]

(SETQ YY (COPY XX)) => [1 2 (A B) 3]

(RPLACA (VREF YY 3) 2) => (2 B)

YY => [1 2 (2 B) 3]

XX => [1 2 (2 B) 3]
```

(NEWVECTOR <num> {<form>}) FSUBR

NEWVECTOR will construct a new vector of <num> elements. <num> is evaluated once. Each element will be the result of consecutive evaluations of {<form>}.

```
(NEWVECTOR 3 NIL) => [NIL NIL NIL]

(NEWVECTOR 0 100) => [ ]

(SETQ X 2)
(NEWVECTOR 4 (SETQ X (ADD1 X))) => [3 4 5 6]
```

Vector Modifiers

These functions destructively modify their vector argument. They should be used with care.

(STORE <vector> <num> <object>) SUBR

Replace the <num>-th element of <vector> with <object>. STORE operates on its argument and not on a copy. It returns the modified vector.

```
(SETQ V [1 A 3])

(STORE V 2 [1 2 3]) => [1 [1 2 3] 3]

(STORE [1 2 3] 4 4) => error       ; index out of range
```

(VECTOR-DELETE-ELEMENT <vector> <num> &OPT (VAL nil)) SUBR

Modifies <vector> by deleting the element at position <num>, moving the remaining elements one position towards the beginning of the vector and setting the last element position to VAL. This function returns the deleted element. Note that this function operates on <vector> itself and not on a copy.

```
(SETQ V [a b c])

(VECTOR-DELETE-ELEMENT V 2) => b
V => [a c nil]

(VECTOR-DELETE-ELEMENT V 1 "foo") => a
V => [c nil "foo"]

(VECTOR-DELETE-ELEMENT V 4) => error    ; index out of range
```

(VECTOR-INSERT-ELEMENT <vector> <num> <obj>) SUBR

Modifies <vector> by inserting <obj> at position <num>, moving the remaining elements one position toward the end of the vector and discarding the last element. It returns the discarded element. Note that this function operates on <vector> itself and not on a copy.

```
(SETQ V [a b c])

(VECTOR-INSERT-ELEMENT V 2 nil) => c
V => [a nil b]

(VECTOR-INSERT-ELEMENT V 1 "foo") => b
V => ["foo" a nil]

(VECTOR-INSERT-ELEMENT V 4 "a") => error    ; index out of range
```

General Vector Functions

(MEMVEC <object> <vector>) SUBR

If <object> is EQUAL to an element in <vector>, MEMVEC returns the first such index, otherwise it returns NIL.

```
(MEMVEC (A B) [1 2 (A B) 3  T (A B)]  =>  3
```

(LENGTH <vector>) SUBR

Returns the number of elements in the vector. LENGTH also works on lists and strings.

```
(LENGTH [ ]) => 0

(LENGTH [1 2 3]) => 3

(LENGTH (NEWVECTOR 10 0)) => 10
```

Environments and State Modifiers

State Modifiers

Except for the function-defining functions DE, DF, and DM, the bindings of variables to values has been a 'non-destructive' kind in the sense that when we leave the context of a LAMBDA expression (or LET or DO) the previous bindings of local variables are restored. The next functions involve 'destructive' assignment to variables, they are LISP's formulation of the assignment statement. As with all LISP forms, they are expressions (rather than a statement) and therefore return a value.

(SETQ {<var> <form>}) FSUBR

Each <var> is bound to the value of its corresponding <form>. The evaluation proceeds sequentially, from left to right (rather than in parallel as in the DO-expression). The value of the SETQ is the value of the last <form>.

```
(SETQ X 4 Y 'A) => A
X => 4
Y => A

(SETQ X 6 Y (CONS X Y)) => (6 . A)  ; not (4 . A)
X => 6
Y => (6 . A)
```

(SET <var> <object>) SUBR

The symbol <var> is assigned <object> as value.

(SET (QUOTE X) <form>) is the same as (SETQ X <form>)

```
(SET 'X '(A B)) => (A B)
X => (A B)
(SET (FIRST X) (CONS X 1)) => ((A B) .  1)
A => ((A B) . 1)
X => (A B)
```

Most common applications involve SETQ, not SET.

(UNBIND <var>) SUBR

This function sets <var> to the distinguished state UNBOUND. Subsequent attempts to evaluate <var> (before it is bound via SET, SETQ or as a formal parameter or local variable during function application) will result in an 'unbound-atom' error.

```
(SETQ A 1) => 1
A => 1
(UNBIND 'A) => A
A => error           ; unbound atom
```

Environment Objects

In addition to modifying the global state, we supply an object to contain local state -- the environment object. Environment objects may be used as arguments to CLOSURE to allow several different functions to share the same state. The environment constructor is ENV.

(ENV {{<var> <object>} or <env>}) LSUBR

ENV returns an environment object with the indicated variables and values. If an argument in the <var> position is an environment instead of a <var>, then that environment is merged into the new environment. Thus:

```
(SETQ E1 (ENV 'A 1 'B 2))
(SETQ E2 (ENV E1 'C 3)) => (ENV 'A 1 'B 2 'C 3)
```

Environments may also be used as functions, interpreting them as finite functions.

```
((ENV 'A 10 'B 20) 'A) => 10
```

Environments may also be updated by supplying two arguments, a variable name and an object. The effect is to update the environment.

```
((ENV 'A 10 'B 20) 'A 11) => (ENV 'A 11 'B 20)
```

Property Lists

LISP property lists are a powerful tool for constructing data bases. A property list consists of a set of attribute-value (or indicator-property) pairs. In TLC-LISP a property list is only associated with a symbol. Therefore a natural model for such property lists is the interpretation of the symbol as a 'dictionary entry' and the attribute-value pairs as that entry's different possible meanings. See the section on "Property Lists" in Part I and compare the features of Property Lists with those of Environments.

(PUTPROP <var> <atom> <object>) SUBR

The <object> is placed on the property list of <var> under the attribute <atom>. Any previous value associated with <atom> is destroyed. The value returned is <object>.

```
(PUTPROP 'WALDO 'AGE 47) => 47
```

(GETPROP <var> <atom>) SUBR

The property list of <var> is searched for the indicator <atom>; if found, the corresponding value entry is returned. If no match is found NIL is returned. Care must be exercised to distinguish between a 'false' indication and the return of a value NIL. Continuing the previous example:

```
(GETPROP 'WALDO 'AGE) => 47

(PUTPROP 'WALDO 'CHILDREN NIL) => NIL

(GETPROP 'WALDO 'MARRIED)  => NIL

(GETPROP 'WALDO 'CHILDREN) => NIL
```

(REMPROP <var> <atom>) SUBR

This function removes from the p-list of <var>, the latest attribute-value pair with attribute <atom>; if none existed, NIL is returned otherwise the value of REMPROP is the removed value. If ADDPROP has been used with indicator <atom>, the next-latest <atom>-<object> pair is made current.

```
(REMPROP 'WALDO 'AGE) => 47

(GETPROP 'WALDO 'AGE) => NIL
```

(ADDPROP <var> <atom> <object>) SUBR

Similar to PUTPROP, except a previous value associated with the attribute <atom> is saved.

```
(PUTPROP 'WALDO 'CHILDREN '(LOUIE SAM)) => (LOUIE SAM)

(ADDPROP 'WALDO 'CHILDREN '(NERD)) => (NERD)

(GETPROP 'WALDO 'CHILDREN) => (NERD)

(REMPROP 'WALDO 'CHILDREN) => (NERD)

(GETPROP 'WALDO 'CHILDREN) => (LOUIE SAM)
```

(PLIST <var>) SUBR

PLIST returns a representation of the property-list associated with <var>. Note that the actual p-list is not returned.

```
(PLIST 'WALDO) => (CHILDREN (LOUIE SAM))

(PUTPROP 'WALDO 'FOO '7) => 7

(PLIST 'WALDO) => (FOO 7 CHILDREN (LOUIE SAM))
```

Symbols and Packages

About Packages

Packages are collections of symbols. Packages are arranged in a tree, each package except the root package having a superpackage. The root package is named SYS:. The superpackage of the root package is NIL. One package is always the "current package". The current package is defined as the value of the symbol PACKAGE. The initial current package is SYS:. Executing (PKG "foo" SYS:), (PKG "bar" SYS:) and (PKG "baz" FOO:) results in the creation of the following package hierarchy:

```
      SYS:
        :
 +------+---+
 :          :
 :        BAR:
FOO:
 :
 :
BAZ:
```

Packages are active at read time. The reader recognizes a symbol as a non-numeric "normal" character followed by zero or more "normal" characters followed by a delimiter (see the section on <u>Input and Output</u> about "normal" characters). For the purposes of this section, a delimiter is any non-normal character, thus comment characters, parenthesis, etc. are all delimiters for symbols and package names. A, A100 and ABCDEF are valid symbols. When a symbol is scanned, the reader searches the current package for a symbol with the same print name. If a matching symbol is found then the reader returns the symbol as the value of the read. If no matching symbol can be found then the superpackage is searched. This process continues until a matching symbol is found or until the root package is searched. If a match was found it is returned. If no match was found in the current package or any super package then a symbol is created in the current package and that symbol is returned.

The reader also recognizes package names. A package name looks the same as a symbol except it ends in a colon character ":". Thus A:, A100: and ABCDEF: are valid package names. If the reader scans a package name followed by a delimiter then the package whose print name matches the name scanned is returned. Unlike symbols, packages are not automatically created by the reader. If a package name is scanned and no existing package's print name matches then a NO-SUCH-PKG error is generated. Print names for packages must be unique. There is no way to have two different packages with the same print name.

When a package name is followed immediately by a symbol instead of a delimiter then the reader returns the matching symbol in the specified package, creating it if necessary. No superpackages are searched nor is the current package (unless it happens to be the same as the specified package). Thus A:FOO returns the symbol FOO in the package A:.

The reader also supports an abbreviation (:) for the root package SYS:. A colon preceded by a delimiter returns the SYS: package and the symbols :FOO, :BAR and :BAZ are all in the SYS: package.

Packages are useful for preventing name conflict. Name conflict occurs when two programs attempt to use the same symbol name for two different purposes. For example an editor program might have a function PUTCHAR to store a character into its buffer while a pretty printer program may have a function PUTCHAR to display characters on the screen. Loading both of these programs would result in one of them using the wrong PUTCHAR. This problem could be solved by loading each program into its own package. Assume the existence of packages PP: and EDIT: both with superpackage SYS:. We make PP: the current package and load the pretty printer. When the symbol PUTCHAR is scanned the reader searches PP: then SYS:, finds no matching symbol so it creates one in PP:. The next PUTCHAR scanned will return the symbol in PP:. We then make EDIT: the current package and load the editor program. When the symbol PUTCHAR is scanned, the reader searches EDIT: then SYS:, finds no match and creates the symbol PUTCHAR in EDIT:. PP: was not searched because it is not in EDIT:'s superpackage chain (not an ancestor of EDIT:). Thus a PUTCHAR symbol exists in two different packages and the conflict is resolved.

For the purposes of printing, the package prefix of a symbol is handled similar to string and character delimiters. If the package of a symbol is not the same as the current package then the functions PRIN0, PRIN1 and PRINT will print the symbol with its package prefix. If the current package is the same as that of the symbol, then no prefix is printed. The functions PRIN2 and PRIN3 never print the package prefix regardless of the value of the current package. See the section on Input and Output for more details.

A common error when using packages involves the attempt to reference a function by name before it has been defined (called a forward reference) and getting two symbols when only one was desired. For example, assume the following code is loaded into the package UTIL:

```
(de :FOO (n)
  (bar n)
  (baz n))

(de BAZ (n)
  (mumble n))

(de :BAR (n)
  (baz n))
```

Note that FOO and BAR are specified to be interned in the SYS: package. However a reference is made to BAR in the definition of FOO. Since no symbol is found in either UTIL: or SYS: when this first reference is scanned, the symbol BAR is (incorrectly) created in UTIL:. The subsequent definition of :BAR creates a second symbol in SYS:. Executing FOO results in an UNBOUND-ATOM UTIL:BAR error. Two solutions are possible. The order of definitions could be changed (to BAZ, BAR, FOO) so that there are no forward references or the reference to BAR in FOO could use a package prefix, SYS:BAR (or :BAR).

Automatic Removal of Symbols

The TLC-LISP garbage collector automatically removes any symbol that meets all the following criteria:

1. The symbol is not built-in (e.g. CONS)

2. The symbol has no value, i.e (BOUNDP <symbol>) is NIL.

3. The symbol is not referenced directly or indirectly by any other symbol or system data structure. The package that the symbol belongs to does not count as a reference.

Thus:
```
          (SETQ FOO '(BAZ BAR))
```

prevents BAZ and BAR from being removed because they are referenced by FOO, and prevents FOO from being removed because it has a value, the list (BAZ BAR). Sometimes a symbol that looks like it should be removable will persist because it is referenced by the system through the run-time stack. (RETFRAME) will remove these references.

Package Functions

(INSERT <str> &OPT <pkg>) SUBR

Find a symbol with print name <str> and return that symbol as value or, if no such symbol exists, construct a new symbol with that print name and return it. If <pkg> is not supplied then the package hierarchy starting with the current package (PACKAGE) is searched. If <pkg> is supplied then only <pkg> is searched. Assuming the existence of the empty packages FOO:, BAR: and BAZ:. FOO: and BAR: are sub-packages of SYS:, BAZ: is a sub-package of FOO:. Then:

```
     (SETQ PACKAGE BAZ:)       ; Set the current package
```

Note; when the current package is something other than SYS: then the system prompt will be prefaced with colon as in :>>>.

```
     (INSERT "temp") => TEMP  ; no prefix so in package BAZ:
     (INSERT "temp" SYS:) => :temp
     (INSERT "temp" BAR:) => bar:temp
     (SYMBOL-PKG 'TEMP) => baz:
     (SETQ PACKAGE FOO:)
     'TEMP => :TEMP                  ; searches FOO:, then SYS:
```

(LOOKUP <str> &OPTIONAL <pkg>) SUBR

Like INSERT, except returns NIL if the desired symbol is not in the symbol table; in this case a new symbol is not constructed. If <pkg> is not supplied then the package hierarchy starting with the current package (PACKAGE) is searched. If <pkg> is supplied then only <pkg> is searched.

```
     (LOOKUP "ABC") => NIL

     (INSERT "ABC") => ABC

     (LOOKUP "ABC") => ABC
```

Continuing the example from INSERT above:

```
     (LOOKUP "temp" FOO:) => NIL
     (LOOKUP "temp") => BAR:TEMP
```

(PNAME <symbol>) SUBR

Return a string that represents the print name of <symbol>. PNAME does not return the actual print name, but a copy. Note that the package prefix is not part of the print name.

```
(PNAME 'ABC) => "abc"

(PNAME 'FOO:BAZ) => "baz"
```

(OBLIST) SUBR

OBLIST (Object-list) returns a list of all the symbols in the current package. The list is manufactured each time OBLIST is invoked and may be destructively modified without risk.

(PF) SUBR

PF (Package family) returns a list of all existing packages. The list returned is the actual data structure used by the system and not a copy. Destructive modification of the list will corrupt the system.

```
(PF) => (sys: util: edit: pp:)
```

(PKG <str> &OPT (<pkg> PACKAGE) (<num> 128)) SUBR

If a package exists with print name <str> then that package is returned. Otherwise, PKG creates a package with print name <str>:, superpackage <pkg> and size <num>. <num> is the number of unique hash values for the symbols of the package and is not the number of symbols that may be inserted in the package.

```
(PKG "FOO" SYS:) => FOO:

(PKG "SYS" SYS:) => SYS:
```

(SYMBOL-PKG <symbol>) SUBR

Returns the first package in the ancestry of the current package that contains <symbol>.

```
(SYMBOL-PKG 'CONS) => sys:

(INSERT "temp" FOO:)
(INSERT "temp" BAR:)
(SYMBOL-PKG 'FOO:TEMP) => foo:
(SYMBOL-PKG 'BAR:TEMP) => bar:
```

(SUPER-PKG <pkg>) SUBR

Returns the superpackage of the specified package.

Continuing the example from the first part of this section,

```
(SUPER-PKG BAZ:)  =>  FOO:

(SUPER-PKG BAR:)  => SYS:

(SUPER-PKG (SUPER-PKG))  => SYS:

(SUPER-PKG SYS:)  => NIL
```

(LOOKUP-PKG <str>) SUBR

Gets the package whose name is <str> if one exists, or returns NIL if no such package exists.

```
(LOOKUP-PKG "SYS")  =>  SYS:
```

(GENSYM &OPTIONAL (<str> "G")) SUBR

Generates a "pseudo symbol" --an object that acts like a symbol, but is not installed on an oblist. Such created symbols are useful within macro creation or other program-manipulating applications. The string <str> is used as a prefix, and a counter is generated to be used as a suffix. Thus:

```
(GENSYM)  => G1

(GENSYM "Local-") => LOCAL-5
```

The Class System

The general notions of class systems were discussed in the Introduction (Part I), so this section will concentrate on the specific operations in TLC-LISP.

Constructors

Classes are created using the CLASS function, specifying
(1) a superclass,
(2) messages and their associated methods,
(3) class variables that will allow shared information between instances of a class, and finally
(4) instance variables that allow each instance some private information.

```
(CLASS <class>
        ({<var-i> <fcn-i>})
        ({<var-j> <object-j>})
        ({<var-k> <object-k>}) ) SUBR
```

creates a class object with superclass <class>, messages <var-i> and associated method functions <fcn-i>, class variables <var-j> and their initial values <object-j> and finally, instance variables <var-k> and their default values <object-k>. If no superclass is desired then NIL is used.

For an introductory example, we define a class called BANK-ACCOUNT, with two messages-method pairs--one to deposit, and one to withdraw. Each instance of a bank account should, of course, have its own private amount. Thus:

```
  (SETQ BANK-ACCOUNT
        (CLASS ()
               (ENV ':W (LAMBDA (AMT)
                          (IF (LE AMT ACCOUNT)
                              (SETQ ACCOUNT (SUB ACCOUNT AMT)))
                    ':D (LAMBDA (AMT)
                          (SETQ ACCOUNT (ADD ACCOUNT AMT))))

                ()
                '(ACCOUNT 0)))
```

Notice that we have prefixed the messages (W and D) with the package prefix (:); this will force the message names into the SYS package so that we'll be assured that instances that send messages (:W and :D) will find matches. There is a single instance variable, named ACCOUNT, that is initialized to zero.

Instances are created using the INST function, specifying the class, and optionally overriding some of the instance variable initializations (how much is initially placed in the particular account) and so we have:

(INST <class> &OPT <env> or ({<var> <object>})) SUBR

returns an instance of class <class> with instance variables <var> initialized to <object>. Instance variables not specified are initialized to the default values defined in the class.

```
     (SETQ MINE (INST BANK-ACCOUNT '(ACCOUNT 22)))
```

For comparison, we also could have written:

```
     (SETQ MINE (INST BANK-ACCOUNT))
     (MINE ':D 22)
```

When a message is passed to an instance that instance's class is first searched for a match. If no match is found the superclass of the class is searched, and so on until a match is found or we reach the root. In this case we generate an UNKNOWN-MESSAGE error.

When a method is applied, the instance variables of the instance and the class variables of the class become the current values similar to the environment of a closure. Any changes made by the method are "remembered" by the individual instance (or class in the case of class variables).

When a message is passed to an instance, the atom SELF is (effectively) bound to the instance. A method may recursively send messages to an instance by sending messages to SELF.

An extended example will serve to illustrate these points as well as set the stage for some others. In particular, we'll explore the issues of making some types of "turtle graphics" first-class in the sense that they will be objects rather than just pictures. These objects can be asked, for example, to grow, shrink, and move. We'll define a class called rectangle, and define some messages that will allow us to manipulate the graphical representation. See the section on turtle graphics at the end of this manual for details of the graphics commands.

```
(SETQ OBJECT (CLASS ()
                    (ENV ':P (LAMBDA () POSITION)
                         ':SETP (LAMBDA (POS) (SETQ POSITION POS)))
                    ()
                    '(POSITION (LIST 0 0))))
```

This gives us a simple class (with no superclass) that can respond to two messages--one to set a value (:SETP) and one to get a value (:P). The single instance variable POSITION will carry some representation of the spatial location of any instance. Thus:

```
(SETQ BLOB (INST OBJECT '(POSITION '(30 40))))
```

defines BLOB and locates it at the specified position.

We'll now define a subclass of the class OBJECT, called RECTANGLE, that will serve as our running example. RECTANGLE's instances will respond to requests to grow, shrink, hide and show, and finally, move.

```
(SETQ RECTANGLE (CLASS OBJECT
                       (ENV ':GW (LAMBDA (INC)
                                   (SELF ':E)
                                   (SETQ WIDTH (ADD WIDTH INC))
                                   (SELF ':S))

                            ':GL (LAMBDA (INC)
                                   (SELF ':E)
                                   (SETQ LENGTH (ADD LENGTH INC))
                                   (SELF ':S))

                            ':E  (LAMBDA ()
                                   (PEN 0)
                                   (POS (SELF ':P))
                                   (PEN 2) (HD 0)
                                   (REP 2 (FD WIDTH) (TR 90)
                                          (FD LENGTH) (TR 90)))

                            ':S  (LAMBDA ()
                                   (PEN 0)
                                   (POS (SELF ':P))
                                   (PEN 1) (HD 0)
                                   (REP 2 (FD WIDTH) (TR 90)
                                          (FD LENGTH) (TR 90)))

                            ':M (LAMBDA (POS)
                                   (SELF ':E)
                                   (SELF 'SETP POS)
                                   (SELF ':S))) )
                         ()
                         '(WIDTH 0 LENGTH 0)))
```

The occurrence of SELF within the methods indicates the operation of sending messages to an instance from within that same instance. So, for example, to show the movement of a rectangle, we erase the existing picture, change coordinates as specified, and then redisplay the object.

Now we can define a specific rectangle:

```
(SETQ RECT1 (INST RECTANGLE '(WIDTH 200 LENGTH 200)))
```

We can show it by:

```
(RECT1 ':S)
```

and we can shrink it by:

```
(RECT1 ':GW -20)
```

or move it by:

```
(RECT1 ':M '(50 50))
```

Of course, this all depends on accurate updating of the "local state" of each instance, so method application is performed with the equivalent of closure object running within an UNWIND-PROTECT to insure the class and instance variables are updated if a THROW or RETFRAME occurs.

Since Lisp programming tends to occur in a dynamic, exploratory setting, it is important to be able to experiment with programming techniques and modify parts and pieces without resorting to the archaic paradigm of edit-compile-run-debug-edit...and iterate. That is, we need have the freedom to modify decisions "on-the-fly". In the next section (Errors and Debugging) we'll illustrate the TLC primitives that will let us build forgiving debuggers--the kind that will let us modify programs that are under execution, let us supply values for undefined functions and then continue, or let us gracefully retreat from a computation that we no longer wish to pursue.

In this section we wish to demonstrate similar kinds of tools for the (somewhat less) dynamic task of class construction and exploration. As with interactive debugging, such tools expect that their user be cognizant of the power they possess.

So let's assume we want to extend the world of RECTANGLEs to recognize rotation. We need a message-method pair obviously. We also need to expand the position information of the class OBJECT, perhaps, or we could assume that orientation in the plane is only of interest to RECTANGLE. We assume the latter, so all modifications will be made to the subclass RECTANGLE. We need a new instance variable (HEAD) and need to modify the methods in the class to set the heading to HEAD, rather than 0. And of course we need to be careful; any instances of RECTANGLE that existed before the proposed modifications, are no longer accurate. Finally, we need operations to access and modify components of existing classes. That is the topic of the next section.

<u>Selectors for Classes</u>

Class objects respond to four messages that allow the user to examine or change the internal state of such an object. These techniques are only supplied for debugging; they are not designed for everyday use.

:SUPERCLASS MESSAGE

Returns the superclass of a class. Setting of the superclass requires care since it could create inconsistencies between prior and subsequent subclasses and instances that rely on this class.

```
(OBJECT ':SUPERCLASS) => nil
(WHO (RECTANGLE ':SUPERCLASS)) => (OBJECT)
```

where we use WHO to locate a name for the structure that represents the super-class.

:MSG-METHOD MESSAGE

Return or set the message-method environment of a class. Setting of message-method environments is useful for a class editor.

```
(RECTANGLE ':MSG-METHOD)
   => (ENV ':GW  ...
           ':GL  ...
            .  .  .)
```

Now we'll add new messages to objects of type RECTANGLE to reflect their orientation in the plane. This will involve new methods that will access the new instance variable, HEAD. For example:

```
(RECTANGLE ':MSG-METHOD
           (ENV (RECTANGLE ':MSG-METHOD)
                ':HEADING (LAMBDA (&OPT HD)
                            (IF (BOUNDP HD) (SETQ HEAD HD))
                            HEAD)))
```

Now we need to add the instance variables.

:INST-VARS MESSAGE

This message allows manipulation of the instance variable component of either a class or an instance. For either type of object, we can return the instance variables and their values; in the case of a class object, we get the default values, and for an instance, we get its actual instance values. A class object is allowed to change the the number of instance variables and any default settings. An instance may only change values, not the number of instance variables, since this latter property is a class-property, not an instance property. Even with these caveats, such surgery requires care; for example, prior and subsequent instances may not be consistent.

We can add a new instance variable to RECTANGLE by:

```
     (RECTANGLE ':INST-VARS
                (ENV (RECTANGLE ':INST-VARS)
                     'HEADING 0))))
```

```
and now:  (RECTANGLE ':INST-VARS)  => (ENV 'HEADING 0)

but            (RECT1 ':INST-VARS) => (ENV)
```

As with the message names, we have to be sure that package communication is maintained, here assuring that the reference to HEADING that appears in the updated messsage-method environment is the same symbol that appears in the updated instance variables.

:CLASS-VARS MESSAGE

Returns or sets the class variables of a class object. The class variables are implemented as an environment object. If the argument is not an environment then it is coerced into one.

:CLASS MESSAGE

Returns the class of a instance.

:PRINT MESSAGE

When an object of type INSTANCE is to be printed, a :PRINT message is sent to it. By defining a :PRINT message-method the print behavior of any class of instances can be redefined.

Errors and Debugging

TLC-LISP supplies a collection of functions to examine the state of the LISP machine. These are useful for debugging as well as building more general control-related programs.

Stack Frames

Whenever a function is applied, the TLC-LISP interpreter constructs a "stack frame". A function can be applied in several ways:

1. Direct call to APPLY, i.e. (APPLY ADD '(1 2 3))

2. Evaluating a list and applying the evaluated first element to the rest of the evaluated elements, i.e. (ADD 1 2 3).

3. Sending a message to an instance, the appropriate method is applied to the arguments, i.e. (MY-CAR ':SPEED 55)

4. Named call, a compiled version of method 2 above where the function name has been preserved.

This stack frame contains information about the state of the interpreter when the function is applied. Specifically, it contains information about which variables are temporarily bound inside this function application and, where possible, the functional value and the arguments to the function.

Several functions are supplied to examine stack frames. Examples of their use follow.

(TYPEFRAME <num>) SUBR

Returns the frame type of the level <num>-th frame as an atom. The following values are possible:

NORMAL -- frames erected by the interpreter for subrs, fsubrs, lsubrs, exprs, fexprs and macros are called NORMAL frames.

NAMED-CALL -- the function name of an named-call frame is available but the argument list has been compiled out.

SYSTEM -- a frame erected by the interpreter during processing of CATCH and UNWIND-PROTECT. No useful information may be extracted.

END -- the original frame erected at system initialization time. Attempts to access frames with level numbers greater than the END frame generate a NUMBER-OUT-OF-RANGE error.

(ARGSFRAME <num>) SUBR

This function returns a list of the arguments passed to the <num>-th pending function invocation. ARGSFRAME should not be used on a NAMED-CALL type frame. See BINDFRAME for examples.

(FCNFRAME <num>) SUBR

This function returns the function applied in the <num>-th previous pending function invocation. See BINDFRAME for examples.

(BINDFRAME <num> &OPT <var>) SUBR

If only <num> is supplied then return a list of the form ({<var> <object>}) which represents the variables whose values have been saved between the erection of frame <num> and the erection of frame <num> minus one. Each <object> represents the saved value, not the value current to the variable in frame <num>. If <var> is supplied BINDFRAME returns the current value of <var> in frame <num>.

```
(de FOO (a)
  (let ( (n 20) )
       (mul 2 (add 4 n 'a))))

(SETQ N 10)
(FOO -7) results in a NUMBER-EXPECTED error
(FCNFRAME 1) => error
(FCNFRAME 2) => add
(FCNFRAME 3) => mul
(FCNFRAME 4) => let

(ARGSFRAME 4) => (((n 20))(mul 2 (add 4 n (quote a))))

(BINDFRAME 4) => (n 10)       ; saved value
(BINDFRAME 4 'N) => 20        ; actual value

(FCNFRAME 5)
   => (lambda (a) (let ((n 20)) (mul 2 (add 4 n (quote a)))))

(BINDFRAME 5) => (a UNBOUND)  ; Since A was unbound when
                              ; FOO was activated.
```

These functions can be used to define a backtrace:

```
(de BACKTRACE (&OPT (count 3)
                    (start 1) )
; print a backtrace of pending function applications
; if COUNT is negative, prints all existing frames
  (if (or (eq (typeframe start) 'end)
          (zerop count) )
      nil
      ; else
      (print-frame start)
      (backtrace (sub1 count) (add1 start)))))

(de PRINT-FRAME (n)
  (terpri)
  (prin3 "+-- FRAME ")
  (print n)
  (if (neq (typeframe n) 'normal))
      (print (typeframe n))
      ; else
      (prin3 "Fcn: ")
      (print (fcnframe n))
      (prin3 "Arg: ")
      (print (argsframe n)))))
```

Continuing with the example in BINDFRAME:

```
(BACKTRACE 5) =>

+-- FRAME 1
Fcn: error
Arg: (number-expected a)

+-- FRAME 2
Fcn: add
Arg: (4 n (quote a))

+-- FRAME 3
Fcn: mul
Arg: (2 (add 4 n (quote a)))

+-- FRAME 4
Fcn: let
Arg: (((n 20)) (mul 2 (add 4 n (quote a))))

+-- FRAME 5
Fcn: (lambda (a) (let ((n 20)) (mul 2 (add 4 n 'a))))
Arg: (-7)

+-- FRAME 6
Fcn: toplev
Arg: nil
```

(WHO <object>) SUBR

Searches all symbols in all packages and returns a list of those symbols whose value is EQ to <object>. This function can be used to find the name of a function returned by FCNFRAME.

Continuing with the previous example:

```
(FCNFRAME 5)
  => (lambda (a) (let ((n 20)) (mul 2 (add 4 n 'a))))
```

If we have forgotten what function this is then:

```
(WHO (FCNFRAME 5)) => (foo)
```

(RETFRAME &OPT <num> ((<object> NIL)) SUBR

RETFRAME returns from the <num>-th pending invocation, using <object> as the returned value. If only the level number (<num>) is supplied then it uses NIL. If no arguments are supplied then it returns from all pending function applications, flushes the stack, and performs all necessary unbinding of variables. Protected forms are evaluated as the stack is unwound (see UNWIND-PROTECT).

Continuing with the previous example:

```
(RETFRAME 2 17) => 34
```

will return from the ADD (frame 2) with the value 17, so MUL (frame 3) can contine, multiplying the 17 by 2 and returning 34 to LET (frame 4), which unbinds N and returns the value 34 to FOO (frame 5), which unbinds A and returns the value 34 to TOPLEV.

(RESTART-FRAME <num>) SUBR

This is logically equivalent to

```
(APPLY (FCNFRAME <num>) (ARGSFRAME <num>))
```

Tapply and &TOP

Notice that the stack frame display functions do not display their own frames. These functions consider the "top" of the stack for their purposes (frame zero) to be the last application of the function TAPPLY.

(TAPPLY <fcn> <list>) SUBR

TAPPLY is identical to apply except that it binds the symbol &TOP to a value that the system can use to begin examination of stack frames. Essentially it defines frame number zero. The definition is equivalent to:

```
(de TAPPLY (fcn arglist &AUX (&top ;special value...;))
  (apply fcn arglist))
```

In the previous examples, the ERROR function (frame 1) executes a (TAPPLY TOPLEV NIL). This binds &TOP and prevents the display of the current TOPLEV frame or any more recent frames (BACKTRACE, FCNFRAME, etc.).

Debugger Functions

TLC-LISP includes a primitive to allow single-stepping of the evaluator to aid in the creation of debugging programs. The evaluator checks the value of the symbol EVALF before each evaluation. If EVALF is non-NIL then a special case occurs. This is illustrated by:

```
(de EVAL (obj)
  (if evalf
      (let ( (evalfn evalf) (evalf nil) )
        (apply evalfn (list obj)) )
      ; else
      ... evaluate obj normally))
```

Note that if EVALF has a non-NIL value, then it must be a function. That function is then applied in an environment where EVALF is bound to NIL, thus preventing infinite recursion.

EVALF is not usually referenced explicitly by user programs. Instead the EVALF feature is exploited by using our next function, EVALHOOK.

(EVALHOOK <form> <fcn>) SUBR

EVALHOOK binds EVALF to <fcn> and then evaluates <form>. The check in the evaluator for EVALF is bypassed when evaluating <form> itself but not in any subsidiary evaluations.

```
     (de HOOK (obj)
       (print obj)
       (eval obj))

     (EVALHOOK '(ADD 1 2 3) HOOK)

       prints: ADD
               1
               2
               3
       => 6
```

The evaluation of the list (ADD 1 2 3) is not hooked but the subsequent argument evaluations are hooked.

```
     (EVALHOOK '(ADD (ADD1 10) 20) HOOK)

       prints: ADD
               (ADD1 10)
               20
       => 31
```

A more elaborate tracer is:

```
     (de HOOK2 (obj &AUX (val (eval obj)))
       (prin0 obj)
       (prin3 "evaluates to ")
       (print val))

     (EVALHOOK '(ADD (ADD1 10) 20) HOOK2)

       prints: add evaluates to add
               (add1 10) evaluates to 11
               20 evaluates to 20

       => 31
```

HOOK2 displays the object and the result of its evaluation. Note that since the evaluator temporarily binds EVALF to NIL when applying HOOK we only trace one level "deep". We can change this by using EVALHOOK recursively in HOOK3:

```
     (de HOOK3 (obj &AUX (val (evalhook obj hook3)))
       (prin0 obj)
       (prin3 "evaluates to ")
       (print val))

     (EVALHOOK '(ADD (ADD1 10) 20))

       prints: add1 evaluates to add1
               10 evaluates to 10
               (add1 10) evaluates to 11
               20 evaluates to 20

       => 31
```

We could, for example, improve the hook function by abbreviating the message when printing constants.

Any LISP function may be executed within the hook function. By making the hook function interactive (using READ) we can create a very subtle debugger.

(ERROR {<object>}) LSUBR

ERROR prints the list of arguments and invokes TOPLEV. The current state of the system is preserved. As with TOPLEV, the system supplied ERROR function can be replaced by re-defining ERROR. The default definition of ERROR is equivalent to:

```
     (de ERROR (&REST l
                &AUX (current-sink <original console-out>)
                     (current-source (stream buffered-console-in))
                     (package sys:) )
       (terpri)
       (prin0 '**ERROR**)
       (mapcar prin0 l)
       (terpri)
       (do () (nil) (tapply toplev nil))))  ; infinite loop
```

Note input/output is temporarily reassigned to the console. Note also that ERROR uses TAPPLY rather than using APPLY or evaluating (TOPLEV) directly. This enables the stack frame functions to work correctly (see the section TAPPLY and &TOP above). If you write a custom ERROR function that invokes TOPLEV it should also use TAPPLY.

(RESTART) SUBR

When RESTART is activated it prompts with:

```
** RESTART **
Unwind stack (y/n)?
```

If "N" is typed then the stack pointer is reset to its initial position, losing all pending stack-frame information. If "Y" is typed then variable bindings are restored and unwind-protected forms are evaluated before the stack pointer is reset. Normally you would respond "Y".

RESTART also sets the symbols CURRENT-SOURCE, CURRENT-SINK, READ-TABLE, TOPLEV and ERROR, to their original values. RESTART should be used only as a last resort. It is useful when you have created buggy versions of TOPLEV and/or ERROR and cannot otherwise regain control of the system. See the reset character type in the section on Input below).

Important note: RESTART will bind READ-TABLE to the vector that was created when the system was first initialized. If you have destructively modified READ-TABLE (via DMC or TYPECH) then you may be unable to regain control of the system. Always do READ-TABLE experiments on a copy of the existing READ-TABLE.

Fatal Errors

The TLC-LISP system constantly checks itself for internal consistency. Such things as pointers into memory areas that were never initialized or internal data structures containing objects of the wrong type can generate a "fatal error". Exhausting free storage is also considered fatal. Misuse of the functions in the Advanced Functions section can generate fatal errors. Fatal error messages are listed in the Errors Appendix.

When an inconsistency occurs that the system may not be able to recover from, a fatal error is generated. The system prints information about its internal state and then prints

```
Press C to (ERROR 'CONTINUE), R to (RESTART),
        E to (EXIT), A to Abort
```

"C" will execute (ERROR 'CONTINUE). Always try this first.

"R" will execute (RESTART). When you run out of free space this response will often free enough memory for you to resume development.

"E" will execute (EXIT), open files are closed and the stack is unwound before control is returned to the operating system.

"A" will immediately return control to the operating system. This is a last resort. The stack is not unwound and open files are not closed thus some data may be lost.

It is also possible that a fatal error will occur due to a problem with the interpreter itself. Write down **all** of the displayed state information along with a description of what you were running prior to the fatal error, then contact The LISP Company.

Input and Output

Though LISP was created in the era of batch-processing, it is most definitely an interactive language. Its exploratory and incremental programming style thrives on a calculator-like immediacy. An expanding part of LISP's interactive nature is its input and output. In this version of TLC-LISP we support several different sytles of I/O behavior, from traditional disk files to generalized stream operations described by arbitrary LISP functions.

We also allow the user to redefine the reading and printing behavior of LISP, thereby redefining the outward appearance of LISP.

We'll begin with the most mundane and move towards the exotic. The simplest way to get information into the machine (other than the console) is to use the LOAD function. In fact, the system invokes LOAD automatically at start-up. It looks for the file LISP.SYS, and if present, reads from it as if it contained keystrokes from the keyboard. The effect, then, is to read and evaluate that input.

(LOAD <str> &OPT (flag NIL)) SUBR

LOAD executes a READ-EVAL loop using the file named <str>. If flag is non-NIL then the value of each expression read from the file is printed.

LOAD is equivalent to:

```
(de LOAD (name &OPT (flag NIL)
               &AUX (file (open name 'read))
                    last-exp )
  (unwind-protect
    (setq last-exp (loadl nil))
    (close current-source) )
  last-exp)

(de LOADl (last-exp &AUX (exp (read file)) )
  (if (eq exp 'end-of-file)
      last-exp
      (loadl (let ((val (eval exp)))
                  (if flag (print val))
                  val))))
```

The remainder of this section will be taken up with a discussion of many of the components that make up LOAD's definition. We'll begin with file names.

Disk File Name Formats

In the following file name descriptions, [xxx] (square brackets) means that xxx is optional, it may occur zero times or once. {xxx} (curly brackets) means that xxx may occur zero or more times. Finally, TLC-LISP appends a .LSP extension to all file name strings that do not designate an extension.

A file name in the CP/M-86 version of TLC-LISP is a string in the following format:

```
[n/] [d:] name [.ext]
```

where:

n is the optional user number specifier (0 to 15)

d is the optional drive specifier (A to P)

name is the file name, 1 to 8 characters

ext is the optional extension, 0 to 3 characters

The following are legal CP/M-86 file specification strings:

"FOO"	foo.lsp, current drive and user
"FOO.LSP"	foo.lsp, current drive and user
"FOO."	foo, current drive and user
"B:FOO"	b:foo.lsp, current user
"2/b:FOO"	b:foo.lsp, user 2
"3/FOO.DAT"	foo.dat, user 3, current drive

A file name in the MSDOS V1.x version of TLC-LISP is a string of the following format:

```
[d:] name [.ext]
```

where:

d is the optional drive specifier (A to P)

name is the file name, 1 to 8 characters

ext is the optional extension, 0 to 3 characters

The following are legal MSDOS V1.x file specification strings:

"FOO"	foo.lsp, current drive
"FOO.LSP"	foo.lsp, current drive
"FOO."	foo, current drive
"B:FOO"	b:foo.lsp

A file name in the MSDOS V2.x version of TLC-LISP is a string of the following format:

```
[d:] [\] {dir\} name [.ext]
```

where:

d is the optional drive specifier (A to P)

\ before the directory name indicates the path; search begins with the root directory. If it is absent then the search begins with the current directory.

dir\ is an existing directory name followed by a backslash. A double dot .. indicates the parent directory of the current directory.

name is the file name, 1 to 8 characters

ext is the extension, 0 to 3 characters

The following are legal MSDOS V2.x file specification strings:

"FOO"	foo.lsp, current directory and drive
"FOO.LSP"	foo.lsp, current directory and drive
"FOO."	foo, current directory and drive
"B:FOO"	b:foo.lsp, current directory
"\FOO"	foo.lsp in the root directory, current drive
"A:\LISP\FOO"	a:foo.lsp in the sub-directory LISP of the root directory
"B:..\..\FOO"	b:foo.lsp two directories "up" from the current directory

Finally, file name specifications may also include "wild-cards"; these extensions are handled in the specific utility packages written in TLC-LISP for each operating system. See files CPM.LSP, MSDOS.LSP, MSDOSV1.LSP, and MSDOSV2.LSP.

How To Build Streams

With knowledge of file names in hand, we can begin to build the LISP objects that will let us connect readers and printers to sources and sinks. The LISP objects that perform these services are called streams.

A stream is made up of:

1. A look-ahead character, accessed when the stream is used for input.

2. A character source (file or applicable object capable of returning characters) if the stream is used for input, or a character sink (file or applicable object capable of accepting characters) if the stream is used for output.

We can build a stream either by opening a file or by explicitly constructing a stream object from a LISP function that will supply or absorb characters. Thus:

(OPEN <str> <mode> &OPT 'RANDOM) SUBR

<Str> designates a file name as described above. <Mode> is the symbol READ, WRITE or UPDATE. The value returned is a stream object, suitable as an argument to READ or PRINT. If <mode> is WRITE then the file is deleted if it exists. If a file with the same file name is already open (determined by its existence on FILE-LIST) the a FILE-ALREADY-OPEN error is generated unless all the references are of type READ. The stream returned by OPEN is added to the list FILE-LIST.

For example, if the file TEMP.LSP exists, then:

```
FILE-LIST => nil
(SETQ F (OPEN "temp" 'READ)) => <A:\TEMP.LSP>   ; MSDOS
FILE-LIST => (<A:\TEMP.LSP>)
```

A stream object may also be explicitly created, using a LISP function as the originator of, or depositary for, characters. Thus:

(STREAM <fcn>) SUBR

Creates a stream object. The <fcn> must be a function of no arguments for streams that will be read, a function of exactly one argument for streams that will be written, or a function with no required arguments and one optional argument for streams that will be read and written.

For example, (STREAM BUFFERED-CONSOLE-IN)

returns a stream that is equivalent to the initial value of CURRENT-SOURCE.

The TLC-LISP Read and Print functions use the values of CURRENT-SOURCE and CURRENT-SINK respectively, as their default targets for input and output. All input/output functions will accept an optional stream object as an argument. As we'll see shortly, OPEN or STREAM can return an object suitable for the following kinds of input operations:

```
(READ F) => first object in the stream F or
(READCHAR F) => first byte in F or
(READLINE F) => first line in F as a string
```

Given a stream, we may locate the sink or source in a specific portion of that stream, or we may close it to indicate that we have completed our operations on it. Thus:

(CLOSE <stream>) SUBR

<Stream> is closed. Input and/or output are no longer permitted, and the operating system records any changes (if written or updated). <stream> is removed from FILE-LIST (and so must have been created by a call to OPEN). Continuing the previous example:

```
(CLOSE F)
FILE-LIST => nil
(READ F) => error        ; cannot read
```

(SEEK <stream> &OPT <num>) SUBR

If <num> is absent, then SEEK returns an integer representing the position of the next character in the file (the first character in the file being at position one). If <num> is supplied and the file was opened for random access, then SEEK sets the internal pointer of the file <stream> to the <num> position. Subsequent reads or writes occur from the new position.

Disk File Functions

(FILE-ACCESS <stream or file>) SUBR

Returns the current access code for the file. Values are as follows:

Bit	Meaning
0	Read access
1	Write access
2	Closed
3	Random access
4-7	Used internally or reserved

(FILE-NAME <str or stream>) SUBR

If the argument is a stream, then the filename is returned. If the argument is a string, then the string is used in the current disk context (current drive, current directory, current user number) and a string that would unambiguously represent the file is returned. (FILE-NAME <str>) is used internally when files are opened to prevent confusion when the disk context is changed during a file's lifetime.

```
(FILE-NAME "foo") => "0/B:FOO.LSP"       ; CP/M

(FILE-NAME "foo") => B:\DIR1\FOO.LSP"    ; MSDOS
```

The functions in the remainder of this section are EXPRS that can be found in the following operating system-specific files: CPM.LSP, MSDOS.LSP, MSDOSV1.LSP, and MSDOSV2.LSP.

(DIR &OPT (<str> "*.LSP")) EXPR

Return a list of strings representing the files whose names match <str>.

```
(DIR "foo.*")
   => ("0/A:FOO.LSP" "0/A:FOO.BAK" "0/A:FOO.BAR")
```

(FILE-ERASE <str>) EXPR

Delete the file(s) that match <str>. If the file is open then a FILE-OPEN error is generated.

```
     (DIR "foo") => ("0/A:FOO.LSP")    ; CP/M
     (FILE-ERASE "foo")
     (DIR "foo") => nil
```

(FILE-RENAME <new> <old>) EXPR

Rename the file <old> to <new>. If file <new> exists then a FILE-EXISTS error is generated.

```
     (RENAME "foo" "baz")
```

causes BAZ.LSP to be renamed to FOO.LSP.

(FILE-EXISTS <str>) EXPR

Return T if a file that matches <str> exists in the directory else return NIL.

```
     (FILE-EXISTS "foo") => t
     (FILE-EXISTS "bar") => nil
```

(FILE-SIZE <str>) EXPR

Return the size of the specified file in bytes.

```
     (FILE-SIZE "foo") => 8192
```

Read Functions

Now that we have files and streams well in-hand, let's apply them to input and output operations. Care must be taken when these stream operations involve a user-defined function (rather than a file). User-defined functions that occur in a stream may not recursively invoke read or make use of the system buffer while a token has been partially read. This is because the token read routines (like the routines that read atoms or strings) make use of the system buffer and are thus not reentrant. Invoking read or using the buffer between the reading of tokens is allowed.

(READ &OPT <stream>) SUBR

If the optional parameter is supplied, it must be a stream object. READ is the main LISP parsing routine. It reads the next well-formed expression from the current input source defined by CURRENT-SOURCE, and returns that expression as value after establishing its internal form.

(READCHAR &OPT <stream>) SUBR

The next byte from the source <stream> is returned as a character. The NEXT field of the stream is used if it is non-NIL (a character). The READ-TABLE is ignored.

(NEXT <stream> &OPT <chr>) SUBR

If <chr> is present, the look-ahead character is set to it, otherwise the current look-ahead char is returned (but not changed). If there is no lookahead character then NIL is returned. For example if a file named TEMP.LSP contained the line of text:

```
123(a b c)"foo"456
```

Then:

```
(SETQ F (OPEN "temp" 'READ))
(NEXT F) => NIL
```

Nothing has yet been read from the file.

```
(READ F) => 123
(NEXT F) => \(
```

The scanner had to examine the next character to be sure that the character 3 was the last digit in the number.

```
(READ F) => (A B C)
(NEXT F) => NIL
```

The list ended with the right parenthesis, so the double-quote (") was not read.

```
(READ F) => "foo"
(NEXT F) => \4
```

The character after the trailing double quote had to be examined to insure that this wasn't an embedded double quote like "foo""".

Now, let's change the lookahead character.

```
(NEXT F \9)
(READ F) => 956
```

The character 4 is lost.

```
(ASCII (NEXT F)) => 13
```

The carriage return character at the end of the line.

```
(READ F) => END-OF-FILE
```

The attempt to read another object fails. The distinguished atom END-OF-FILE is returned.

(READLINE &OPT <stream>)

Returns a string of all the characters up to the next carriage return. The carriage return is not included. Returns the atom END-OF-FILE if a character of type eof (^Z) is encountered. A function to print the contents of files could be written as:

```
(de TYPEFILE (name &AUX (file (open name 'read)) )
  (unwind-protect
    (do ( (line (readline file) (readline file)) )
        ( ((eq line 'end-of-file) nil) )
        (print3 line) )
    (close file) ))
```

(ED-READLINE &OPT <stream>)

This is a special version of READLINE for the editor. It is identical to READLINE except that it expands tabs (^I) by replacing them with spaces upto the next column that is a multiple of eight. The string that is created by this process is terminated by an eof (^Z) character. ED-READLINE could be substituted for READLINE in the above TYPEFILE example to correctly print files with embedded tab characters.

Print Functions

The print functions can also take an optional stream argument. The default stream is the value of CURRENT-SINK.

(PRIN0 <object> &OPT <stream>)

Print the (representation of) <object> to the stream followed by a space. Strings and characters print with their delimiters ("" and \) and symbols print with their package prefix if their package is not the current package.

```
(PRIN0 (ADD1 10)) prints 11<space>

(PRIN0 ':FOO) prints foo<space> if PACKAGE is SYS:
              prints :foo<space> if PACKAGE is not SYS:
```

(PRIN1 <object> &OPT <stream>)

Print the <object> to the stream. Strings and characters print with their delimiters, symbols with their package prefix if their package is not the current package.

```
(PRIN1 "foo") prints "foo"   => "foo"

(PROGN (PRIN1 "foo")(PRIN1 43)) prints "foo"43  => 43
```

(PRIN2 <object> &OPT <stream>)

Print the <object> to the stream followed by a space. Strings and characters print without their delimiters, symbols print without their package prefixes.

```
(PRIN2 "foo") prints foo<space>

(PRIN2 ':FOO) prints foo<space>
```

(PRIN3 <object> &OPT <stream>)

Print the <object> to the stream. Strings and characters print without their delimiters, symbols print without their package prefixes. No trailing space is printed.

```
(PRIN3 "foo") prints foo

(PRIN3 ':FOO) prints foo
```

(TERPRI &OPTIONAL <stream>) SUBR

Print a carriage return, followed by a line feed.

(PRINT <object> &OPTIONAL <stream>)

Prints the <object> followed by a carriage return and line feed. Packages, characters, and strings print with their prefixes or delimiters.

This function is equivalent to:

```
(PROG1 (PRIN0 <object> <sink>) (TERPRI <sink>))
```

Console Functions

Though the keyboard and screen are just input/output devices, their requirements are sufficiently unique to warrant a separate section. The functions of this section deal with the rapid response to interactive input and output.

(TYS) SUBR

Checks the status of the keyboard. If a key has been struck, T is returned, otherwise NIL is returned. Does not affect the input stream. This function is used in situations where we wish to interrupt a computation if there is keyboard input.

(CONSOLE-IN) SUBR

Returns a character from the console without echoing. If you want an input stream that "fires" as soon as parentheses balance you can use CONSOLE-IN combined with echoing; something like:

```
(SETQ CURRENT-SOURCE
      (STREAM (LAMBDA () (PRIN3 (CONSOLE-IN)))) )
```

(BUFFERED-CONSOLE-IN) SUBR

Returns a character from the line buffer. The buffer management routine accepts the backspace characters ^H and RUB and the line erase character ^X. The line may be edited using these characters until RETURN (^M) is entered. Once RETURN is entered the characters in the buffer are returned to the caller of BUFFERED-CONSOLE-IN until exhausted, in which case the buffer manager refills the buffer using CONSOLE-IN (echoing to CONSOLE-OUT) and the process repeats. The default value for CURRENT-SOURCE is equivalent to:

```
(STREAM BUFFERED-CONSOLE-IN)
```

(CONSOLE-RESET) SUBR

This function flushes the CONSOLE-INPUT typeahead buffer and the BUFFERED-CONSOLE-IN line buffer. Causes subsequent calls to TYS to return NIL until a character is typed. Useful for error handlers.

(CONSOLE-OUT <char>)

(CONSOLE-OUT <str> &OPT <num1> <num2>) SUBR

If only one argument is supplied then the character or the characters of the string are sent to the terminal.

If three arguments are supplied then CONSOLE-OUT begins sending from index <num1> or from the first character in the string if <num1> is absent, and continues sending characters until a ^Z is reached, then sends spaces until <num2> characters have been sent. If <str> is exhausted before <num2> characters have been sent, CONSOLE-OUT sends spaces until <num2> characters have been sent. If a tab (^I) character is reached then spaces are sent until the cursor position is a multiple of eight (the beginning of the string is assumed to be at column one for the purpose of expanding tabs.)

CONSOLE-OUT with the optional arguments is useful for fast screen update in editor programs.

```
(CONSOLE-OUT \a) prints a

(CONSOLE-OUT "foo") prints foo

(CONSOLE-OUT "foo" 1 2) prints fo

(CONSOLE-OUT "foo" 2 10) prints oo followed by 8 spaces
```

Altering Read Behavior

The LISP reader recognizes various special character types. These types are stored in a vector READ-TABLE. The type for a character C is stored at index (ADD1 (ASCII C)) in READ-TABLE. READ-TABLE may be changed or temporarily bound like any other variable. Later we will describe how to modify and extend these character types, but now we will discuss the default settings of standard TLC-LISP.

These characters include:

()	the list delimiter characters
'	the quote character
^	the control character prefix
[]	the vector delimiter characters
‘	the backquote character
,	the backquote escape character
.	the dot character
"	the string delimiter characters
;	the comment character
\	the character character
#	the number-base prefix character
:	the package character
^G	the abort character
^K	the reset character
^S	the pause printing character
^Q	the resume printing character

A description of each follows:

The list delimiters () delimit literal lists and dotted pairs.

The quote character '. Instead of requiring the user to type (QUOTE <exp>), TLC-LISP supports the abbreviation '<exp>. This is implemented as a built-in read macro.

```
'(A B) is the same as (QUOTE (A B))

''(A B) is the same as (QUOTE (QUOTE (A B)))
```

The control character prefix ^. To simplify the input of non-printable ascii characters, the abbreviation ^<char> is supported. This is implemented as a read macro and loaded from the file LISP.LSP.

```
^A and ^a are equivalent to (ASCII 1)
```

The vector delimiter characters []. The reader treats objects between opening and closing square brackets as the elements of a literal vector. The elements between the brackets are not evaluated. This is implemented as a read macro and loaded from the file LISP.LSP.

```
[1 2 A B] is equivalent to (VECTOR 1 2 'A 'B)
```

The backquote ‘. To simplify and improve the readability of macro definitions the backquote is supported. A backquoted list is treated like a quoted list except that objects preceded by a comma are evaluated, objects not preceded by a comma are quoted. This is implemented as a read macro and loaded from the file LISP.LSP.

```
‘(do ( (,var 1 (add1 ,var))) (((eq ,var ,end) nil) ))
```

is equivalent to

```
(list 'do (list (list var 1 (list 'add1 var))) (list
  (list (list 'eq var end) nil)))
```

The dot character . is used in the representation of dotted pairs.

```
(A . B) is equivalent to (CONS A B)
```

The string delimiter ". String literals are presented to TLC-LISP as arbitrary character sequences bracketed by a pair of double quote characters. Thus "ABCD" is a string as is "(foO". To include the character " in a string use a two consecutive double quotes, thus the string:

```
"a single "" mark"
```

contains one double quote character.

The comment character ;. A comment begins with ";" and ends either with another ";" or an end-of-line indication. Thus:

```
(DE MAGIC (N ;an integer; L ;a non-empty list;)
  (COND
    ( (ZEROP N)  ;in this case M must be  4
      (CHECK M) )
    ( (NULL (REST L))
       ... )            ; another comment
    ...    ))
```

contains four comments.

The character character \. This character is used to designate a single character literal. Note that the string "A" is not the same as the character \A just as the list (A) is not the same as the symbol A.

```
\a is equivalent to (ASCII 96)
```

The number-base character #. This character can be used in two different ways. If the character following the # is a left square bracket then the number inside the square brackets is taken as the base of the number following the brackets. The number inside the brackets is always interpreted as a decimal number. This is implemented as a read macro and loaded from the file LISP.LSP.

```
#[2]1101 is the base two representation of decimal 13
#[16]A0 is the hexadecimal representation of decimal 160
```

If the # is followed by a alphanumeric character then the base defaults to the value of INPUT-BASE which may take values from 2 through 36. Thus if INPUT-BASE is sixteen then:

```
#10 is decimal 16 and #10A is decimal 266
```

The digits of floating point numbers are always printed in decimal. The output behavior of fixnums and integers is determined by the value of OUTPUT-BASE. If OUTPUT-BASE is ten, then the undecorated form is printed; otherwise the prefix #[<n>] is used where <n> is the current value of OUTPUT-BASE in decimal.

```
12       (output-base is ten)
#[4]23   (output-base is four)
```

Note that the input behavior for numbers may be changed by redefining the \# character macro. The output behavior may be changed via the PR function.

The package character :. See the section on Symbols and Packages.

The abort character ^G. If this character is typed at the console, the system executes an (ERROR 'USER-ABORT). The console does not have to be explicitly read for this action to occur, anytime a character is typed and inserted into the typeahead buffer the READ-TABLE value is checked.

The reset character ^K. If this character is typed at the console the the system executes the fatal error routine (see the Fatal Errors section) The console does not have to be explicitly read for this action to occur, anytime a character is typed and inserted into the typeahead buffer the READ-TABLE value is checked. See the cautions under RESTART in the Error Section. This feature can save you when a user defined version of ERROR or TOPLEV has bugs in it but always try the abort character (^G) first.

The pause printing character ^S. If this character is typed at the console then subsequent print attempts will cause the system to wait for a resume printing character.

The resume printing character ^Q. Resumes printed output after a pause print (^S) is typed.

Besides these default special characters, TLC-LISP also provides the the ability to define read macros. These macros have single-character names and take effect when that single character is recognized in the input stream. For example, the special quote-character, ', is a built-in read macro.

(DMC <char> <list> {<exp>}) FSUBR

<Char> is the name of the character macro; <list> designates the local variables (initialized to NIL) which will be used during the evaluation of the macro body, {<exp>}. The value returned from the DMC declaration is <char>; the value returned (to the LISP reader) when the macro is activated is the value of the last <exp>. For example, we could declare the ' macro by:

```
(DMC \' ()
  (LIST (QUOTE QUOTE) (READ)) )
```

The macro declaration is accomplished by two actions: first, the body of the definition is treated as a DE; then, the entry in READ-TABLE for <char> is modified to reflect its new position as a macro.

(TYPECH <char> &OPTIONAL <num> or <fcn>) SUBR

If the second argument is missing, TYPECH gives the current READ-TABLE value for <char>. If the second argument is <num>, TYPECH sets the READ-TABLE value for <char> to <num>. If the second argument is a function then the <char> becomes a read macro. With TYPECH the user can redefine the syntax accepted by the reader at a very low level.

Acceptable values for <num> are the following:

0: totally ignore the character

1: the character is like the dot (.) in "dot-notation".

2: the character begins a comment; ignore all input until a comment-end character is seen. (e.g. ;)

3: the character ends a comment. (e.g. ; and <cr>)

4: the character is a separator (e.g. space and tab)

5: not used

6: the character is a string delimiter, (e.g. ")

7: these are digit characters (e.g. 0 thru 9)

8: these are normal characters, (e.g A thru Z)

9: the character-characters. e.g. \

10: the left parenthesis character, (

11: the right parenthesis character,)

12:

13: not used

14: the end-of-file character, ^Z

15: the package prefix character, :

16: the abort character, ^G

17: the reset character, ^K

18: the print resume character, ^Q

19: the print pause character, ^S

END-OF-FILE   SYMBOL

When a character with a read table value of 14 (end of file) is scanned, the reader and the scanner will return the symbol

END-OF-FILE.

If this character is detected by the reader while parsing a list (this can occur when loading a file with an unmatched left parenthesis) then an UNEXPECTED-END-OF-FILE error is generated.

Altering Print Behavior

Print routines for any object type may be supplied by the user. To supply your own print function or redefine one already supplied the PR function is used.

(PR <type> &OPT <fcn>) SUBR

Returns or sets the print behavior of objects of type <type>. The print behavior is invoked when an object is printed. For example, the print behavior for vectors could be defined as follows:

```
(de PRVECTOR (v)
  (prin3 \[)
  (prvectorl v 1)
  (prin3 \]))
```

where PRVECTORl is responsible for printing the interior portions of the vector.

```
     (de PRVECTOR1 (v i &AUX (len (length v)))
       (cond
         ( (gt i len) )  ; end
         ( (eq i len)    ; no trailing space on last element
           (prin1 (vref v i)) )
         ( t (prin0 (vref v i))
             (prvector1 v (add1 i)) )))
```

The routine is installed by:

```
     (pr 'vector prvector)
```

Always test custom print routines before installing them with PR. Bugs in such routines can cause you to lose control of the system. See RESET and RESTART for techniques and options on recovery.

Operating System Specific Functions

Each operating system has its own set of special features. This section covers the specific operations that TLC-LISP supports for the two versions of MSDOS, and CPM/86.

MS-DOS Functions

The following functions are supplied in MSDOS versions of TLC-LISP.

(AXO <char>) SUBR

Send <char> to the auxiliary output device.

(AXI) SUBR

Get a character from the auxiliary input device.

(LPR <char>) SUBR

Send <char> to the lineprinter device.

Utility Functions Supplied in the file MSDOS.LSP

For the time functions, <time> is a four element list of the following format:

(hours minutes seconds hundredths)

(TOD) EXPR

Return or set the time of day clock.

(SUBTIME <time1> <time2>) EXPR

Return the difference between the two times.

(TIME {<form>}) FEXPR

Prints the amount of time consumed in evaluating {<form>}.

Additional Functions for MS-DOS V2.x

(MKDIR <str>) EXPR

Create a sub-directory named <str>, <str> may contain both drive and pathnames.

(RMDIR <str>) EXPR

Remove the directory specified by the drive and pathname in <str>.

(CHDIR &OPT <str>) EXPR

Return or set the current directory to the pathname specified in <str>.

(EXEC &OPT <str>) EXPR

If no argument is supplied, then this function invokes a second copy of the MSDOS command interpreter. If <str> is supplied, then the program and arguments specified by <str> is executed. Sufficient memory for the program's needs must be available; thus the M command line option must be used when LISP is invoked (see Part II). If no extension is supplied on the program name then ".COM" is assumed.

```
(EXEC "sort <foo.lsp>bar.lsp")
```

runs SORT on input, FOO.LSP, with output directed to BAR.LSP.

Control is returned to LISP whenever the process terminates, either by a programatic exit or by typing EXIT to the MSDOS process.

CP/M-86 Functions

The following functions are supplied in CP/M-86 versions of TLC-LISP.

(AXO <char>) SUBR

Send <char> to the paper tape punch (AXO:) device.

(LPR <char>) SUBR

Send <char> to the lineprinter (LST:) device.

(AXI) SUBR

Get a character from the paper tape reader (AXI:) device.

(USER &OPT <num>) EXPR

Return or set the current user number.

```
(USER) => 0
(USER 1)
(USER) => 1
```

Autoloading Functions and Values

The major constraint on TLC-LISP is the size of available memory. Sophisticated applications can soon exhaust all of the free space, even in a full 8086 environment. One way to forestall this difficulty is to "virtualize" large programs that may only be needed for short durations. Rather than explicitly expunging functions to reclaim their space, TLC-LISP contains a "virtual" type called ALOAD. Objects of type ALOAD are described by a file name (string) and a relative position in that file. Whenever an attempt is made to access an object of type ALOAD, the TLC-LISP evaluator retrieves the actual value from the file.

A typical autoload file consists of two parts: a directory file that contains calls on ALOAD, and the text file that contains the actual Lisp code. Use LOAD to install the directory file, and then subsequent references to any ALOAD objects will access the text portion of the Autoload pair, performing a READ-EVAL pair on the text it finds.

The directory of an ALOAD file can be constructed by reading an existing file, writing out its contents onto a file, while using SEEK to discover the actual position of that information, and finally recording that position information on a directory file. See ALOAD.LSP for details.

Two types of autoloading are available: "smash" and "no-smash". A "smash" object is loaded in and replaces the ALOAD object; subsequent references to that symbol will retrieve the value without accessing the disk. This type of loading is useful for functions and values that expect to be re-used. The system also saves the autoload information so that the value may be "unsmashed" when the object is no longer needed. This is done by UNSMASH, a function defined on the file ALOAD.LSP.

A "no-smash" value is ethereal. Every access to it will cause a seek to the disk. Such values are useful for "one-shot" evaluations, like initialization code.

All of the autoloading system is written in LISP except for the primitive to create objects of type ALOAD:

(ALOAD <str> <pos>) SUBR

An object of type aload is created, using <str> as the file name, and using <pos> as the position in that file which determines the begining of the object.

(AUTO <filename> <symbol> {<var>}) FEXPR

The indicator <symbol> is either SMASH or NO-SMASH, <var>s are the symbols that will become aload objects that will access file <filename> when evaluated.

Miscellaneous Utility Functions

(GC &OPT flag) SUBR

This function makes an explicit call on the garbage collector. If <flag> is supplied then a list of the number of objects marked for each type (of the form ({<type> <count>})) is returned for all non-zero <counts>.

GCBEEP ATOM

When GCBEEP is non-NIL, LISP will execute a (CONSOLE-OUT ^G) before every garbage collect.

(EXIT) SUBR

This function returns control to the operating system. First, the stack is unwound and all protected forms are evaluated; then any files remaining on FILE-LIST are closed. Finally LISP relinquishes control to the operating system.

(FREE &OPT (<type>)) SUBR

If <type> is absent, return a list ({<type> <num>}) <num> being the length of the freelist of type <type>. Lengths of zero are not returned. If <type> is present, return the length of the associated freelist as a number. In addition to objects, FREE also computes the space remaining on the stack, the number of pages in object space that are currently unallocated and the number of bytes in byte space that are unallocated.

```
     (FREE) => (symbol 7 list 19 expr 91 fexpr 188 macro 180
       float 183 integer 160 stream 181 subr 152 file 88 string
       12 vector 92 pkg 92 stack 16226 bytes 115543 pages 411)

     (FREE 'STREAM) => 181
     (FREE 'STACK) => 16266
```

To get an accurate reading of the free space in the system, preface the call to FREE with a call to the garbage collector.

(BUFFER &OPT <num>) SUBR

If <num> is absent, return the current size of the system buffer in bytes, else set the buffer to size <num>. The size of the buffer limits the size of new strings, vectors and environments -- strings require one buffer byte per character, vectors, environments, for example, require two buffer bytes per element. If the buffer is too small for an attempted operation, a BUFFER-OVERFLOW error occurs.

```
(BUFFER 100)
(NEWSTRING 110 \a) generates a BUFFER-OVERFLOW error
(BUFFER 1000)
(NEWSTRING 110 \a) => returns the string
```

(STACK &OPT <num>) SUBR

Return or set the total size of the stack in bytes. The number is coerced into the nearest "good" size. The current stack is first unwound thus the current state (stack frames and bindings) is lost. The stack is allocated from Byte space. If a block of sufficient size is not available a BYTE-SPACE-EXHAUSTED error is generated. If (STACK <num>) used in a file that is being loaded, the load will terminate due to the unwinding of the stack.

```
(STACK) => 4092
(STACK 60000) => 59996
```

(INBYTE <num>) SUBR

(INWORD <num>) SUBR

Return an 8 or 16 bit number from I/O port <num>.

(OUTBYTE <num1> <num2>) SUBR

(OUTWORD <num1> <num2>) SUBR

Sends an 8 or 16 bit number <num2> to I/O port <num1>.

(EXAMINE <num>) SUBR

(EXAMINE-WORD <num>) SUBR

Return the byte or word at memory location <num> as an unsigned number.

(DEPOSIT <num1> <num2>) SUBR

(DEPOSIT-WORD <num1> <num2>) SUBR

Memory location <num1> is set to the byte or word <num2>. These functions both return <num2> as their value.

(INTERRUPT <num> &OPT (<vect1> NIL) (<vect2> NIL)) SUBR

Generates an 8086 software interrupt at level <num>. <Vect1>, if supplied, is a vector with exactly ten elements. The elements correspond to the 8086 CPU registers as follows:

Element	Register
1	AX
2	BX
3	CX
4	DX
5	DI
6	SI
7	BP
8	ES
9	DS
10	FLAGS

Each element of <vect1> must be a number or NIL. If an element is a number, then the corresponding register is set to that value before the interrupt is generated. If the element is NIL, then the register value is unknown. If <vect1> is not supplied or is NIL, then no registers are set.

After the interrupt, <vect2> is examined. If <vect2> is NIL or was not supplied then INTERRUPT returns NIL. Otherwise <vect2> must be a vector with exactly ten elements, each element being either NIL or non-NIL. If an element is non-NIL then that element is set to the value of the corresponding register after the interrupt. <Vect2> is returned as the value of INTERRUPT. Omitting either vector (if that makes sense for the interrupt) results in faster execution. Also, the more elements of either vector that are NIL, the faster the execution.

For example, in CPM-86

```
(INTERRUPT 224 [NIL NIL 32 2 NIL NIL NIL NIL NIL NIL])
```

invokes the CP/M-86 set user number function call, setting the current user to 2. The command (32) is passed in register CL and the value (2) is passed in register DL.

For example, on an MSDOS machine, we can invoke the read console buffer as follows:

```
(SETQ STR (STRING (ASCII 80) (NEWSTR 80 \ )))

(INTERRUPT 33 [#A00 NIL NIL (OFFSET STR) NIL
  NIL NIL NIL (SEGMENT STR) NIL] )
```

or equivalently using the INT macro

```
(INT 33 :ax #A00 :dx (offset str) :ds (segment str))
```

invokes the MSDOS read console buffer function using STR as the buffer. The function code (10) is passed in register AH. The buffer location is passed in DS:DX. The first byte of the buffer is expected to be the length.

See the INT macro in the section on the file SYS.LSP

Functions Defined in the File SYS.LSP

For more information on the following functions the file SYS.LSP may be examined for the commented source code.

(NEQ <object1> <object2>) MACRO

Expands to (NOT (EQ <object1> <object2>)).

(?STRING <object>) EXPR

Convert arg to string if not already.

(IFTRUE <pred> {<form>}) MACRO

Expands to (IF <pred> (PROGN {<form>})).

(MIN <object1> <object2>) EXPR

Return lesser arg.

(MAX <object1> <object2>) EXPR

Return greater arg.

(MEMBER <object> <struc>) EXPR

Return <object> position in <struc> or NIL.

(?RPLACB <sexp1> <sexp2>) MACRO

If second arg NIL then (RPLACD <sexp1> NIL) else (RPLACB <sexp1> <sexp2>).

(?SETQ <var> <object>) EXPR

SETQ <var> only if it is currently unbound.

(FOREVER {<forms>}) MACRO

Expands to (DO () (NIL) {<forms>}).

(FOR (<var> <init> <final> &OPT <incr>) {<forms>}) MACRO

Expands to:

```
(DO ( (<var> <init> (ADD <var> <incr>)) )
    ( ((GT <var> <final>) NIL) )
    {<forms>} )
```

(MAPCAR <fcn> <list>) EXPR

Applies <fcn> to elements of <list> sucessively, returns a list of the results.

(MAPC <fcn> <list>) EXPR

Like MAPCAR, but no list of results is built.

(MAPCARF <fcn> <list>) EXPR

A version of MAPCAR that works for special forms.

(PLO &REST {<forms>}) EXPR

PRIN3s each of the arguments.

(PL &REST {<forms>}) EXPR

Equivalent to (PROGN (PLO {<forms>}) (TERPRI)).

(EPRINO <object>) EXPR

Equivalent to (PRINO <object> CURRENT-ERR).

(APPEND+ {<forms>}) MACRO

Expands to (APPEND <form-1> (APPEND <form-2>...<form-n>)))).

(CONCAT+ {<forms>}) MACRO

Expands to (CONCAT <form-1> (CONCAT <form-2>...<form-n>)))).

(REMOVE <object> <list>) EXPR

Destructively removes the first element of <list> that is EQ to <object>. Returns the modified list.

(LET* ({(<var> <object>)}) {<forms>}) MACRO

Form of LET that binds <var>s sequentially rather than in parallel. Expands to:

```
(LET ( {(<var> NIL)} )
  { (SETQ <var> <object>) }
  {<forms>}) )
```

(INC <var>) MACRO

Expands to (SETQ <var> (ADD1 <var>)).

(DEC <var>) MACRO

Expands to (SETQ <var> (SUB1 <var>)).

(THROW-ERROR (&REST <list>)) EXPR

Error handler that throws to label ERROR without printing error messages.

(PRINT-THROW-ERROR (&REST <list>)) EXPR

Error handler that prints error message(s) to CURRENT-ERR then throws to label ERROR.

(LINE-EDITED-STREAM <source-fcn> <echo-fcn>) EXPR

Returns a stream suitable for reading that handles the line editing characters ^X and backspace. <Echo-fcn> must support cursor left motion when sent a backspace (^H) character. <Source-fcn> is expected to be non-echoing. The built-in function BUFFERED-CONSOLE-IN is equivalent to:

```
(LINE-EDITED-STREAM CONSOLE-IN CONSOLE-OUT)
```

(MAC <form>) EXPR

Return the <form> (which is assumed to be a macro) before the second evaluation. Useful for debugging macros.

```
     (MAC '(INC FOO)) => (SETQ FOO (ADD1 FOO))
```

(RECORD <name> {<forms>}) FEXPR

Evaluates {<forms>} such that printed output is written to the file <name> as well as to the console.

```
     (RECORD "obl" (MAPC PRINT (OBLIST)))
```

Prints the symbols of the oblist on the screen and to the file OBL.LSP.

(INT <num> { <keyword> <form> }) MACRO

More elegant way to invoke the INTERRUPT function. <Keywords> are any of: AX BX CX DX DI SI BP ES DS FLAGS RETURN.

```
     (INT 21 :ax 10
             :bx (offset str)
             :ds (segment str)
             :return (add ax bx) )
```

expands to

```
     (let ( (int-arg [0 0 nil nil nil nil nil nil 0 nil])
            (int-ans [0 0 nil nil nil nil nil nil nil nil]) )
       (store int-arg 1 10)
       (store int-arg 2 (offset str))
       (store int-arg 9 (segment str))
       (interrupt 21 int-arg int-ans)
       (add (int-ans 1) (int-ans 2)) )
```

Advanced Functions

These functions deal with the LISP system itself and its implementation. Most users will not and should not ever use any of the functions in this section. They are included here for completeness and for users writing extensions to LISP at a very low level.

(OBJADR <object>) SUBR

Returns a number representing the physical location of <object>. Note that for strings and vectors the location returned is that of the descriptor and not the body (see POINTER). Since the internal representation of objects is subject to change in subsequent versions of TLC-LISP, this function should be used with extreme caution. OBJADRs of fixnums and characters are not allowed since these objects are not pointers and have no location.

```
(OBJADR 'FOO) => 345610  ; or something
```

(POINTER <object>) EXPR

Returns a number representing the location of the body of <object>, suitable for EXAMINE/DEPOSIT. This function only works on objects that have bodies in byte space (i.e. strings, vectors, etc.). Be aware that the compaction performed by the garbage collector will move bytes and change pointers. You should not rely on a pointer being valid after any operation that may invoke the GC.

```
(ASCII (EXAMINE (POINTER "abc"))) => \a
```

(SEGMENT <object>) SUBR

Returns the 8086 segment part of the body of a descriptor based object (string, vector, etc.).

(OFFSET <object>) SUBR

Returns the 8086 offset part of the body of a descriptor based object (string, vector, etc.).

(ALLOCATE <symbol>) SUBR

<symbol> is a symbol representing a type. ALLOCATE allocates an object of that type. For example, we could define CONS as:

```
(DE CONS (X Y &AUX (Z (ALLOCATE 'LIST)))
       (RPLACA Z X) (RPLACD Z Y))
```

(PUT-OBJ <object1> <num> <object2>) SUBR

<object2> replaces the <num>-th entry in <object1>. <num> is a (base 0) byte-index into the structure. So CONS could also be defined as:

```
(DE CONS (X Y &AUX (Z (ALLOCATE 'LIST)))
     (PUT-OBJ Z 0 X) (PUT-OBJ Z 2 Y))
```

(GET-OBJ <object> <num>) SUBR

Get the object that is defined at the <num>-th entry in <object>. <num> is the same kind of index that we saw in PUT-OBJ.

(NUMTYPE <num>) SUBR)

<num> is the numeric representation of a TLC-LISP type. NUMTYPE returns the symbolic type corresponding to that number.

```
(NUMTYPE 3)  =>  fexpr
```

(TYPENUM <symbol>) SUBR

<symbol> is the name of a LISP type. TYPENUM returns the internal number corresponding to that type. Useful for dispatching on the type of an objects.

(WHAT <num>) SUBR

Return the object represented by the 16 bit number <num>. Inverse of WHERE.

```
(WHAT 0) => NIL
```

(WHERE <object>) SUBR

Return the internal representation of <object> as a 16 bit number. Inverse of WHAT.

```
(WHERE NIL) => 0
```

(BYTES &OPT FLAG) SUBR

Return a list of the length of all the blocks in byte space that are currently unallocated. If FLAG is supplied, prints the locations of the blocks.

(BSPACE &OPT FLAG) SUBR

Return list of lengths of blocks of bytes known to LISP (allocated or unallocated). If FLAG supplied, prints physical locations of blocks.

The Editor

TLC-LISP includes a screen-oriented editor that acts like a subset of WordStar (trademark of MicroPro International Inc.) This appearance may be altered by changing EDIT.LSP, so those who prefer EMACS-like editors are welcome to customize the code.

The editor is best mastered by using it. We sketch the commands and their more subtle effects below.

For a quick introduction to the editor see the section "The Editor" in Part II of this manual.

Editor Commands

Cursor commands:

```
^S -- left
^D -- right
^E -- up
^X -- down
^A -- left word
^F -- right word
```

Scroll commands:

```
^Z -- scroll up
^W -- scroll down
```

Page commands:

```
^C -- next page
^R -- previous page
```

Jump commands:

```
^QR -- jump to beginning of buffer
^QC -- jump to end of buffer
^QS -- jump to left margin
^QD -- jump to right margin
^QB -- jump to block start
^QK -- jump to block end
```

Deletion commands:

^G -- delete current character
^H, backspace, rub -- delete previous character
^Y -- delete current line
^QY -- delete rest of line

File commands:

^KS -- save buffer contents and return to editor
^KR -- read a text file into the buffer
^KA -- read an atom's value into the buffer
^KF -- change the name of the current file

Block Commands:

^KB -- mark block start
^KK -- mark block end
^KH -- make block visible or invisible
^KC -- copy block to cursor
^KV -- move block to cursor
^KY -- delete the block
^KW -- write the block to a disk file

Escape command:

<esc> -- clear screen but preserve buffer. Execute read-eval-print until (THROW ESC {<form>}) is evaluated then resume editing.

Termination commands:

^KX -- update current file and exit editor.
^KQ -- quit editing without updating file
^KD -- update current file, erase buffer, and prompt for new file.

Evaluation commands:

^JJ -- evaluate one expression from cursor
^KJ -- evaluate buffer, read buffer into system

Editor-related Functions

(EDIT {<var or str>}) FEXPR

<Str>s are assumed to be filenames. The specified file(s) are read into the edit buffer. <var>s are assumed to be existing functions or data, the specified values are pretty-printed into the edit buffer. If only one file is specified then it becomes the current file and is updated before the editor is exited. If more than one file is specified then the editor will prompt for a new file name before updating the file. EDIT is protected by UNWIND-PROTECT to insure that changed files are updated. The editor expands tabs to the next column eight multiple when files are read and compresses spaces to tabs where possible when files are written (see the following functions).

(ED-READLINE &OPT <stream>) SUBR

A special case of READLINE, described in the section on Read Functions.

(ED-PRINTLINE <str> &OPT <stream>) SUBR

Print <str> as a line of text to <stream> followed by carriage return and linefeed. Spaces outside literal strings are compressed to tabs when possible. The algorithm stops compressing when a double quote (") is reached and resumes at the next double quote. Spaces and double quotes immediately following the literal character character (\) are ignored. ED-PRINTLINE cannot handle an odd number of double quotes (not counting literal characters) in a string and will generate a STRING-FORMAT error if an unmatched double quote occurs. ED-PRINTLINE is not smart enough to recognize comments, thus an error will be generated by a line like the string:

```
" ; this is a double quote (") "
```

ED-PRINTLINE uses the system buffer to build the compressed string before printing to <stream>. <Streams> with user defined functions must be used with great caution here because they might use the system buffer (see BUFFER), thus corrupting the printed string. ED-PRINTLINE is guaranteed to work with streams that use subrs and files.

If this behavior is incompatible with your particular application then you may replace the occurrences of ED-PRINTLINE in the file EDIT.LSP with PRINT3. This will prevent space to tab compression and will result in files increasing in size by 10 to 20 percent (typically).

Turtle Graphics

Turtle graphics similar to TLC-Logo are supported by TLC-LISP. The hardware supported so far includes the IBM PC and the SCION Microangelo. Many of these functions operate on the "current turtle". The current turtle is defined to be the value of the symbol TURTLE.

A turtle object contains the following properties, most of which may be examined or changed by the functions described in the following sections.

Name
Horizontal Position
Vertical Position
Heading
Pen State
Ink Color
Turtle Shape
Visibility State
Local Storage

About Turtle Shapes

Turtle shapes are implemented as strings, the characters of the string are interpreted as unsigned eight bit numbers. The string must be of the following format:

Char	Meaning
1	number of bytes (not pixels) in X direction
2	number of bytes in Y direction
3 thru N	actual bytes to put in display memory

See the file TURTLE.LSP for examples.

About Coordinate Systems

World coordinates reflect the maximum values that the turtle's horizontal and vertical position may take. We currently implement positions as 16 bit signed integers so that the range is +/- 32767. Since turtles use 16 bit signed arithmetic internally, the maximum distance a turtle may move during the execution of a single command is 32767 units even though the world coordinate system spans 65536 units. Distances greater than 32767 units will result in a NUMBER-OUT-OF-RANGE error.

Screen coordinates refer to the visible pixels on the screen measured from the lower left corner (0, 0). The range of screen coordinates is hardware dependent. The mapping of world coordinates to screen coordinates is affected by the viewport (VP) function, described below.

In either coordinate system the X coordinate increases left to right and the Y coordinate increases bottom to top.

Special Symbol Values

The turtle graphics system expects the following symbols to have certain values:

TURTLE

The current turtle is defined to be the value of this symbol.

TURTLE-DEFAULTS

Vector of turtle shapes to use when (SHAPE nil) is in effect. The length may range from 2 to 255.

TURTLE-SHAPES

Vector of turtle shapes to use when (SHAPE <num>) is in effect. The shape used is the <num>-th element of TURTLE-SHAPES which is expected to be a string (see About Turtle Shapes above).

Turtle Functions

(HATCH <var>) SUBR

Creates a new turtle identical to the current turtle except that its name is <var>.

(TINIT) SUBR

Performs internal initialization for the turtle system. Initializes the turtle family list and hatches the initial turtle named STUDS. Destroys any existing turtles, but preserves the values of TURTLE-SHAPES and TURTLE-DEFAULTS. May perform hardware initialization.

(SHAPE &OPT nil or <num> or <str>) SUBR

Return or set the shape of the current turtle. See About Turtle Shapes above. The following arguments are accepted:

> (SHAPE nil) uses the vector of strings TURTLE-DEFAULTS. If the turtle's heading is between 0 and 30 degrees then the shape at index one is used, 30 to 60 uses the shape at index 2, and so on for twelve (possibly) different shapes.
>
> (SHAPE <num>) uses the number as an index into the vector TURTLE-SHAPES. The shape is independent of heading.
>
> (SHAPE <string>) uses the string as a shape.

(POS &OPT <num1> <num2>) SUBR

Return or set the position of the current turtle.

```
(POS) => (0 0)
(YPOS 100)
(POS) => (0 100)
(POS 10 20)
(YPOS) => 20
```

(VP &OPT <num1> <num2>) SUBR

Return or set the viewport. The viewport defines the world coordinates of the center of the screen. The world coordinates range from -32K to 32K in both axis.

> (VP 0 0) puts the origin at the center of the screen. On the IBM PC this results in visible world coordinates of -170 to 170 horizontal and 100 to -100 vertical.
>
> (VP 170 100) puts the origin at the lower left corner of the screen. On the IBM PC this results in visible world coordinates of 0 to 340 horizontal and 0 to 200 vertical.

(XPOS &OPT <num>) SUBR

Return or set the X coordinate of the current turtle.

```
(POS) => (100 100)
(XPOS) => 100
(XPOS 150)              ; turtle moves
(POS) => (150 100)
```

(YPOS &OPT <num>) SUBR

Return or set the Y coordinate of the current turtle.

```
(POS) => (100 100)
(YPOS) => 100
(YPOS 150)              ; turtle moves
(POS) => (100 150)
```

(HD &OPT <num>) SUBR

Return or set the heading (direction) of the current turtle in degrees. <Num> is reduced to the range 0 to 359 degrees.

```
(HD) => 90       ; turtle is facing "up"
(HD 180)         ; turtle rotates 90 degrees counter-clockwise
                 ; now faces "left"
(HD) => 180
```

(FD <num>) SUBR

Move the current turtle forward <num> pixels, maintaining the current heading.

```
(POS 0 0)         ; move to center
(HD 45)
(FD 100)
(POS) => (170 170)
```

(VIS &OPT <num>) SUBR

Return or set the visibility status of the current turtle.

(VIS 0) means the turtle shape is not displayed.

(VIS 1) means the turtle shape is displayed. The actual shape is determined by the SHAPE function.

(INK &OPT <num>) SUBR

Return or set the ink color of the current turtle.

(PAPER &OPT <num>) SUBR

Return or set the paper (background) color.

(PEN &OPT <num>) SUBR

Return or set the state of the current turtle's pen. Three values for <num> are valid:

0 The pen is up.

1 The pen is down, and movement will draw a line in the current ink color.

2 The pen will erase. Turtle motion will draw a line using the current background color.

(TM &OPT <num>) SUBR

Return or set the turtle mode. The following values are accepted:

(TM 0) is window mode, turtles can range over the entire range of world coordinates (+/- 32K). If the turtle is on a visible portion of the screen (see VP) then it is displayed.

(TM 1) is fence mode. Turtles that attempt to move off the visible screen will be stopped at the edge.

(TM 2) is fence error mode, similar to mode 1 except a STOPPED-AT-FENCE error is also generated when the turtle hits the fence. This mode can be used in conjunction with CATCH to define behaviors like "wrap" or "rebound".

(TILE &OPT nil or <num> or <str>) SUBR

Return or set the drawing mode of the current turtle. The following values are accepted:

(TILE NIL) the turtle will draw lines as it moves if the pen is down.

(TILE <num>) the turtle will leave "tiles" as it moves (if the pen is down). The tile shape is from the <num>-th element of the vector TURTLE-SHAPES.

(TILE <str>) the turtle will leave "tiles" as it moves (if the pen is down). The tile shape is the <str> (see *About Turtle Shapes* above).

(MY &OPT <object>) SUBR

Return or set the local data field of the current turtle. The MY slot is available for use by application programs.

(TF) SUBR

TF (Turtle Family) returns the list of turtles known to the system.

(CS) SUBR

CS (Clear Screen) clears the screen of turtle tracks. Visible turtle shapes within the bounds of the screen are redisplayed.

IBM Personal Computer Functions

These functions are specific to the IBM PC hardware. MSDOS users please note: MSDOS version 2 and ANSI.SYS are recommended for proper operation of the editor and the split screen turtle graphics functions due to the use of escape sequences to change colors and screen modes.

Editor Support

These functions make use of variables defined in the editor code found in the file EDIT.LSP

(IBM-UPDATE-LINE <num> &OPT <str>) SUBR

If <str> is not supplied then the element of ED-BUFFER at index, (ADD YOFFSET <num>) is used. If the vector element is a string then this function moves the string to screen display memory, line <num>. XOFFSET characters are skipped at the beginning of the string. If the string is shorter than the screen-width, then space characters are used for the remaining locations in screen memory's line. If the vector element is a character, then a blank line is displayed. See ED-INIT and ATTRIBUTE.

(IBM-UPDATE-SCREEN) SUBR

Move the strings from elements YOFFSET to (ADD YOFFSET 23) from the vector ED-BUFFER to screen display memory. Similar to IBM-UPDATE-LINE.

(ED-FAST <t or nil>) SUBR

If the argument is NIL then ED-FAST only allows the editor support subrs IBM-UPDATE-SCREEN and IBM-UPDATE-LINE to alter display memory during vertical retrace; this results in a cleaner (but slower) screen update. If the argument is non-NIL then display is updated faster but with interference. The default value is equivalent to (ED-FAST NIL).

(ED-INIT &OPT <num> <symbol>) SUBR

Initialize internal variables for the editor support subrs IBM-UPDATE-SCREEN and IBM-UPDATE-LINE. <Num> is the line width in characters (either 40 or 80). <Mode> is either the symbol BW or COLOR. BW selects the IBM Monochrome Display and Printer Adapter, 6845 CRT controller at ports 03B0H through 03BFH and display memory at 0B000H. COLOR selects the IBM Color/Graphics Monitor Adapter, 6845 at ports 03D0H through 03DFH and display memory at 0B800H. If no arguments are supplied then ED-INIT uses the equipment interrupt (11H) to determine the initial display configuration as determined by the switch settings on the main circuit board. Failure to execute ED-INIT before using the editor support subrs results in an ED-INIT error.

(ATTRIBUTE &OPT <num>) SUBR

Return or set the attribute value of the characters displayed by IBM-UPDATE-SCREEN and IBM-UPDATE-LINE. The default value is 70H. The eight bit number has the following definition:

Color Display

Bit	Affects
7	Blinking characters
6	Red background
5	Green background
4	Blue background
3	Character intensity
2	Red character
1	Green character
0	Blue character

Thus the value 00011111B (1FH, 31 decimal) produces high intensity white characters on a blue background. The default setting 01110000B (70H, 112 decimal) produces black characters on a white background.

Monochrome Display

Bits	Affect
7	Blinking characters
6-4	All set for white background, all reset for black
3	Character intensity
2-0	All set for white characters, all reset for black

Thus the default value 01110000B produces black characters on a white background.

(GRBASE &OPT <num>) SUBR

Return or set the beginning of graphics memory. <Num> is the segment part of an 8086 double word address. The default is 0B800H for the IBM Color/Graphics Display Adapter.

BIOS Interrupt Functions

(V-WINDOW &OPT <num>) SUBR

Return or set the size of the text window (in character lines) below the graphics display area. Allows the line clipping routines to clip lines that would appear in the text area. Also prevents CS (clearscreen) from erasing the text area. Use zero if no text window is desired. This function is only used to set up internal limits for the graphics routines, it does not create windows. Most users will use the SS function (described below) which calls V-WINDOW internally.

(V-MODE &OPT <num>) SUBR

Return or set the video display mode which determines the density (high or low resolution) and the type (text or graphics) of the display. Allows line drawing routines to correctly clip lines that are partially visible. Most users will use the SS and TS functions (described below) which call V-MODE internally. Acceptable values for <num> are as follows:

Value	Meaning
0	40x25 black and white text
1	40x25 color text
2	80x25 black and white text
3	80x25 color text
4	320x200 color graphics
5	320x200 black and white graphics
6	640x200 black and white graphics

When setting the video mode, BIOS interrupt 10H and command 0 is used. When reading the mode, the last value set is assumed to be valid. This may not be the case if the operating system has used interrupt 10H internally. For CP/M, you must send the proper escape sequence to the console when changing the video mode to insure that TLC-LISP and CP/M agree on the video mode. A related problem occurs when using V-MODE to select a graphics mode; in this case, the time-of-day clock will continue to be updated at the bottom of the screen unless you invoke a graphics mode escape sequence, making CP/M shut off the clock. See the code in WIN.LSP for examples.

(V-CURSOR &OPT <num1> <num2> <num3>) SUBR

This function uses the IBM ROM BIOS interrupt routine (interrupt 10H commands 2 and 3) and may confuse the operating system as to the screen cursor location. Use with care. The following argument combinations are accepted:

(V-CURSOR) returns the cursor position on video page zero as a list of numbers (column row).

(V-CURSOR <num1>) returns the cursor position on page <num1>.

(V-CURSOR <num1> <num2>) sets the cursor on page zero to column <num1> row <num2>.

(V-CURSOR <num1> <num2> <num3>) sets the cursor on page <num3> to column <num1> row <num2>.

(V-PAGE &OPT <num>) SUBR

Return or set the current video page. Uses the IBM ROM BIOS interrupt 10 command 5 to set the page. Uses an internal copy of the last (V-PAGE <num>) call to return the current page, thus the return value may be incorrect if this interrupt is used directly by the operating system. The initial page is assumed to be zero.

(V-PUTCHAR <char> <num1> <num2> <num3>) SUBR

Put <char> at column <num1> row <num2> with attribute <num3> on the last page selected by V-PAGE. Internally calls (V-CURSOR <num1> <num2>). Uses the IBM ROM BIOS interrupt (interrupt 10H command 9). May confuse the operating system as to the current location of the cursor. Use with care.

(V-PALETTE <num1> <num2>) SUBR

Sets the color value of color palette <num1> to color <num2>. Uses the IBM ROM BIOS interrupt (interrupt 10H command 11).

(V-SCROLL-UP <num1> <num2> <num3> <num4> <num5> <num6>) LSUBR

(V-SCROLL-DOWN <num1> <num2> <num3> <num4> <num5> <num6>) LSUBR

Scroll a screen area beginning at column <num1> row <num2> (upper left corner) and ending at column <num3> row <num4> (lower right corner) by <num5> lines using the fill character <num6>. A count (<num5>) of zero will erase the window. Uses ROM interrupt 10H commands 6 or 7.

(V-INSERTCHAR <char> <num1> <num2>) SUBR

Inserts <char> at column <num1> row <num2> in graphics memory. Moves the remaining characters in the line one position to the right. The last character in the line is lost. Directly manipulates graphics memory using GRBASE and the screen dimensions set by V-MODE. Useful for editors.

Graphics Functions

(TS) EXPR

Text screen mode, 80 by 24 lines of text.

(SS &OPT <num>) EXPR

Split screen, graphics and <num> lines of 40 column text display. If <num> is absent, the last value given to SS is used.

(BG <num>) EXPR

Set background color, <num> from 0 to 15.

(FG <num>) EXPR

Set foreground color, <num> from 0 to 15. In split screen mode only 4 foreground colors are possible and <num> modulo 4 is used.

Function Index for Reference Manual

T L C - L I S P   D O C U M E N T A T I O N

A P P E N D I X   I

The Pseudo Code Module

The P-code Module: Compiler, Assembler, and Disassembler

This module contains the compiled files (*.P) and source files (*.LSP) for the TLC-LISP compiler, assembler and disassembler. Please respect the copyright of the information. Below is a short description of the contents of each source file:

COMPDECL.LSP
Collection of declarations used throughout the compiler

COMPMACS.LSP
Collection of macro definitions used similarly.

COMP.LSP
The top-level compiler drivers.

COMPARGS.LSP
The formal parameter processor.

COMPDO.LSP
The Do-compiler.

COMP2.LSP
The code generators.

COMPFILE.LSP
Thie file handling portion--where to put the code.

OPT.LSP
A simple code optimizer--not very clever, but effective.

LAP.LSP
The assembler to generate runnable code.

DISPCODE.LSP
The disassembler so you can see what the compiler did to you.

P-code versions of these files are in the *.P counterparts.

To install the P-code Module in your system see the file named

INSTALL.CMP

on your P-code Module Disk.

General Notes On the TLC P-machine

The P-machine is a straightforward stack-oriented architecture with primitive operations based on the requirements of LISP-like languages. These operations include arithmetic and basic list-processing. Of more interest are the operations that make p-code functions compatible with interpreted TLC-LISP, allowing the mixture of interpreted, P-code, and primitive functions. These operations involve the access and updating of variables, either through the interpreter's value-cells, or locally through slots in stack-frames.

A subr is the application of a built in function, usually with the same name as the opcode. A binary subr pops two objects from the stack and pushes one result. A unary subr pops one argument object from the stack and pushes one result.

The best way to get a feeling for the machine is to examine the compiler output. This can be done in two ways:

* Use DIS to disassemble p-code in memory. For example

```
     (DIS mapcar)     might give

17AB:09E4  2 0 2 0
17AB:09E8  fget2
17AB:09E9  nil
17AB:09EA  fset0
17AB:09EB  dupl
17AB:09EC  fsetl
17AB:09ED  not
17AB:09EE  iffl 09F6
17AB:09F1  fget0
17AB:09F2  ncl freverse
17AB:09F5  ret
17AB:09F6  fget3
17AB:09F7  fgetl
17AB:09F8  car
17AB:09F9  funcall 2
17AB:09FB  fget0
17AB:09FC  cons
17AB:09FD  dupl
17AB:09FE  fset0
17AB:09FF  disc
17AB:0A00  fgetl
17AB:0A01  cdr
17AB:0A02  fget0
17AB:0A03  jmpl 09EA
```

all of which is pretty meaningless without knowing the Op-codes.

* You can also use the LIST option in the compiler to examine compiled code in source form.

For example, (COMPILE <source> (LIST <file name>)) will deposit a LAP (Lisp Assembly Program) form of the compiled code on the designated file.

Note on Terminology

TOS means the object at the top of the stack. TOS-1 means the next object after TOS on the stack.

General Caution

The current P-machine instruction set is arbitrary, irregular, and non-optimal. A new machine will be forthcoming, along with the necessary conversion programs.

Opcodes

ADD
Binary subr, replacing TOS with the sum of TOS and TOS-1.

ADD1
Unary subr. TOS gets TOS+1

APPEND
Binary subr. TOS gets (append TOS-1 TOS)

ARG <symbol>
Argument bind. TOS is popped and shallow bound to <symbol>. This handles the compilation of non-local variables in a dynamically scoped LISP. But see FFGET and FFSET.

ASRT <type>
Assert. Generate arg-wrong-type error if TOS is not <type>. Does not pop TOS. This is a static type-check op-code.

ASSOC
Binary subr. TOS-1 is used as an index into the list found in TOS, and an ASSOC is performed. An historical artifact of limited utility.

BIND <symbol>
Save the current binding of <symbol> in the shallow binding stack.

CAR
Unary subr. TOS gets (CAR TOS).

CARCDR
TOS is popped. The cdr is pushed then the car is pushed.

CATCH <symbol> <word>
Erect a catch frame. The appropriate throw will cause execution to resume at relative offset <word>.

CDR
Unary subr.

CONS
Binary subr. TOS gets (CONS TOS-1 TOS)

CONST <object>
Constant. Push the <object>.

DISC
Discard. Pop TOS.

DIV
Binary subr. (DIV TOS-1 TOS)

DUPL
Duplicate the top of the stack, i.e push TOS.

EQ
Binary subr.

ESC
Escape to machine code. 8086 code is expected to follow.

EXCH
Exchange. Swap TOS and TOS-1.

FFGET <frame> <num>
Push the <num>-th local object from the <frame>-th previous frame on the stack. This instruction (and its partner FFSET) are of particular importance for a language like Scheme, that depends on lexical scoping. The current TLC-LISP86 compiler does not use these instructions, but depends on &SPECIAL to compile references through ARG, etc.   See FFSET.

FGET <num>
Push the <num>-th local object. If <num> is from 0 to 7 then a special one byte form of the opcode may be used instead of the two byte form. The current compiler suffers from a design error, mapping the single byte forms onto their complement. For example, in a ternary function (FGET 0) maps onto (FGET3) and (FGET 1) maps onto (FGET2). The system is internally consistent so that the correct thing happens. This idiocy will be fixed in a revised P-machine. See FSET.

FIX <num>
Push the fixnum <num> which is embedded in the opcode and may range from 0 to 7.

FLUSH <num>
Pop TOS. Remove the next <num> objects from the stack. Restore TOS.

FFSET <frame> <num>
Pop the stack into the <num>-th local object from the <frame>-th previous frame. See FFGET.

FSET <num>
Pop the TOS into the <num>-th local object slot on the stack. If <num> is from 0 to 7 then a special one byte form of the opcode may be used instead of the two byte form. See the remark in the description of FGET.

GE
Binary subr. TOS gets (GE TOS-1 TOS).

GT
Binary subr. TOS gets (GT TOS-1 TOS).

IFF <word>
IFFS <byte>
IFFL <dword>
If false. Pop TOS and jump relative if nil.

IFFN <word>
IFNL <dword>
IFNS <byte>
If false no pop. Pop TOS. If nil then jump relative else push it back.

IFT <word>
IFTL <dword>
IFTS <byte>
If true. Pop TOS and jump if not nil.

INT
Pseudo-machine breakpoint. Evaluates the form (apply pm-break pc sp) where pc is the p-machine program counter and sp is the stack pointer. The arguments pc and sp are passed as number objects suitable for examine/deposit. The function pm-break is supplied by the user.

JBOUND

Jump if bound. Pop TOS. Don't jump if unbound else push it back and jump. This is used to compile code for optional arguments. If the corresponding actual parameter is unbound, then we know that a value was not supplied and the optional value should apply for the position.

JMPL <dword>
JMPS <byte>
JMP <word>

Unconditional relative jump.

JTYPE <type> <word>

If type of TOS is <type> then jump. TOS is preserved.

LE

Binary subr. (LE TOS-1 TOS).

LENGTH

Unary subr. (LENGTH TOS).

LIST <count>

Applies the list subr to the <count> arguments on the stack, popping these arguments and pushing the resulting list object. The arity <count> is embedded in the opcode and may range from 1 to 8.

LT

Binary subr. (LT TOS-1 TOS).

MARK

Save the current name stack pointer. The value pushed is not a valid Lisp object. See UNMARK.

MINUSP

Unary subr. (MINUSP TOS).

MUL

Binary subr. (MUL TOS-1 TOS).

NC <arity> <symbol>

Named call. Pop the arguments and apply the value of <symbol>. Equivalent to (apply <symbol> (list <args...>)) but no list is allocated. A short form of the opcode with embedded arity may be used if the arity is less than 8.

NCONC

Binary subr. (NCONC TOS-1 TOS).

NIL

Push the NIL object.

NOT

Unary subr. (NOT TOS).

NTAIL <arity> <symbol>

Named pcode tail recursion elimination. For symbols that are pcode only. Reuse the current stack frame.

NTH

Binary subr.

PC <arity> <word>

Pcode call. <Word> is relative. A short form of the opcode with embedded arity may be used if the arity is less than 8.

POPVAL <symbol>

Setq <symbol> to TOS. Pop TOS.

POPW <offset>

Pop word. Pop the TOS into the specified <offset> in the interpreters data segment. Ugly instruction.

PTAIL <arity> <word>

Pcode tail recursion elimination. Reuse the current stack frame.

PUSHVAL <symbol>

Push the current value of <symbol>.

PUSHW <offset>

Push word. Push the object at the specified <offset> in the interpreter data segment.

REPINIT <label>

Check the numeric object on the top of stack and jump to label with TOS NIL if zero or negative. If the value is greater than zero, then a 32 bit integer representation replaces the object on the stack. Used to compile REP.

REPJUMP <label>

Remove the value on the top of the stack, decrement the 32 bit integer now on TOS and jump to label if the result is non-zero. Otherwise replace the 32 bit integer with the old TOS value and fall through.

For example,   (rep n (foo)) compiles to:

```
     (PUSHVAL n)
     (REPINIT end)
loop (NCO foo)
     (REPJUMP loop)
end
```

RET
Return to the caller of the pcode function, using TOS as the returned value.

RFIELD <offset>
Read Field. Extract the object located <offset> bytes into the object on the top of the stack. The new object replaces the original object on the stack.

RPLACA
Binary subr implementing (RPLACA TOS-1 TOS)

RPLACD
Binary subr. (RPLACD TOS-1 TOS)

SET
Binary subr. (SET TOS-1 TOS)

SOT <byte>
Smash the <byte>-th location down the stack with TOS. Pop TOS. SOT 0 is equivalent to DISC.

STRING <count>
Applies the string subr to the <count> arguments on the stack, popping these arguments and pushing the resulting string object. The arity <count> is embedded in the opcode and may range from 1 to 8.

SUB
Binary subr.

SUB1
Unary subr.

THROW <symbol>
Equivalent to (throw <symbol> TOS).

TOS <byte>
Push the <byte>-th object down the stack. TOS 0 is equivalent to DUPL.

TYPE
Unary subr.

UNBIND
Unary subr.

UNBOUND
Push the special object UNBOUND.

UNCATCH
Pop TOS. Remove the catch frame erected by CATCH. Restore TOS.

UNMARK
Restore the name stack pointer

VREF
Binary subr. TOS gets (VREF TOS-1 TOS)

VSET
Three argument subr. TOS gets (STORE TOS-2 TOS-1 TOS)

WFIELD <offset>
Smash the object located <offset> bytes into the object at TOS-1 with TOS. The stack is popped once.

ZEROP
Unary subr.   TOS gets (ZEROP TOS)

What Is A Compiler?

We will not attempt to give a whole course on compiler construction here. We will ignore the syntactic problems of parsing, symbol table construction, and internal code representation; those issues are already solved for us by LISP. Rather, we'll concentrate on the semantic issues: the general definition of a compiler, the issue of code generation, and finally the issue of optimization and performance. For a deeper discussion of all these issues see "Anatomy of LISP", or "Structure and Interpretation of Computer Programs".

In a phrase, a compiler translates a program in one language into a program in a second language such that the execution of the second program has the same effect as the execution of the first program. To make the problem non-trivial, we assume that the second language is not the same as the first. Specifically, we will transform TLC-LISP86 code into TLC-LISP P-code. So that:

```
(EVAL <expression>)  =  (P-run (COMPEXP <expression>))
```

where P-run represents the (internal) execution device to sequence through and execute the compiled P-code. COMPEXP is an (internal) piece of the Compiler.

We must also reconcile the result of the P-machine with the expectation of EVAL, defining the value of a P-code computation to be the object located on the top of the stack after the sequence of instructions has been completed.

The task of the compiler is thus to translate each TLC-LISP expression into such a seqeunce. We'll begin with a simple but representative set of LISP constructs-- constants, function applications, conditional expressions, and variable references.

```
constants  -- 1, 'A, [1 2 4]

function calls -- (foo 1 2 3)

control constructs  -- (if (foo 1 2 3) 4 5)

variable references -- (de foo (x y z)  (cons x y) ... )
```

A quick examination of the P-machine shows that:

* constants are easy -- (CONST 1) will fill the bill for the constant 1.

* function calls are easy --

```
(CONST 1) ; push 1
(CONST 2) ; push 2
(CONST 3) ; push 3
(NCALL 3 foo) ; call foo
```

will handle (FOO 1 2 3)

So more generally, (COMPEXP (<function> <argl> ... <argn>) is

```
(APPEND+   (COMPEXP <argl>)
           (COMPEXP <arg2>
              . . .
           (COMPEXP <argn>)
           (LIST (LIST 'NCALL n <function>) ))
```

* control constructs are easy --

```
      (CONST 1)
      (CONST 2)
      (CONST 3)
      (NCALL 3 foo)
      (IFF xx1)
      (CONST 4)
      (JMP xx2)
xx1   (CONST 5)
xx2
```

will compile (IF (FOO 1 2 3) 4 5)

So (COMPEXP (IF <pred> <a> <b>)) is

```
(APPEND+  (COMPEXP <pred>)
          (LIST (LIST 'IFF <label-1>))
          (COMPEXP <a>)
          (LIST (LIST 'JMP <label-2>))
     (LIST <label-1>)
          (COMPEXP <b>)
     (LIST <label-2>)
```

* Variable references are interesting -- Since the most interesting variable references involve access to actual parameters within a function, we need to know that the TLC-LISP86 interpreter builds a frame that contains the actual parameters before it enters the P-code translation of the function. Therefore a reference to an actual parameter is indicated by a zero-based reference to a slot in the latest frame.

Thus recalling the P-Code FGET, we see:

(FGET 0) gets the first parameter, (FGET 1), the second. So

(DE FOO (X Y Z) (BAR (CONS X Y))) compiles as;

```
(DEFPCODE FOO (X Y Z)
          3 0 0 0         ; arity info--checked on entry
          (FGET 0)        ; X
          (ARG X)         ; update X's symbol
          (FGET 1)        : Y
          (ARG Y)
          (FGET 2)        ; Z
          (ARG Z)
          (PUSHVAL X)     ; X
          (PUSHVAL Y)     ; Y
          (CONS)          ; A primitive op-code, result to TOS
          (NCALL 1 BAR)   ; general (named) call to BAR
          (RET)))         ; Return to caller
```

After these symbolic descriptions are translated by LAP into bytes and installed as the definition of FOO, anyone can call FOO, and BAR can be arbitrary executable TLC-LISP86 code.

We can now sketch the extensions for the compilation of a definition:

(COMPILE '(DE <name> <formals> <body>)) is something like:

```
       (CONCAT 'DEFPCODE
               (CONCAT <name>
                       (APPEND (ARITY-LIST <formals>)
                               (COMPILE-EXP <body> <formals>))))
```

where we assume ARITY-LIST can generate appropriate entries for the prolog, and COMPILE-EXP can compile the <body> to reference the appropriate values of formal parameters. COMPILE-EXP is a generalized form of COMPEXP.

Since variable references can occur within arbitrary sub-expressions in the body of the compiler, COMPILE-EXP must pass the formal parameter information, so that a variable reference to the i-th formal parameter becomes an (FGET/FSET i). For example, to generate the appropriate index we might use:

```
(de make-ref (name formal-list &OPT (count 0))
   (cond ((null formal-list)) (error "non-local reference"))
         ((eq name (first formal-list)) count)
         (t (make-ref name (rest formal-list) (add1 count)))))
```

Seems simple enough. The only complexity involves the error condition "non-local reference". This corresponds to LISP's use of dynamic scoping. For example, if BAR uses X, Y, or Z free within its call on FOO, then we must be sure that the binding made on entrance to FOO is available to BAR. Since the frame-mechanism (called "deep binding") is not used by the TLC-LISP interpreter (it uses "shallow binding"), we included the ARG-operation in the P-machine to force the parameter into BAR's view (or into the view of anyone called by BAR).

The use of ARG makes the compiled code have the same functionality as its interpreted precursor. That's the good news. On the other hand, it takes time to update the value-cells. So if no one cares about those values, we can maintain the values strictly within the P-machine's domain by using the stack frame and the FGET's as the repository of the values. So if no one needed X, Y, or Z, we compile as:

```
(DEFPCODE FOO (X Y Z)
          3 0 0 0         ; arity info--checked on entry
          (FGET 0)        ; X
          (FGET 1)        ; Y
          (CONS)          ; A primitive op-code, result to TOS
          (NCALL 1 BAR)   ; general (named) call to BAR
          (RET))))        ; Return to caller
```

The compromise position--that some parameters are needed dynamically and some not--is handled by declaring the dynamic variables as "special". Thus:

```
(DE FOO (X Y Z &SPL Z) (BAR (CONS X Y))) compiles as;
```

```
(DEFPCODE FOO (X Y Z)
          3 0 0 0         ; arity info
          (FGET 3)
          (ARG Z)         ; stuff the value-cell
          (FGET 0)        ; X
          (FGET 1)        ; Y
          (CONS)          ; A primitive op-code, reslut to TOS
          (NCALL 1 BAR)   ; general (named) call to BAR
          (RET) )         ; Return to caller
```

... and the correct value of Z will be available within BAR. X and Y will be inaccurate if BAR tries to access non-local versions of them.

This mixture of frame-based and value-cell-based parameter maintenance allows us to muddle the boundary between the two machines--the inner P-machine, and the outer LISP-machine. This "muddling" is useful for efficiency reasons only. It is important to keep a clear understanding of the issues at the boundaries between each of the TLC machines.

Some General Comments About Compilation

Of course the full compiler is more complex, but the last section is an accurate picture of the kernel ideas. So why all the fuss about compilers in the traditional computer science view of the world?

It is our belief that compilers have held too high a position in the pecking order of Computer Science. As the last section illustrates, they can best be understood after one has a good grasp of the semantics of the source language, and that understanding is much more easily developed and debugged using an interpreter as the operational model of the semantics.

Compilers explain nothing; they only make what exists execute faster by translating the expressions in the source language into instructions in another language called the target language. Another interpreter still has to execute the resulting translation, or perhaps the code can again be (micro) compiled into yet a more detailed machine. At each stage the next level of machine gets more specific and more complex, making the linkage between the final executable code and the initial high-level expression tenuous at best.

In many ways, compilers just make the programming problem harder. At the semantic level, the detailed dissection that compilation implies gives little insight into the intended meaning of the original expression. From the practical side of things, the compiler model of computation implies that high-level expression be completely specified before execution can begin (for otherwise compilation could not proceed). Particularly when learning a language, this requirement for static oompleteness becomes bothersome. When this requirement for completeness couples with a static type structure, then interactive programming becomes nearly impossible. But of course, advocates of compiler-based, strongly typed languages see interactive programming as a tool of the devil anyway.

The LISP tradition dictates that the operational semantics of a language be given by a meta-circular interpreter--one written in the language in question, and which manipulates encodings of programs written in that language. This is an old trick of computation theory, demonstrating the universality of a notation by displaying a program that can simulate any program in the language. Two constructs are required: first, the demonstration of a mapping that takes any program into a data structure of that language. This is LISP's famous "program-as-data" trick. Second, one must demonstrate a program that will manipulate instances of that representation, simulating the execution of the pre-image of the encoding. That program is the EVAL function of LISP.

One can control the detail of the simulation by tuning the level of the implementation structures that are utilized. For example, the simplest of simulations tends to be a recursive description, leaving the implementation of recursion beyond the pale. However, one can also made the details of recursion explicit, thereby demonstrating a non-recursive simulation. The structures involved in such a model are more like the machine-level details of the compiler model, but the expressions are still those of the high-level language, not some target language.

The real contribution of compiler technology is to tradeoff generality of expression and flexibility of execution, for speed of execution. Regardless of the speed, the overriding requirement on compiler technology is that the semantics of any valid program must be preserved: interpreted results must agree with compiled results.

That means, whatever the target machine might be, the result of executing (interpreting) the original program must be the same as the response we get when executing the compiled code on the target machine.

The situation becomes more complex when, as is the case for TLC-LISP, we have several target machines, and we wish to be able to pass back and forth between compiled code and interpreted oode.

In TLC-LISP, there are three choices for target language.

* P-code -- Pseudo-code for a TLC-designed stack-oriented software defined machine. The P-machine is the default target for the compiler.

* Code -- a "macro-expanded" versions of P-code, replacing P-machine instructions with sequences of 8086 instructions. Peep-hole optimizations are performed, subject to the idiosyncracies of the 8086.

* Asm -- raw 8086 code. All bets are off unless the user maintains semantic compatibility with the desires of the calling routine. Asm objects are used for the most speed-sensitive or hardware-dependent applications.

Currently P-code and Code objects are the target languages for the compiler, but all three types of code are recognized and handled by the interpreter.

We think of each of these levels as representing a machine, nested within its immediately surrounding machine with LISP at the outer level. The code from the previous layer is passed to the inner layer for execution. As we go down in layers, we get closer to the 8086-based hardware machine (which, of course, is also interpreting instructions).

There are space/speed trade-offs between these different levels. At each level, certain compatibility conventions must be adhered to so that information may flow cleanly between machines. Specifically:

* Between interpreted LISP and P-code. We must solve the parameter-passing and value-return problem. This involves knowledge of the structures that TLC-LISP86 employs between function calls and returns. Specifically, parameters are passed through a LISP frame and controlled by the P-machine in its local control structure, the P-stack.

  Of course we must describe a complete translation of TLC-LISP code into a collection of P-code. No modification are made to TLC-LISP data--The P-machine still deals with LISP objects.

* Between P-code and Code. The parameter-passing situation is that of P-code, because we are able to map P-machine's stack onto the hardware stack of the 8086 architecture. We can still communicate with the LISP frame information, but now we must know the hardware layout of the frame and use 8086 instructions to access those fields. Furthermore, at the Code Level we must transform LISP data objects into physical addresses of the 8086; the 8086 doesn't understand LISP's notion of objects, it only recognizes bits, bytes, and words. Even things as simple as integers have a different representation in TLC-LISP than that supported by the hardware. This is the Object-to-address problem.

* Between Code and Asm. Here we dispense with the niceties of TLC-LISP's frame structure and deal directly with 8086 registers for passing paramters and values. Specific register conventions are given for passing arguments, and a specific register is designated as the repository the object computed by the function.

All this care is required because we must be able to intermix interpreted LISP, P-code, Code, and Asm objects --not just statically, but dynamically. This means that we must be able to compile and install functions on the fly, again with no change in program semantics. These are reasonably agressive requirements.

The strategy adopted by TLC involved the design of an intermediate code machine --the P-machine-- and in parallel the design and implementation of a compiler that would take TLC-LISP constructs into sequences of P-code. The resulting code is then executed by the P-machine. The P-code is more compact and executes more rapidly than the direct interpretation of TLC-LISP. This initial system was designed on (and for) the Z-80 version of TLC-LISP. Later we used the P-code output to generate several versions of TLC-Logo, as well as aid in the development of LISP itself. More recently, we have added transformation and code optimization phases to translate the P-code into reasonably efficient 8086 native code.

Before going into the lower levels we'll look at a more realistic example of compilation.

```
; A TLC-LISP implementation of rplacb.
;
(de xrplacb (l1 l2)
  (rplaca l1 (car l2))
  (rplacd l1 (cdr l2))
  l1)
```

The compiled output looks like:

```
(defpcode xrplacb (l1 l2)
   2 0 0 0
        (fget 0) ; push l1
        (fget 1) ; push l2
        (car)    ; (car l2)
        (rplaca) ; (rplaca l1 (car l2))
        (fget 0) ; push l1
        (fget 1) ; push l2
        (cdr)    ; (cdr l2)
        (rplacd) ; (rplacd l1 (cdr l2))
        (fget 0) ; l1
        (ret))
```

At first blush, the compiler seems to have been overly stupid:

* The result of the rplaca is left on the stack, and
* the result of the rplacd is left on the stack.

Both of these results are stripped off by the (ret), but still...

A moment's reflection makes us recall that the value of rplaca/d is its first argument, and thus the (fget1)'s that follow the (rplaca/d)'s are redundant. This leads us to the following simpler P-code:

```
(defpcode xrplacb (l1 l2)
   2 0 0 0
        (fget 0) ; push l1
        (fget 1) ; push l2
        (car)    ; (car l2)
        (rplaca) ; (rplaca l1 (car l2)
        (fget 1) ; push l2
        (cdr)    ; (cdr l2)
        (rplacd) ; (rplacd l1 (cdr l2))
        (ret))
```

But further reflection convinces us that these optimizations are in fact available at the source level:

```
(de xrplacb (l1 l2)
    (rplacd (rplaca l1 (car l2)) (cdr l2)))
```

... and so P-code optimization doesn't help here.

This is not to say that the compiler always puts out optimal code, but in the majority of cases, hand-optimization of P-code is not productive. P-code is most useful in cases where the compiler will not/cannot recognize that certain P-codes are applicable.

In the case of xrplacb, further speed increases must come from Code- or Asm-objects.

How to use the TLC-LISP Compiler

The interactive loop between the editor and the interpreter makes for a rapid program development cycle. Once some portion of the program has been deemed debugged, it is often useful to compile it, thereby gaining some speed of execution. The simplest technique is to use the compiler within the editor. In this mode we place the cursor at the beginning of a function definition and instead of typing ^JJ, we type ^JC. This command tuple compiles the indicated definition into memory.

Once the conventions are understood (and perhaps some delarations are made) The edit-compile-run loop makes an effective companion to the edit-interpret loop. Even after compiling a set of functions, you can return to their LISP form using the editor. Simply return to the editor, position the cursor in front of the definition and execute ^JJ. The P-code version will be discarded, and the LISP version will re-appear. Don't forget an occasional ^KS or ^KW to keep the external files current.

The compiler can also be invoked explicitly using the Special Form COMPILE, in several different ways. The simplest cases generate P-code that is compatible with interpreted code:

1. (compile <symbol>) compiles the definition associated with <symbol>, saving the current definition on the p-list of <symbol> where DECOMPILE can retrieve it. The editor will recognize ^JC and compile functions within the editor (currently the compiler must be resident to execute this command).

2. (compile <filename>) compiles the file <filename> into the system without generating a P-code file.

3. (compile <filename1> <filename2>) compiles the definitions on file <filename1>, generating a P-code file named <filename2>.

The compile-function will also accept several additional arguments:

```
(special ...) -- compile, assuming  designated variables non-local.
                 (special) makes all parameter references local.

(code ... )  -- compile functions to 8086 code, rather than P-code.
                Requires the Native Module.

(list <file name>) -- build a source form of the compiled code on
       the designated file.
```

Compiler-related Functions

DECOMPILE -- expr (pcode)

(decompile <symbol>) restores the original definition if <symbol> has been compiled using (compile <symbol>)

PLOAD -- subr

(pload <filename>) loads a P-code, Code, or Asm file named <filename>.

DIS -- expr (pcode)

(dis <code object>) disassembles the code into a readable representation. DIS knows about Pcode and if the Native Module is installed, will disassemble Code, and Asm objects.

REMOVE-MACROS

(remove-macros) is used in conjunction with the loading of p-code files. It causes all macros that appear in a p-code file to be thrown away during a pload. This technique will save memory (without harm) when those macros are only used within that p-code file. (remove-macros) cannot be used if macros are embedded in any of the following forms:

catch, throw, unwind-protect, and all Fexprs.

Conventions and Declarations

The default compilation mode for TLC-LISP gives code that is totally compatible with interpreted code. Specifically, all references to variables are made through "value-cells"--the storage locations known to the interpreter.

For example:

```
(de foo (x y) (bar x y) ...)
```

would compile as:

```
(2 0 0 0
     (fget 0)   ; get the first actual parameter
     (arg x)   ; save it as x's value
     (fget 1)   ; get the second actual parameter
     (arg y)   ; save it as y's value
     (pushval x)     ; push x's value
     (pushval y)     ; push y's value
     (nc2 bar)       ; call the function
```

where the 2 0 0 0 preamble explains that foo expects two required parameters and no optional parameters.

The meat of the code follows the preamble and explains that the actual values passed to foo will be bound to x and y, respectively. However, if bar doesn't reference x or y, (and no one within bar wants these variables either) then this careful maintenance of the variables is for naught. The values from the environment surrounding foo will be restored.

In the case where values are only used locally, we can compile more rapid and compact code:

```
(fget 0)
(fget 1)
(nc2 bar)

 ...
```

But as we said earlier, anyone who accesses x or y within bar will not see the right values.

Once this problem is understood, we can proceed with some optimizations that will be effective in files that are to be compiled. Specifically, we have supplied a declaration for the compiler's benefit:

```
(declare (SPECIAL {<symbol>}) )
```

indicates to the compiler that code to reference these <symbol>s must reference the <symbol>'s value-cell.

Furthermore, any variables not so listed will be assumed local and compiled as stack-relative references. So for example:

```
(declare (special y))
```

preceding the definition of foo will result in the following code:

```
(fget 1)    ; get the second actual parameter
(arg y)    ; save it as y's value
(fget 0)    ; get the first actual parameter
(pushval y)     ; push y's value
(nc2 bar)       ; call the function
```

and in this case y's value will be accurate within the execution of bar.

This use of declare at the beginning of a file will insure that all occurrences of the designated variables will be compiled non-locally.

Since a large majority of TLC-LISP programs tend to use variables locally, it is common to preface a file with:

```
(declare)
```

meaning that all variables are to be compiled locally, and then we can target specific variables in specific functions to be handled non-locally by using a key-word prefix, &special (or &spl) in the formal parameter list to indicate those non-local variables. Thus the previous example could have been handled by

```
(declare)

(de foo (x y &spl y) (bar x y) ...)
```

FOR NOW: any non-local variables that are referenced inside catch, throw, unwind-protect, or any Fexpr must be declared special. See the *.LSP files for examples of special declarations.

Another compiler optimization allows us to specify symbols that are to be treated as constants. Thus:

```
(constant *buffer-size*  pi)
```

This declaration will allow the compiler to replace code like:

```
(pushval *buffer-size*)
```

with `(const  <*buffer-size*>)`

where <*buffer-size*> is the compile-time value for *buffer-size*

An Overview of LAP--the LISP Assembler

The actual output from the TLC-LISP compiler cannot be executed directly. Each compiled function is represented as a list of instructions and labels, as the examples in the prior sections demonstrate. Those instruction lists must be translated into sequences of byte codes, and references to labels must be converted into references to the machine location that corresponds to the label. These conversions are accomplished by LAP, the LISP Assembler. The structure is simple: a static table containing opcdoes, and a dynamic table that contains labels and their associated locations in the P-code object. There is another internal table of object references, that has to be retained when the P-code is loaded. The problem arises when a function makes reference to a symbolic constant. In interpreted code, such references will be marked by the garbage collector. When the function is compiled, such references become fields in P-code operations, and are not so easily tracked by the garbage collector. This problem is discussed in the Native Mode Module.

There isn't much else of complexity in the P-code assembler --a simple stack, a frame of arguments, and thou beside me. By comparison, the 8086 assembler in LISP is 2-1/2 times as large, and thou hast to sit somewhere else.

T L C - L I S P   D O C U M E N T A T I O N

A P P E N D I X   I I

The Native Code Module

The Native Code Module
From the Sublime to the Ridiculous

This package contains two basic components.

1. A full-fledged 8086 assembler so that you may write your own assembly code in to comfort of TLC-LISP and then link your creations into the calling structure of LISP. This portion does not require the P-code Module.

2. The P-code to Code translation expanders coupled with their peep-hole optimizers. To utilize this feature you must possess the P-code Module.

The Files

ASM.LSP
    The 8086 Assembler

DISASM.LSP
    The 8086 disassembler. Compare its size with that of DISPCODE.

PASM.LSP
    The P-code to Ccde transformer. Attaches itself to the back-end of the compiler.

ASMOPT.LSP
    A simple-minded 8086 code optimizer. Finds many idiocies, misses others. No, it's not an expert system.

The assembled versions of these files exist as *.P

To install the Native Code Module in your system see the file

INSTALL.ASM

on your Native Code Module Disk.

From P-code To Code

The P-code compiler does an effective job of boosting the execution speed, while decreasing the nodes consumed in Object Space. All the P-code, except a descriptor resides in byte space. Actually, that's a partial lie. We must make sure that any objects that are referenced by the interpreted code, are also maintained by the P-code. For example. if an interpreted function references a quoted list (A B C) then the list will get marked by the garbage collector since it is just a portion of the list structure. However, a P-code reference will simply be an instruction to place a reference to the object (probably) on the stack. How could the garbage collector find this? It could examine each instruction in each P-code object, searching for a markable object--a highly expensive operation, even for LISP. Rather, we encode a vector in the header of a P-code object, each element of which is an object reference that must be marked. The morbidly curious may examine this vector invoking the EXTRA selector on a P-code object. So this vector consumes some Object Space, but substantially less than the list representation for moderate programs.

Why bring this EXTRA detail up here, rather than in the P-code section? You don't have to be morbid to write P-code, but morbidity is a prerequisite for 8086 coding. More generally, the levels below P-code will require much more attention to the internal details of the specific implementation (8086, 68K, ...). So a great deatil of the following material will deal with the internals of TLC-LISP. We will not give a course on 8086 programming, but will assume familiarity with that architecture. Actually, the TLC-LISP implementation uses only a small subset of the possible instructions, and you can learn about that portion of the 8086, at least, by writing some LISP code, compiling it and diassembling the result. It's not pretty, but someone has to do it. Now let move on to the details of Code Objects.

The portion of this Module that deals with compiled code takes the machine intependent code and transforms it into more specific 8086 instructions interspersed with calls to TLC-LISP run-time routines.

As we mentioned in the P-code Module, the major issue to resolve when moving from P-code to Code is the transformation of LISP objects into 8086 addresses. These details, though important, are handled generally by the run-time support. However, if you decide to write your own 8086 native-code routines, they must conform to the conventions.

So to give a flavor of the transformations, the P-code instruction

```
                (CDR)
```

translates into the following sequence:

```
                (POP CX)        ; since TOS has object
                (OBJADR CX)     ; transform object to address
                (GET CX CDR)    ; select the CDR
                (PUSH CX)       ; The value to TOS.
```

These instructions expand further into 8086 instructions or sequences of such. For example

```
                (OBJADR CX)  becomes

                mov bl cl
                and bx 0003
                shl bx 1
                mov es (bx)
                mov bx cx
                and bl FC
```

which certainly must be eye-opening -- more about this later.

There is a certain amount of local optimization done on push-pop, mov-push, and pop-push sequences. For example, the GET-PUSH above will become a single 8086 instruction: (PUSH (ES BX 2)). However no "life-time" analysis of registers is done; and hand optimization may have something to offer here.

The parameter passing for Code objects is still done through full LISP frames as is the case with P-code, but since these frames are stored on the 8086 stack, we can utilize certain frame conventions within 8086 code. For example, the TLC-LISP interpreter builds frames so that as a Code object begins execution, SI points at the bottom of the current frame, and thus

```
      (FGET 0)    translates to    (PUSH (SS SI 12))
```

with similar instructions for the other FGETs

Named calls (NCALL ...) are translated into a sequence of 8086 instructions. The format is not important here, only the fact that compiled and interpreted code can still be freely mixed, modified, and replaced in a dynamically changing environment. The interpreter insures this by building frames for itself, for P-code, and for Code all of which are compatible with one another. The next code objects --of type Asm-- are not as gentile, but are potentially faster.

From Code To Asm

Asm objects are as close to the bits as anyone should get (in fact one can argue that LISP is as close to the bits as any **rational** person should ever get.) The interpreter does not build a frame before calling such a function. Rather the parameters are passed in the 8086 registers-- CX, DX, and DI--respectively. The function is entered with a far call, and exit is expected using a far return with the value in CX. If you don't know what a "far call" or "far return" is, you should get an 8086 book.

Though arbitrary 8086 code is supported at the ASM level, we advise that most code follow a particular stylized form that we call Block asm. The name "Block" comes from our intention to use this code style to support Block compilation. In this scheme of things we are able to compile away all stack references and replace named-calls with direct 8086 call-instructions.

As an interim stage, the Version 1.51 TLC-LISP86 interpreter has been modified to support the necessary internal register maintenance. The next few paragraphs outline the conventions of Block Asm objects. This way we can move from Code objects to these Block objects by a collection of (mechanical) source-to-source transformations.

Since ASM object have no frame, but have their arguments passed in the registers (unless there are more than three of them.) We have two things to accomplish (1) save the registers, and (2) be able to return to the caller.

The arguments are pushed onto the stack (think "8086 stack", not "P-code stack"--they may not be the same), after we push BP and reset BP to the current value of SP. As a result:

* The FGETs transform to BP-relative addresses.

* The RETs transform into restoration of the stack pointer and BP followed by a return to the caller.

The call and return problem is our next issue.

Within a Block we need to call other Asm code within that Block. The 8086 call-instruction is the operation we need, but we must reconcile it (a short call) with the far return required to return to the interpreter. So:

* The entry is transformed into a local internal call followed by a far return. The local code sets up BP and saves the registers.

* Internal recursive calls (that are named calls) can be replaced by direct 8086 calls to the internal routine. This speeds up function calls immensely, but of course removes all possibilities to break, trace, or backtrack broken code at the LISP level.

This scheme requires that BP be maintained across calls to the P-machine, internal routines, or other code blocks.

Finally, we can also "open code" many of the p-machine operations as direct sequences of 8086 code. This requires a certain amount of knowledge about the internal structure of objects, and it requires a certain degree of care, but the results can be dramatic. For example, the TAK function

```
(DE TAK (X Y Z)
   (IF (GE Y X) Z
       (TAK (TAK (SUB1 X) Y Z)
            (TAK (SUB1 Y) Z X)
            (TAK (SUB1 Z) X Y))))
```

computes (TAK 18 12 6) in the following times on an 8-Mhz 8086:

P-code	24.6 Seconds
Block code	9.3 Seconds
+ Open SUB1	6.2 Seconds
+ Open GE	1.9 Seconds

We include the final TAK Block code below.

```
(defasm TAK ()
        (call inttak)           ; initialization ritual
        (retf)

inttak  (push bp)               ; set up a simple frame
        (mov bp sp)
        (push cx)               ; save the arguments
        (push dx)
        (push di)               ; end of initialization

        (mov cx (bp -4))        ; (fget 1)
        (mov dx (bp -2))        ; (fget 0)
        (cmp cl dl)             ; y-x  -- open (ge)
        (jb iff7)               ; jump    if x > y
        (mov cx (bp -6))        ; (fget 2)

xit     (add sp 6)              ; synch the stack
        (pop bp)
        (rtn)                   ; (ret)

;continued
```

; continued from previous page

```
iff7    (mov cx (bp -2))        ; (fget 0)
        (dec cl)                ; open (subl)
        (mov dx (bp -4))        ; (fget 1)
        (mov di (bp -6))        ; (fget 2)
        (call inttak)           ; (nc3 tak)
        (push cx)               ; the result

        (mov cx (bp -4))          ; (fget 1)
        (dec cl)                  ; open (subl)
        (mov dx (bp -6))          ; (fget 2)
        (mov di (bp -2))          ; (fget 0)
        (call inttak)             ; (nc3 tak)
        (push cx)                 ; the result

        (mov cx (bp -6))            ; (fget 2)
        (dec cl)                    ; open (subl)
        (mov dx (bp -2))            ; (fget 0)
        (mov di (bp -4))            ; (fget 1)
        (call inttak)               ; (nc3 tak)
        (mov di cx)                 ; the result set up ..
        (pop dx)                ; as last argument to  ...
        (pop cx)                ; inttak
        (call inttak)
        (jump xit)     )))
```

Of course, the code could still be improved,

* generally, the open-coded (ge) could be performed before saving all the registers. These changes drop the time to 1.1 seconds, but require deeper analysis by the compiler.

... still the 1.9 seconds is a long way from 24 seconds.

Interfacing to the Assembler

TLC-LISP supports three different degrees of assembly code:

P-Code: This is a LISP pseudo code, emitted by the compiler. As such, there is no need for the LISP programmer to deal directly with the compiler output. However for completeness we mention that the pseudo-machine is a single stack device whose operations take their operands from the stack and place their results on the top of the stack.

Code: This level of code is a mix of 8086 native code, macro calls that expand to 8086 code, and 8086 calls to LISP internal routines to interface between 8086 objects and TLC-LISP objects. As with P-code objects, parameter passing is policed by the required/optional preamble.

Asm: This level of code is as close to the raw machine as one should ever get. Parameters are not checked, and arguments are passed in the hardware registers. When speed is of the essence, ASM-objects are in order.

* Asm objects are passed arguments in CX, DX, and DI (for functions of 0 - 3 arguments).

* The asm object is invoked via a far call thus must do a far return.

* The asm object must preserve DS, SS, SI and must maintain SP. Do not invoke any internal lisp functions without the values in DS, SS and SI as they were on entrance to the asm routine.

Three assmbler drivers are included on the file LISP.SYS: DEFPCODE, DEFCODE, and DEFASM. Since arguments are checked for CODE and PCODE objects, these drivers expect their code to begin with a preamble. ASM object expect their arguments in registers as described above.

The following sections are of interest only to those who wish to work with code-like objects below the level of the Assembler. Above that level, macros handle the problems. The internal descriptions will help those who want to write their own machine code, or who wish to use DIS to examine Code or asm objects.

A Brief Outline of The Design of TLC-LISP86

The following sections can be ignored without peril. They contain a discussion that may interest implementors and possibly those who plan to write at the LISP assembly-level.

The basic memory model of TLC-LISP86 is a BIBOP implementation. We encode pointers in a 16-bit word accessing a 4-byte quantum. Since this means that the bottom two-bits of a address pointer are unused, so we use them to select one of four segments that represent object space.

The BiBop Table

In a BiBop scheme, the object address space is partitioned into pages, each of which can contain objects of a single specific type. The type of each such page is contained in a separate map called the BiBop table. The key to the success of BiBop is the speed of the translation process. In the TLC languages we take the top byte as the index into the table. Thus:

The following table is a list of the embedded types in the current version of TLC-LISP86. The relative position of the type'in the table is the numeric value used by the P-code function named ASRT. For example,

```
ASRT #E
```

verifies that the top of stack contains a string object.

The type	Its value
symbol	0
list	1
expr	2
fexpr	3
macro	4
closure	5
float	6
integer	7
class	8
instance	9
stream	10
subr	11
aload	12
file	13
string	14
env	15
vector	16

package	17
code	18
pcode	19
turtle	20
fix	21
char	22
unbound	23
none	24
illegal	25
asm	26

How To Use All This Information

Notice that objects as simple as integers have an encoded form, and thus are subject to an object-to-address translation before they can be manipulated by the 8086. Because of this and the extensive run-time object management that's necessary to turn a traditional processor into a LISP processor, we cannot produce pure native-code for any non-trivial operations. Compatibility between the two machine architectures is maintained by a set of macro instructions that expand into a sequence of 8086 native code and calls on TLC-LISP primitives. For example:

```
(objadr <reg>)
```

must be invoked to transform a TLC-LISP object into a physical 8086 address. The actual code that defines objadr is:

```
mov bl cl                ; (objadr cx)
and bx 0003
shl bx 1
mov es (bx)
mov bx cx
and bl FC
```

This code selects the bottom two bits of an object reference and maps them through a table of segments. Then uses the extra segment to do the actual selection. So, for example,

```
(get cx cdr)   becomes    (mov cx (es bx 2))
```

In other cases, we need to make explicit reference to TLC-LISP internals. Thus:

```
(assert cx, string)

     mov bx type-table
     mov al ch
     xlat
     cmp al #0E          ;    0E = string type
     jz labl
     callf cs:likecx
labl:
```

Here we need to know:

* the location of the BIBOP table (type-table)
* the numeric value of string-types (0E)
* the location of an error routine to announce type-errors
   (this location has two components: a segment, and an offset)

The correspondence between types and numbers is static and was given in a previous section. The locations of routines will depend of specific versions and machines. That information is accessible through a TLC-LISP function named INTERNALS.

The INTERNALS table is located by invoking:

(INTERNALS) SUBR
      Returns a list of offsets and segment values of interest to assembly language programmers. The values returned are:

BIND -- a far routine that shallow binds the symbol in CX to the value in DX.

TYPE-TABLE -- the offset in DS of the BIBOP type table. Useful for open coding type checks in assembly language as follows:

```
  mov bx,type-table
  mov al,ch
  xlat          ; al now has type of object in cx
```

NAME-STACK -- the offset in DS of the name stack pointer (points into the stack segment). Useful for compiling DO and LET.

CODE-SEGMENT -- the value of the lisp interpreters code segment

OBJTOINT -- far label of a routine that converts a number object in CX into a 32 bit integer in DX:AX.

APFINI -- far label to a routine that removes the current frame and returns to the frame's erector. Useful for code objects only, do not use for asm objects.

NAMED-CALL -- far label for application of functions by name. Arity is in AX, symbol is in CX. Arguments are on the stack and are removed by this routine. Value is returned in CX.

SCALL 0
SCALL 1
SCALL 2
SCALL 3 -- far routines for invocation of built-in functions (subrs). BX has the offset of the subr. CX, DX, or DI has the LAST argument to the subr for aritys of 1, 2, and 3 respectively. The other arguments are on the stack, first argument pushed first. Subr values are returned in CX. The arguments are removed from the stack by this routine.

DISPATCH -- far routine for invoking internal functions witrh arguments in registers. Code offset in BX, arguments in other registers. Lsubrs are invoked this way by setting BP to point above the first argument and AX to the arity. Remember that lsubrs do not remove their arguments.

UNBIND -- far routine that unbinds the name stack to the value in CX. The name stack pointer is not changed by this routine and must usually be set by the caller to the argument which is returned in CX.

LIKECX -- far routine that generates ARG-WRONG-TYPE error with the object in CX.

INTTOOBJ -- far routine that converts a 32 bit integer in DX:AX into an object in CX.

SELFBIND -- far label of a routine that saves the current value of the symbol in CX in the name stack without changing the symbols value. Useful for compiling DO and LET.

MEM -- far label to a routine that does a fast version of MEMQ. Used by the compiler

Routines that are far are invoked via far calls using the offset and the code-segment value. Subr offsets are found via (examine-word (objadr subr)).

To make these ideas more concrete and immediately useful, we give a sequence of examples that utilize a specific setting of (INTERNALS) and a specific loading of routines in memory on one specific afternoon.

Here was the result of (INTERNALS):

```
(#[16]26F9       ; bind
 #[16]540        ; type-table
 #[16]694        ; name-stack
 #[16]B55        ; code-segment
 #[16]2705       ; objtoint
 #[16]1F44       ; apfini
 #[16]270D       ; named-call
 #[16]2786       ; scall 0
 #[16]278B       ; scall 1
 #[16]2791       ; scall 2
 #[16]279E       ; scall 3
 #[16]2783       ; dispatch
 #[16]26FD       ; unbind
 #[16]615F       ; likecx
 #[16]2709       ; inttoobj
 #[16]26F5       ; selfbind
 #[16]26E0       ; mem
 #[16]139F       ; alloc
 #[16]155B       ; zballoc
 #[16]FFFF
 #[16]FFFF
 #[16]FFFF
 #[16]FFFF)
```

```
;  A Code example:  Xrplacb, to replace the car- and cdr-portions
;                    of a pair with new objects.
;     A  binary operation with arguments on stack,  and result to
;       replace top of stack.  Return must clean up machine state
;       through the apfini entry in the internal table.
;


(defcode XRPLACB (11 12)
  2 0 0 0
  (fget 0)       ; L1
  (fget 1)       ; L2
  (pop cx)          ; L2
  (pop dx)          ; L1
  (objadr cx)       ; L2
  (get cx cdr)
  (get di car)
  (objadr dx)       ; L1
  (put cdr cx)     ; rplacd
  (put car di)     ; rplaca
  (push dx)         ; return L1
  (ret)
)))
```

```
; The result of (dis xrplacb) was:
;

C24F:E2A1  02 00 00 00
C24F:E2A5  push (ss si 0012)    ; (fget 0)
C24F:E2AA  mov cx (ss si 0010)  ; (fget 1) (pop cx)
C24F:E2AF  pop dx               ; (pop dx)
C24F:E2B0  mov bl cl            ; (objadr cx)
C24F:E2B2  and bx 0003          ;  ... expansion
C24F:E2B6  shl bx 1             ;  ... continues
C24F:E2B8  mov es (bx )
C24F:E2BA  mov bx cx
C24F:E2BC  and bl FC
C24F:E2BF  mov cx (es bx 0002)  ; (get cx cdr)
C24F:E2C4  mov di (es bx )      ; (get di car)
C24F:E2C8  mov bl dl            ; (objadr dx)
C24F:E2CA  and bx 0003
C24F:E2CE  shl bx 1
C24F:E2D0  mov es (bx )
C24F:E2D2  mov bx dx
C24F:E2D4  and bl FC
C24F:E2D7  mov (es bx 0002) cx ; (put cx cdr)
C24F:E2DC  mov (es bx ) di      ; (put di car)
C24F:E2E0  mov cx dx            ; (push dx) (ret)
C24F:E2E2  jmpf 0B55:1F44       ;  = jmpf cs:apfini


; An ASM example -- A fast xrplacb (without type checking)
;                   arguments in CX and DX; result to CX.
;

(defasm XRPLACB (l1 l2)
  (objadr cx)      ; L2
  (get cx cdr)
  (get di car)
  (objadr dx)      ; L1
  (put cdr cx)     ; rplacd
  (put car di)     ; rplaca
  (retf)
```

T L C - L I S P   D O C U M E N T A T I O N

A P P E N D I X   I I I

The Editor Customization Ritual

How To Install The Editor In MSDOS and CPM/86 Systems

***** Note *** This section does not apply to PC-Dos compatible versions of TLC-LISP.**

The TLC-LISP editor is composed of three pieces:

EDIT.P -- the P-code file that defines the higher-level functionality of the editor. This editor is an extension of the basic editing facilities of WordStar, including search, replace, and block operations, but not including the text-justification and formatting commands. The basic repertoire of the editor has been extended, however, to include some LISP-specific operations -- parenthesis balancing, incremental evaluation, and incremental compilation. See pages 126-127 of the TLC-LISP Reference Manual for a command summary. The source for the EDITOR is available from TLC as part of the System Module.

TERM.P -- the P-code file that defines the basic screen primitives. This file is automatically loaded by EDIT.P. This file contains all the terminal dependent references made by the editor. The file includes basic drivers for a myriad of terminal types -- ANSI versus binary, smart versus stupid, insipid, vapid, flatuent, and even urbane terminals; all are handled. The source for this mystical file is included in the Kernel Module as TERM.LSP.

*.CUS and CUSTOM.LSP -- these files select specific drivers for the user's specific terminal type from TERM.LSP. The files named *.CUS are examples of completed customizations. The specific CUS file that is used by TERM.P is copied into CUSTOM.LSP.

The next section explains how to build a *.CUS file or, if an appropriate CUS file exists, how to get CUSTOM.LSP installed.

Using The Terminal Customizer

Before the Editor can be used, you must install a driver for your specific terminal. We have included a few such drivers on the system disk. To examine your options from the comfort of LISP, type:

```
LISP      (followed by a carriage return).
```

and when you get the prompt

```
>>>
```

type

```
(stat "*.cus")           ; followed by carriage-return.
```

This operation will give you an indication of what's currently available, since the file names are descriptive of the terminal type. To install one of the CUS fles, or to make your own, enter LISP and then load the customizer by typing:

```
(LOAD "CUSTOMIZE")             ; again, supply a return.
```

Be sure that both leading and trailing double-quotes (") are present. If the file does not seem to be loading, then (1) make sure you typed the carriage return, (2) or if you left off the trailing parenthesis, type it followed by return. Or (3) if you missed the trailing double-quote("), hold down the control key, strike the g-key, (this is written ^g or ctl-g) and follow this incantation with a return. You'll receive an error message ("user abort"), and then type (LOAD "CUSTOMIZE") again.

Once the customizer is loaded, you will be asked a series of questions. Below is a sample interaction to customize the editor for a Fujitsu FM-16. Text that follows a semi-colon (;) is commentary. User responses are underlined. Other text is printed by the system. The first time you'll see from the customizer is:

```
Below is a list of pre-defined terminal types
1   ADM16.CUS
2   ANSI.CUS
3   FOOBAR.CUS
4   TUTI.CUS
5   TVI950.CUS
Type the number associated with your terminal,
or type 0  to define a new type 0       ; we want to define a new type.

Please supply a terminal name for this new type
fujitsu

; If one of the existing types was  appropriate, then we would
; have typed the necessary numeric response and the corresponding
; *.CUS file would have have been copied into CUSTOM,LSP and we'd
; be done.  But zero opens up a series of questions:

Number of columns on your screen? 80

Number of rows on your screen? 24

; In the next line we'll see an "escape-sequence" -- a sequence
; that begins with the escape character and is followed by a
; sequence of bytes.  Escape sequences are a common way to
; indicate terminal commands. When we type the escape-key in the
; customizer it will echo as the five-character string *ESC*.
; Thus:

Erase to end-of-line? (return if not) *ESC*T
```

```
; This will make the string escape-followed-by-uppercase-t.


; Next we ask about highlighting. Highlighting is used for block
; mode commands. Some terminals can't highlight, some reverse the
; background, and some change the characters' color.

Start of Highlighting? (return if not) *ESC*Go

; Since we have specified that highlighting can be turned on,
; we must also specify how to turn it off.

End of Highlighting? (must be supplied) *ESC*GG

; The next entry illustrates a new feature of the customizer:
; the ability to over-ride the implied character-oriented
; escape sequence. It is frequently convenient to  mix character
; and numeric quantities in the same terminal command.  For
; example, we assume that *ESC*0 means the escape character (27)
; followed by  the  character 0 (48). To get escape followed by
; the  number 0, we preface 0 with a percent sign, as in *ESC*%0.
; Numbers that  follow the percent are assumed to be decimal.

; In this first case we need a simple numerical value. Thus:

Clear the Screen? (must be supplied) %26

; The sequences for cursor-on/off are more complex:

Turn the cursor off? (return if not) *ESC*.%32%0

; So the period is a character and the 32 and 0 are numeric.
; We could have represented the 32 by a character space, but
; the null character (0) is harder to come by.

; Since the cursor can be turned off, we expect to be given a
; sequence that will turn it on:

Turn the cursor on? (must be supplied) *ESC.%0%7

; The Fujitsu terminal emulator does not have character-insert
; or character-delete commands, so we respond with carriage-
; return (written RET), and the editor will simulate these
; operations  using software in TERM.P. The morbidly curious
; may see the code in TERM.LSP.

Insert character mode? if none, return RET

Delete character mode? if none, return RET
```

```
; Insertion and deletion of lines is supported:

Insert line  mode? if none, return *ESC*E

Delete line  mode? if none, return *ESC*R

; The major command the Customizer must install is the one to
; position the  cursor at an arbitrary location on the screen.
; There are two basic rituals for such positioning:

;  Binary -- numerical values are given for cursor positions
;            usually x-y coordinates offset by some constant.
;            For example a frequent offset is 32, so 20 would
;            be represented as 52.

;  ASCII -- the cursor coordinates are given in ASCII. Thus
;           20 would turn into the string "20". ANSI terminals
;           use this style.

; The customizer supports both forms.

; Regardless of the form, terminals expect cursor positioning
; to be prefaced with a command. Thus:

What initiates a cursor positioning command? *ESC*=

; Immediately following the preface is the row/column information.
; We need to know which comes first. Row? Column?

Which comes first, row or column? R for row, C for column r

; Now comes the major decision-- ascii or binary row/column
; data.

Is the positioning in ascii or binary? A for ascii, B for binary b

; Since we have assumed a binary representation for row/column
; data, we are expected to give an offset to be added to each
; position. Thirty-two is a common value, and that's what this
; terminal expects.

Offset to add to line? 32

Offset to add to column? 32

Customization complete
```

That's all there is to it. A file named FUJITSU.CUS is formed and a copy is made and named CUSTOM.LSP. The latter file is used by the editor. The former becomes part of the terminal library. An examination of CUSTOM.LSP (with TERM.LSP in hand) will show the curious how the customization process works.

The dialog for an ASCII terminal is only slightly more complex. These terminals expect a separator between the row and column information and a terminator to end the cursor command. The Ascii-branch of the customizer prompts for this information and acts accordingly.

The current customizer seems to take care of most terminal anomalies. Special cases may occur and can be addressed by overlaying portions of TERM.P with new code. For example, more complex screen initialization may be required to enter the editor, and correspondingly restoration may be needed on leaving the editor. Those drivers are in TERM.LSP and can be modified and used in interpreted form without degrading the performance of the editor. If these modifications are required, be sure to install them in the EDIT: package.

A final remark: There is a two-fold purpose for supplying the customizer in source form. First, it illustrates some programming techniques unique to LISP. Specifically, the elements of the dialog are carried in several tables that consist of a message to be printed and a function to be applied to the response. This makes it easy to build a flexible response with minimal code. No, this is not the rudiments of an expert system!

Second, it might be convenient to modify the customizer for your particular terminal. If so, we'd be interested in hearing about the enhancements you've made.

On Keyboard Customization

Another possibility for variation is the interpretation of keystrokes. Though this portion of the editor has not been decoupled from the terminal specifics as completely as the terminal emulator, we can still redefine and/or enhance much of the activity. (More detailed modification can be accomplished using the editor source in the SYSTEM Module). This current section outlines the basic keystroke-to-action mechanism of the editor and duplicates the code of the editor so that modifications may be done. Such modification is a semi-advanced exercise, not recommended for the beginner.

The interpretation of the keystrokes is accomplished though a 128 element vector named cmd that resides in the edit package. Behavior of the editor can be modified by changing that table.

The following is the current configuration of cmd:

```
; CMD -- the function to store an action into the appropriate
;          slot in the keyboard vector.

(de CMD (chr fcn)
  (store cmd-vector (addl (ascii chr)) fcn)   nil )))

; CMD-VECTOR -- vector of commands corresponding to characters typed

; Build the initial vector and initialize each entry with an
;  error function. (Many of these entries will be over-written)

(setq cmd-vector (newvec 128 illegal-cmd))


; Now set the ordinary keys to be inserted into the text stream.
;
(for (i (ascii *space*) (ascii \~))
  (cmd (ascii i) insert-cmd) )    ; printable characters get inserted

; carriage return is nothing special either:
;
(cmd *cr* insert-cmd)

; Treat tab (ctl-i) specially, please:
;
(cmd ^i tab-cmd)
```

```
; Now define the simple Wordstar-like commands:
;
(cmd ^x down-cmd)
(cmd ^e up-cmd)
(cmd ^d right-cmd)
(cmd ^s left-cmd)
(cmd ^r prev-page-cmd)
(cmd ^c next-page-cmd)
(cmd ^z scroll-up-cmd)
(cmd ^w scroll-down-cmd)
(cmd ^f word-right-cmd)
(cmd ^l find-replace-again-cmd)
(cmd ^a word-left-cmd)
(cmd ^y delete-line-cmd)
(cmd ^t delete-word-cmd)
(cmd *backspace* backspace-cmd)
(cmd ^g delete-cmd)
(cmd *rub* backspace-cmd)

; The next three commands spoil the clean mapping by expecting
; a second key to chose between options. Thos options are not
; spelled out (yet).
;
(cmd ^k k-cmd)
(cmd ^q q-cmd)
(cmd ^j j-cmd)

(cmd *esc* esc-cmd)  ; To leave the editor with the escape key.
```

One extension of this mechanism that we've seen is the desire to extend the editor to handle more that 128 possible keys. It's easy:

0. Enter the edit: package: (setq package edit:)

1. Define a new vector of the appropriate size and copy cmd into it:

```
   (setq newcmd (newvector <size> 0))
   (for (i 1 128) (store newcmd i (cmd i)))
```

2. Now augment the new slots with the the desired functions. For example if you what the new positions to be normal keys then:

```
    (for (i 128 <size>)  (store newcmd i insert-cmd))
```

3. Now install the new command vector: (setq cmd newcmd)

4. Remember to exit from the edit: package: (setq package sys:)

If you install this code on LISP.SYS as part of the initial editor load, then the editor will know about characters above 128.

If you wish to (and are able to) type in characters whose ASCII codes are above 128, then another modification needs to be made. Specifically, each character is assigned a "character type" using the current (vector) value of READ-TABLE. Since TLC's vectors are one-based, the type for the character whose ASCII code is I will be found in (READ-TABLE (ADD1 i)). READ-TABLE's value is a vector of 256 elements whose top 128 elements are initially the same as the corresponding elements below 128. So if you wish to read characters in this portion of the table, set their type values accordingly. See pp.108-109 of the Reference Manual.

TLC
The LISP Company
End User Program License Agreement
June, 1984

CAREFULLY READ THE FOLLOWING LEGAL AGREEMENT REGARDING YOUR USE OF THE ENCLOSED TLC PRODUCT. IF YOU DON'T AGREE WITH WHAT IT SAYS, PROMPTLY RETURN THE UNUSED SOFTWARE AND DOCUMENTATION AND YOUR MONEY WILL BE REFUNDED.

You are required to return the End User Agreement Acknowledgement Form to receive customer support and product updates.

The LISP Company (TLC) develops computer programs and related materials (its Products).

End User (the consumer) desires to obtain the benefits of TLC's Products and by opening this package agrees to abide by the terms of this License. Therefore, subject to the following terms and conditions, TLC grants to End User a non-transferable license to use its Products only as indicated below.

Article 1: General Copying Restrictions. End User shall only make copies of TLC Products when authorized to do so by TLC. Unauthorized copying of TLC Products (including Products that have been modified, merged, or included with other software) and the acquisition and use of unauthorized copies of TLC Products may be both criminal and civil offenses for which End User may be liable for fines, damages, and attorney's fees. TLC has the right to terminate this license and to take legal action if the terms of this license are violated. TLC has the right to trace serial numbers at any time and in any reasonable manner.

Article 2: Archival copies. End User may make archival backup copies of TLC diskettes, but only if such copies are for End User's personal use within the scope of this license. Any copying of documentation is strictly prohibited.

Article 3: Proprietary rights of TLC. The TLC logo, product names, software, manuals, documentation, and other support materials are either patented, copyrighted, trademarked, or owned by TLC. End User agrees not to remove any product identification or notices of such proprietary restrictions from TLC Products. TLC retains exclusive ownership of the TLC software and of TLC printed materials.

Article 4: Use with multiple computers or terminals. This license is limited to use of the TLC Products included in this package on a single computer and may not be transferred or assigned. In the event End User intends to use a TLC Product on more than one computer, or if End User's computer is or becomes capable of allowing multiple terminals to access common disk memory, End User shall notify TLC of the proposed configuration and apply for a multiple use license. All multiple use license fees shall be in accordance with TLC's fee schedule then in effect and shall be paid directly to TLC.

Article 5: Customer service. End Users may obtain customer service from TLC (at the address below) only if a properly signed End User Agreement Acknowledgement Form is on file at TLC's main office.

Article 6: Update Policy. TLC may from time to time revise or update its Products. Revisions will be provided to End Users only if a properly signed End User Agreement Acknowledgement Form is on file at TLC's main office. TLC is not obligated to make any Product revisions, or to supply any such revisions to End User.

Article 7: Termination of End User license. If any of the terms and conditions of this Agreement are broken by End User, in addition to all other legal rights and remedies, TLC may terminate this license. Upon termination, End User shall return to TLC all TLC Products and copies thereof, whether modified, merged, or included with other software, and shall certify in writing to TLC that End User has not retained any TLC Products or copies thereof. The provisions of this license which protect the proprietary rights of TLC shall continue in force after termination.

Article 8: Governing law. When entered into in the United States, this Agreement shall be interpreted in accordance with the laws of the State of California. Otherwise, this Agreement will be interpreted in accordance with the laws of the United States or such other law as may be required to protect the legitimate interests of TLC.

Article 9: End User Agreement Acknowledgement. End User may obtain updates, customer service, and TLC newsletters only if End User signs and mails the TLC End User Agreement Acknowledgement Form.

DISCLAIMER OF SOFTWARE WARRANTIES AND LIABILITIES

1. TLC SOFTWARE IS DISTRIBUTED AND LICENSED "AS IS." All warranties, either express or implied, are disclaimed as to the software and its quality, performance, or fitness for any particular purpose. You, the consumer bear the entire risk relating to the quality and performance of the software. In no event will TLC be liable for direct, indirect, incidental, or consequential damages resulting from any defect in the software. If the software proves to have defects, you, and not TLC, assume the cost of any necessary servicing or repairs.
2. 30-DAY LIMITED WARRANTY ON DISKETTES. TLC warrrants the enclosed diskette(s) to be free of defects in materials and workmanship under normal use for 30 days after purchase. During the 30-day period, you may return a defective diskette to TLC, at the address given below, and it will be replaced without charge unless the diskette is damaged by accident or misuse. Replacement of a diskette is your sole remedy in the event of a defect. This warranty gives you specific legal rights, and you may also have other rights which vary from state to state.
3. Some states do not allow the exclusion or limitation of implied warranties or liability for incidental or consequential damages, so the above limitation or exclusion may not apply to you.

The LISP Company, P. O. Box 487, Redwood Estates, CA 95044
(408) 354-3668

END USER AGREEMENT ACKNOWLEDGEMENT FORM

Please complete and return this form. Keep the End User Agreement in your files.

The undersigned End User of TLC product materials hereby acknowledges that he or she has read and fully understands the terms of the End User Agreement, the terms and conditions of which are hereby incorporated in this form and acknowledged by this reference.

The undersigned hereby agrees that by signing this document he or she becomes a party to said End User Agreement and agrees to be bound by all terms, conditions, and obligations contained therein.

End User's Signature ____

Please print legibly:

Product Name TLC-LISP Version 1.51 Serial # ______

Date of Purchase ___________________

End User's Name ____

Address ____

City _________________ State _______ Zip Code _____

Country _____________________________

End User's Company's Name (if applicable) ______________________

Address ____

City _________________ State _______ Zip Code _____

Country _____________________________

End User's Computer Make and Model ______________________________

Serial # ____________

(Tape or staple here)

```
+--------+
:        :
:        :
:        :
:        :
+--------+
```

The LISP Company
POB 487
Redwood Estates CA 95044

(fold here)

The TLC User's Group has been formed to encourage communication and cooperation between users of TLC-LISP and TLC-Logo. We produce a quarterly newsletter containing technical information as well as topical (if not irreverent) comments on the fields of AI, expert systems, language design, education, and crop rotation.

___ Yes, I wish to participate in the TLC User's Group. Please bill me for $25.00 for the first four issues.