Table of Contents

Introduction

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 wants to know quickly 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 idiosyncrasies, it is imperative that ALL prospective users read Part II Section A, An Introduction to TLC-LISP.

Though this manual does have a collection of examples and catalog of the LISP library, this manual is not organized as a cook book that can produce LISP programmers like chocolate chip cookies. It is unfortunate that no suitable LISP primer exists yet; we do plan to provide a self-contained instructional primer for TLC-LISP in the near future, however. In the meantime we will emphasizes 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.

The bibliography references several sources of LISP information. As yet, there is no totally satisfactory text which intoduces the novice to LISP. Two of the references, Artificial Intelligence Programming, and The Little LISPer, deal with programming aspects of LISP; Anatomy of LISP involves abstract programming concepts and LISP implementation techniques; the August issue of BYTE magazine discusses several interesting LISP applications; the other bibliographic references deal with more philosophical or technical issues related to LISP in particular and programming in general.

Though an active LISP user might find the manual sufficient to explore the system, a LISP novice could spend an inordinate amount of time discovering how LISP can be used effectively. Due to time constraints, the TLC-LISP manual does not contain comprehensive, well-documented LISP applications. Fortunately the recent book "AI Programming" by Charniak, Riesbeck, and McDermott contains several substantial applications and will help both the novice and the expert in developing their understanding of LISP.

Furthermore, one of the most important lessons to learn in LISP programming is that of "style". The power and flexibility of LISP can lead to programming excesses; it is easy to write incomprehensible LISP code. Again it is fortunate that the issue of style has high priority in "AI Programming".

Though the LISP dialect represented by TLC-LISP is not the same as that discussed in "AI Programming", it is a simple matter to convert from the AIP dialect to TLC-LISP. The appendix of this manual discusses a few of the inconsistencies, thereby mininizing some of the transitional difficulties. The appendix also contains annotated TLC-LISP code that implements many of the techniques discussed in AI Programming as well as general examples of TLC-LISP.

First, an historical note. Both TLC-LISP and the dialect of the AIP book have the same ancestor: the original MacLISP for the DEC PDP-6 developed at MIT. When Stanford received a PDP-6 that LISP was converted to run under the DEC monitor; 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 becoming 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". 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 were pressing. The new implementation, known as BIBOP, 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 The LISP Company LISP that you have purchased from us. Another major factor in this LISP is the MIT LISP machine experience. That machine and its LISP dialect is again a consolidation: this time including architectural considerations in the equation.

So, though the ancestor of these two LISPs is the same, the paths since that date have been quite different. This version of TLC-LISP represents another point of consolidation, bringing a truly powerful LISP into the micro-computer domain.

This Z-80 TLC-LISP is a preview of things to come. We have consolidated some of the twenty-years experience with LISP1.5, and later with MDL, CONNIVER, MACLISP, and the MIT LISP machine, to present a capable, expandable and clean 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. In that light, we have deferred some of LISP's more exotic features to future implementations.

So, dear reader, understand the past, enjoy the present, and anticipate the future!

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

Copies of this manual may be obtained from: (T.(L.C))
Box 487
Redwood Estates, CA 95044

A General Introduction to 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 the director of Stanford University's Artificial Intelligence Laboratory. At that time McCarthy had just become co-founder (with Marvin Minsky) of the MIT Artificial Intelligence Project. One of McCarthy's concerns was a 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 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, algebraic simplification might rewrite 2*(x+6*y)+x as 3*(x+4*y). (For a detailed discussion of Algebraic Manipulations systems see "LISP-based Symbolic Math Systems" by D. R. Stoutemyer in the August 1979 issue of BYTE). The design of such algorithms involves the solution of two problems: a representation for algebraic expressions, and specification of the algorithms which manipulate that representation.

1. The Representation Problem: How to encode algebraic expressions in a manner that maintains the properties which are important to such symbolic manipulation algorithms? We could assign numbers to each component of the expression and then encode the expression as a vector of those numbers (recall that this is 1958 and Fortran is the only high level language). Assuming that appropriate conventions distinguish between the numbers that are coefficients and the numbers that are representing components like *, +, (, and), we would 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?" In this problem we need a representation that makes the interrelationships more apparent. Here LISP introduced Symbolic Expressions. Symbolic Expressions are a very general, abstract notation which have fascinating theoretical

properties comparable to those of the natural numbers, and yet 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.

We will discuss Symbolic Expressions and their representations in more detail later; for now, we will confine our attention to their application. For our problem we choose to represent algebraic expressions as a special kind of Symbolic Expression called a "list". A list contains zero or more elements. The empty list is represented by a pair of balanced parentheses --thus ( ); a non-empty list may contain other lists as elements, as well as containing atomic (non-list) elements. These atomic elements are called atoms. For the purposes of this example, an atom is either a number, as in most other programming languages, or may be a non-numeric object called a <u>literal atom</u>. Some LISPs, including TLC-LISP, call literal atoms <u>symbols</u>. Literal atoms are commonly called "<u>identifiers</u>" in most other languages --that is, strings of letters and digits (and perhaps special characters) such that the first character in the identifier is a letter. Reflect for a moment that in other languages, identifiers are present in the syntax of the <u>language</u> but are not present as <u>data objects</u>. The following are literal atoms of LISP:

ROCKET TLC TIMES A

So far we have said that ( ) represents the empty list and that lists may have atoms and lists as elements, but have not described how one represents objects as elements of a list. Given elements e1, e2, e3, we can create several lists; one of which is (e1, e2, e3), another is (e2, e1, e3). So one creates lists by separating the elements with commas, and surrounding the conglomeration with the appropriate parentheses. As the examples illustrate, the order of the elements is important; these are not sets, but sequences of elements. The notation can be simplified by omitting the commas, writing (e1 e2 e3) for example.

As indicated earlier, these LISP data structures are interesting abstract objects; however, our main concern now is their effective exploitation in the solution of complex problems. In particular, how can we use these data objects to represent the algebraic expressions? For example we could represent the expression 6y as a list (TIMES 6 Y) where we write TIMES and Y as the representation of * and y, respectively. Notice that the first element of the list represents the operation and the remainder of the list represents the operands. Continuing, the expression x+6y would be represented as (PLUS X (TIMES 6 Y)). Note that the notation still makes clear which components are operations and which are operands. Finally, 2*(x+6y)+x is written as

```
(PLUS (TIMES 2 (PLUS X (TIMES 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 expressions. Given this representation, we proceed to our algorithm.

2. Design The Algorithm: how to write the algorithm which encodes the process which we wish to capture. The concept of algorithm transcends any notion of a specific programming language; that is, we should conceive our algorithm in an atmosphere which is as free as possible from syntactic 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.

A further identification of tasks in the solution formation is useful. An algorithm can be viewed as consisting 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.

Since much of our knowledge is captured in appropriately abstract data structures, the major business of a programming language is to supply a complete set of tools for data structure maintenance, along with a complementary set of control constructs. The control constructs tend to complement the data structures since the flow of control is often based on the structure of the data. The LISP control constructs which we need for our algebraic simplification problem are: the conditional expression and recursion.

The LISP conditional expression is similar to the "if-then-else" construct of other languages. The application of a conditional expression is appropriate when we encounter a data object which can be one of several forms. For example, a term in a polynomial may be a variable, a constant, or a product of variables and constants. Our algorithm will contain a conditional expression which tests for the occurrence of these variants, and performs actions accordingly.

The form of such a conditional expression is:

```
(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.

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 which 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 ((ZEROP (FIRST-ARG TERM)) (SECOND-ARG TERM))
      ((ZEROP (SECOND-ARG TERM)) (FIRST-ARG TERM))
      (T TERM))
```

where FIRST-ARG AND SECOND-ARG are LISP functions defined to select the first and second arguments from the representation of the sum. Of course, before we can run such a program fragment 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 declarative 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-ARG, called <u>recognizers</u> and <u>selectors</u> 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 <u>constructor</u> is used to make a new element of the desired type. Data type specifications in LISP are handled through these constructors, selectors, and recognizers. Thus in LISP, data items have an associated type, while variables are <u>type-free</u>, 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.

The macro facility in LISP helps to support these programming techniques while maintaining efficiency. The data type manipulating functions may be defined as macros which can either be destructively replaced at run time by the representation dependent code, or if a compiler is available, can be expanded into code equivalent to that produced if the representation was used directly.

Regardless, we should strive to write the algorithm in an "abstract" way which expresses the "process" rather than encodes the representation, and relegate the details of representation to well-defined interface specifications. As we have just seen, LISP contains excellent mechanisms for supporting this style of programming. It is amusing to reflect on how much and how little we have learned about programming in the last two decades.

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 (PLUS X (TIMES 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 which 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 or programs).

When a text editor manipulates a piece of source program it is acting on program elements as text items. Of course, most text editors view programs as simple strings of characters without structure or content; this view is an archaic remnant of the keypunch days; and of course, the program could be transformed from its internal representation into a form which the editor could manipulate and then retranslate. However, unless "programs" are a data type of the language in which the editor is expressed, the transformation program cannot be expressed in that language. That "missing data type" unnecessarily increases the machine-dependent component of the implementation. For pure economy of expression it is beneficial to include a full complement of program manipulation operations. Given this facility, it becomes easy to write a program editor in the language itself.

A debugger, again by definition, must be able to manipulate programs. Once an error is discovered, the debugger must be able to modify the program and possibly continue from some modified state. Again, with programs represented as data, expressing debuggers in the language is straightforward.

The term "program transformation" system spans a spectrum from 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 which 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; all phases of an intelligent programming environment come into appropriate proportion.

In summary, LISP is best thought of as a "high level machine for programmers"; it contains a library of operations, including the 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 Structures

This section gives 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 non-numeric objects --called literal atoms--are a versatile naming structure for LISP data. They are used as constants of the programming language (T and NIL), as primitive data objects (TIMES, PLUS, and the variable names in the previously discussed algebraic examples), as representations for all the programming language constructs, and, as we will see momentarily, literal atoms can also be used to capture or attract large collections of data using a literal atom as a name in a dictionary.

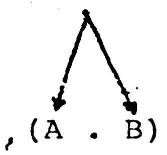
Many LISP implementations (including TLC-LISP) include character and string data types. This allows the manipulation of atom-like character sequences and, with conversion programs, allows the dynamic generation of new literal atoms just as numeric operators can introduce new numbers into the programming environment. This dynamic creation of data objects is a hallmark of LISP that is particularly apparent in non-atomic objects.

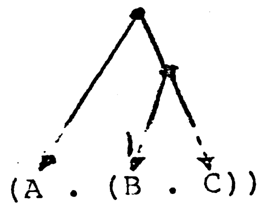
One characteristic of LISP is its ability to take two existing objects and build a new structure from them. Since this construction operation can be repeatedly applied, we can define quite complex structured objects. Traditionally, the construction operation is called CONS.

It is sometimes helpful to visualize the CONS operation as constructing a binary tree (recall that CONS is a binary operator), such that the first operand of CONS is the left branch of the tree and the right branch of the tree is the second operand of CONS. Given two branches, we graft them together (note that the structure need not necessarily be a tree; since operations like (CONS X X) are allowed we may introduce shared structures.)

The most general form of these binary trees are called Symbolic Expressions, S-expressions or S-exprs for short. Typically one manipulates these S-expressions in a notation called dot notation --a notation which represents a tree with left and right branches br-l and br-r respectively, as (br-l . br-r).

Here are a few examples of trees and their dot representation:





For all intents and purposes a special form of S-expr called list notation suffices. We saw list notation in the algebraic simplification example. Recall a list was either empty --denoted by ( )--or was of the form (el, ... en) where each ei was either an atom or a list itself.

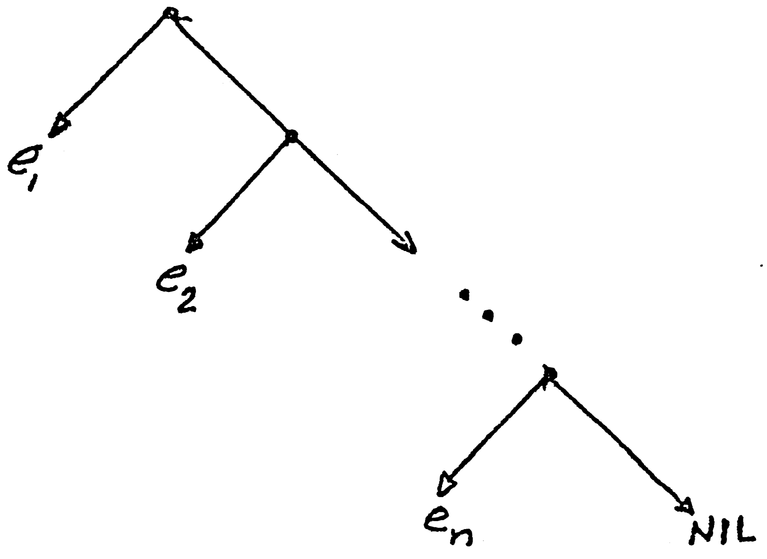
We may represent list notation as an S-expression by the following rules:

1. Map () onto the atom NIL

2. Map (el, ..., en) onto

   (el . (e2 . (... (en . NIL) ...)));

   or in terms of a tree representation we have:



Besides being able to construct new objects, we must also be able to examine the components of such constructed objects. Operations which allow such examinaion are called selectors; they select components. At the S-expression level we have two selectors: one to select the left branch of a tree, called CAR, and one to select the right branch, called CDR. Since non-atomic S-exprs only have two branches, CAR and CDR suffice. 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.

At the list-notation level we have another collection of selectors and constructors. 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. The basic constructor is called CONCAT. In almost every implementation of LISP, FIRST, REST, and CONCAT are identical in implementation to CAR, CDR, and CONS; however it is good style to program at the S-expr level using operations based on CAR, CDR, and CONS, and program at the list level using FIRST, REST, and CONCAT. Do not mix them. This dichotomy is our first example of abstract programming. That is, we should strive to program using 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. 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.

LISP also includes data types that carry implementation information. For example, input and output functions must interface to the underlying file system. Therefore we have a data type which encapsulates file control information. Also, every LISP implementation must have a collection of primitive functions to manipulate data and control the flow of the algorithm (CAR, CONS, COND, etc.), and usually a library of useful definitions (APPEND, COPY, etc.). These definitions are typed objects of the class of executable "micro code" called SUBRS or FSUBRS. Both "file" and "code" data types are included in TLC-LISP.

Evaluation

With the previous sections as background, we can present an abstract description of the LISP evaluation process.

The family of LISP expressions consists of the following:

Class of expression	Examples
constant	1 T '(1 2 3) CAR "xyz"

These are constant of the class: number, truth-value, list, function, and string respectively (in an implementation, "constants" like T and 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.)

```
        variable              X            FACT
```

These are variables which might be found naming simple variables and functions respectively (recall that variables are type-free!)

```
        combination           (CONS 'A (FIRST L))
```

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

```
        conditional           (IF X
        expression                (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 string literals, 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.

   For example: it is clear that the expression (CONS 1 1) should evaluate to (1 . 1); however, what does (CONS A A) represent? We must be able to distinguish between the atom A acting as a variable and the atom A acting as a constant. LISP's solution is to prefix constant S-expressions with a single-quote. Thus (CONS 'A 'A) gives the value (A . A), and (CONS A A) means make a dotted-pair both of whose branches are the value currently attached to the atom A. Note that this difficulty -- differentiating language constructs from language data structures--is only a problem in a language like LISP that allows language constructs to be data structures!

   To tell the complete truth, the single-quoting convention is only an external abbreviation. Internally, LISP will translate '<sexpr> into (QUOTE <sexpr>), making (CONS 'A 'A) into (CONS (QUOTE A) (QUOTE A)) which is a true LISP data structure.

   Note that besides simple constants like S-expressions, numbers, 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 refedinition of system-level functions is a special instance of "dynamic scoping" --LISP's strategy for evaluation of variables.

2. The evaluation of a variable: Recall that LISP variables are "type-free" meaning that a variable is free to take on any type of value --number, string, list, or even a 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. All this means that the evaluation process for variables is reasonably straightforward: using the variable name, extract its value from within the implementation.

   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, meaning 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 their values are requested; that is, 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".

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 is 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 procedure.

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))

(ADD1 X)
```

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.

4. 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. For example, in TLC-LISP 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. See the manual for examples of special form definitions.

   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 to be added to the evaluation routine.

The above four 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 the semantics, or meaning, of 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 and testing instructions --CAR, CDR, CONS, ATOM, and EQ--as well as the control instructions --QUOTE and COND. All others 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 which 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. 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 to 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 <u>side-effect</u>, 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 a value through IX effectively changes a value in IY. This is a problem of sharing values called <u>aliasing</u>. Sharing of values is not problematic provided one cannot modify values. The alternative to modifying values is to copy them; this is what pure LISP does. 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; situations similar to the FORTRAN example can occur, except in the LISP world, it is nowhere near as apparent when structure is being shared and as a result, modification operations must be used with great care. These operations are described in their own section of the TLC-LISP manual.

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 intepretively 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 to pass 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. Recall that a list can be constructed by a sequence of CONSes. We would like an operation called LIST that would take an arbitrary number of arguments and perform like the nested CONSes.

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

In the next chapter we will show how to define LIST as a macro. 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 2 (LIST 3)). Finally (LIST 3) will decompose into (CONS 3 NIL), and the process will terminate after evaluating

```
(CONS 1 (CONS 2 (CONS 3 NIL)).
```

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

A slightly related idea is called read macros. The read macro is applied at the input phase of LISP programming. A procedure can be associated with a character; when this character is recognized in the input stream, the procedure is activated. That procedure 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. For more details on both kinds of macros, see the TLC Manual. For examples of their application, see the file EXAMPLE.AIP; this file contains several annotated examples from the Artificial Intelligence Programming book.

Property-lists

As with the previous sections, this section is present to illustrate another programming concept which is unique to the LISP programmer's view of the world: the use of property lists.

A property list --also called a "p-list"--is a data structure consisting of a collection of pairs: one element of the pair is called a property name; the other element is called a property value. Typically one accesses the property list using a property name, and extracts a property value or changes that value. In this regard, a property list is similar to a more traditional record structure. However LISP p-lists have two additional and important attributes. First, they are dynamic; they may grow and shrink at run-time. This makes them an extremely flexible storage mechanism; since their storage need not be declared ahead of time. Second, this flexibility combines beautifully with LISP's program-data duality, giving rise to a technique called data driven programming.

Recall our example of algebraic simplification. There we organized the program as a large conditional expression, each branch testing for a type of term --variable, constant, or product. A similar organization was used in our description of the evaluation process. That organization can be characterized as a monolithic algorithm that tests and decomposes its input, taking actions accordingly. We can organize these problems in an orthogonal manner, viewing the fragments of the algorithm which pertain to specific data types, as in fact, <u>properties</u> of the data type itself. Thus, for example, the class of LISP variables possesses an algorithm for evaluation of any element of that class. Using LISP property lists, we can implement this idea by placing an "evaluation" property name on the property list of the class "variable", and associated with that name, a LISP function to carry out that evaluation. In general, the evaluation is performed by extracting the algorithm from the class which contains the current instance, and then applying that algorithm to that instance. This process of distributing the algorithm using the idea of objects being instances of classes, is called data driven programming; it is a most powerful programming technique.

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; arrays and arbitrary precision numbers 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 builtin 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 builtin "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 <u>table-driven</u> 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. For more details about syntax extension, see the "Examples" section in Part II. Similar techniques can be used to format output.

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.

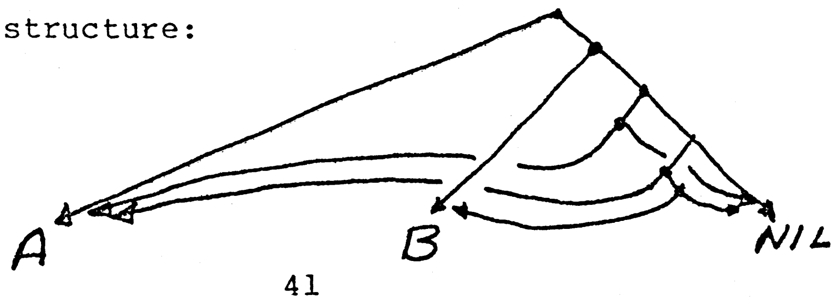
The power of LISP is truly astounding. There is not one single feature which is the source of this power; it is a blend of several aspects. In combination, these ingredients give a most powerful, but controllable programming language.

How LISP Works

This section is not a description of the implementation of any particular LISP; rather, it is an overview of several techniques which 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 interrelationships between the practical techniques of systems 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 which 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 from--but we restrict attention to the primeval reader. Functionally, there are two components to the reader; the most primitive piece is the LISP <u>scanner</u> 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. LISP atoms play a role similar to that of words in a natural language dictionary; in fact since property lists are most usually associated with atoms, 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 RATOM make every reference to a particular atom point to the same dictionary entry. This means, for example, that the list (A B (A) (A B)) would have the following structure:



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

A major implementation decision involves 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 (see the discussion of ASSOC in the TLC-LISP manual). With shallow binding, we position the value of the variable on the property list of the atom which represents the variable. In this case the search routine need only examine the property list. The "value property" is always found in the value cell of the variable; no search is required.

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") which 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 used, 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. This difficulty is called the "funarg problem", or "functional argument problem". This issue is beyond the scope of either this discussion or this implementation; again, see Anatomy of LISP for details.

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. In distinction to this are the bindings which we encounter with assignment statements. These later bindings -- called "destructive bindings"--cut through program structure as surely as to 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; 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, the 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 which 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 which 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.

Bibliography

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.

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.

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

Friedman, D., The Little LISPer, SRA Publishers, Menlo Park, CA., 1974.

Kowlaski, R., Algorithm=Logic+Control, Communications of the ACM, Vol 22, No 7, pp424-436.

Bibliography

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, pp33-71.

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.

I N D E X

Introduction to TLC-LISP

This version of TLC-LISP represents the initial strand in a sequence of powerful LISP dialects for the next generation of microcomputers. We have avoided those features of preceeding LISP's which represent historical anachronisms. The future LISP machine will be a personal computing environment with no encumbering operating system; thus that environment must be prepared to service the general computing requirements of the user. To that end we have included a full complement of arithmetic features as well as including the character and string data types and associated operations. We have allowed string-typed variables as sources and sinks for LISP input and output, respectively. For example readers and printers can use the terminal, the file system, or lists-of-strings as their targets.

With some reservation, we have retained the "dotted-pair" as the basic structured data type of LISP. The major practical benefit of dotted pairs 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. However, it may be precipitous to fly in the face of history; dotted pairs remain, but with a very strong admonition: if you program with lists, use the list primitives, not the S-expression primitives. Furthermore, when programming higher-level constructs invent names for the structure-manipulating operations that reflect the semantics of the programming task. Don't write stuff like (CONS (CADDAR X) (CDDADR Y)); its hard to read, hard to maintain, and downright anti-social. 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. The abstract approach to LISP programming, which we are advocating, is gracefully supported by LISP macros; the subject of the next paragraph.

A powerful LISP programming construct was invented after the implementation of LISP 1.5; this is the LISP macro. This represents an excellent 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. See the appropriate sections of the manual for detailed discussion.

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-expression; SELF, an elegant form of the LABEL construct derived from VLISP; and the MacLISP CATCH-THROW pair that embodies a powerful technique for non-structured exits. We have also included a version of the ancient LISP SELECT-expression, more recently re-invented as the "case-statement" in Algol-like languages. 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.

To enhance readability of TLC-LISP programs, an embedded comment convention has been included. A comment begins with a semi-colon (;) and is terminated by either a carriage return or a second semi-colon. The second convention allows comments to be embedded within arbitrary list structure. For example:

```
        (IF <predicate> ;then; <term>
                        ;else; <term>)
```

has ";then;" and ";else;" as comments which highlight the semantics of the IF-expression.

We have also 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 usually at the sacrifice of some very helpful parameter-checking information: if too many arguments are supplied, their values are discarded; if too few are supplied, the missing parameters are gratuitously bound to NIL.

A further relaxation of parameter passing is also desirable: the ability to supply an arbitrary (therefore variable) number of arguments. For example we would rather write (PLUS X Y (ADD1 Z)) than (PLUS X (PLUS Y (ADD1 Z))). Many instances of this variadic call can be accomplished by macro expansion, however the problem begs for a general solution.

Finally, a common application of the "PROG-feature" is the declaration and immediate initialization of "PROG-variables". We can accomplish all of these desirable features with variations on a small set of conventions.

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

<u>&OPTIONAL</u> <u>&REST</u> <u>&AUX</u>

with the most general formal parameter list being:

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

where any or all of these groups may be absent.

<required> is a sequence of zero or more atom names; <optionals> and <auxs> are non-empty sequences of either atoms or lists whose first elements are atoms. In this second case, the remainder of the list is to be interpreted as a value to be assigned to the variable represented in the first element. For example (X (PLUS Y N)) would mean "assign the sum of Y and N to X." Finally, <rest> must be a single atom.

The algorithm for parameter matching in this extended form is as follows:

1. First, the required parameters must be matched. If these requirements cannot be satisfied, an error is signalled.
2. If actual parameters still remain and <optionals> were catered 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 UNBOUND if no default value was supplied.
3. If after step 2, actual parameters still remain and a &REST parameter was declared, then the <u>list</u> of the remaining parameters is bound to the <rest> variable. If

no REST parameter was supplied then an error will be signalled.

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 discipline; 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, and powerful mechanism to control the passing of parameters. See the next section for a selection of examples.

Finally, in preparation for later introduction of functions as first-class objects (or as "mobile data" as discussed by V. Pratt in the August BYTE) we have consolidated the treatment of "simple value" and "function value"; there is at most _one_ "value" associated with an atom at any one time.

We have _not_ attempted to implement a "full-funarg" Z-80 LISP. Our treatment is reasonably standard: a shallow-bound stack-oriented, but robust LISP.

Introduction

The information in this section can be skimmed by the knowledgeable LISP afficionados to familarize themselves with TLC-LISP. Novices can use these examples in conjunction with the detailed TLC-LISP manual, the machine, and the description of the TLC-LISP interpreter (Part II, Section d), to develop a through understanding of LISP. Like learning to drive, the best way to learn a programming language is to do it; experiment.

A Convention or Two

We will distinguish between user input and LISP output by underlining input text.

LISP is a calculator; the default listen loop will invoke the evaluator as soon as a well-formed expression is supplied (of course this behavior may be changed by the user --see TOPLEV). If the expression is a combination, then the activation is initiated when the parentheses balance. If the expression is atomic, then we must supply explicitly an appropriate terminator. The terminator will be designated by the construction {ter}.

We will also use > as the input prompt character; again redefinition of TOPLEV can change this.

A Simple Example

To begin, type LISP <carriage return> to CPM. TLC-LISP will respond:

```
     ---- (T . (L . C)) Interpreter, version n.mm ----
     ----  Copyright (c) 1980, The LISP Company  ----
     >
```

where the > means that LISP is waiting for input. The following interchange will discover the values of the atoms T, NIL, and CONS.

```
     >T{ter}   ;recall that we underline user input
     T
     >NIL{ter}
     NIL
     >CONS{ter}
     2DF7                ;this represents the primitive
     >                   functional object, CONS.
```

Now assume we wish to evaluate the expression (CONS NIL T), 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)       ;note:  no {ter} is needed
     (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-1)! for n greater than 1
     >(DE FACT1 (N) (COND ((ZEROP N) 1)
                          (T (MUL N (FACT1 (SUB1 N))))))
     FACT1
     >(FACT1 3)
     6
     >(DE FACT2 (N) (SELECTQ N
                             (0 1)
                             (OTHERWISE (MUL N
                                             (FACT2 (SUB1 N))))))
```

```
FACT2
>(FACT2 3)
6
>(DE FACT3 (N) (FACT3* N 1))
FACT3
>(DE FACT3* (N M) (IF (ZEROP N)
                      M
                      (FACT3* (SUB1 N) (MUL N M)))))
FACT3*
>(FACT3 3)
6
>(DE FACT4 (N) (DO ((M 1 (MUL N M))
                    (N N (SUB1 N)))
                   (((ZEROP N) M)) ))
FACT4
>(FACT4 3)
6
>(SETQ FACT5 (LAMBDA (N) (IF (ZEROP N) 1 (MUL N (SELF
(SUB1 N))))))
(LAMBDA (N) (IF (ZEROP N) 1 (MUL N (SELF (SUB1 N)))))
>(FACT5 3)
6
```

Definition 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-1.

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.

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.

Also note that we may use our extended parameter description syntax to simplify the FACT3-FACT3* example:

```
>(DE FACT3!  (N &OPTIONAL (M 1))
     (IF (ZEROP N)
         M
         (FACT3!  (SUB1 N) (MUL N M)))))
FACT3!
```

FACT3! will supply a value of 1 for M when FACT3! is called initially.

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, it is more natural to define the DO 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.

So far all these examples could be formulated quite easily in most other modern programming languages. In FACT5 we begin to see some of the power of LISP. In this example, we construct a functional object using the LAMBDA operator and assign that object to the variable FACT5 (if you don't like the name "LAMBDA", think "PROC"). The ability itself, to construct functional objects, is novel; to pass them around as values in the language is most unique (in fact most LISP systems do not treat functional values with the regularity that TLC-LISP possesses). Given that FACT5 has a functional value, we can apply it in the context of a combination like (FACT5 3).

In the body of the functional object assigned to FACT5 we find the name "SELF". "SELF" is a way to refer to the functional object which contains the SELF-reference. This allows us to construct an anonymous recursive functional object. Assigning it to FACT5 only gives it a name (note it would not equivalent to write:

```
>(SETQ FACT5 (LAMBDA (N) (IF (ZEROP N)
                             1
                             (MUL N
                                  (FACT5 (SUB1 N))))))
```

for if we subsequently performed (SETQ FACT6 FACT5) we would not have successfully transferred the functional object to FACT6. Initially FACT6 would work, but after an assignment like (SETQ FACT5 NIL) FACT6 would fail. SELF solves this problem, separating the transitory naming from the object itself.

Regardless of this extra power, the examples come from the traditional numeric domain. It would be most instructive to see the data structuring facilities in action. The next section will discuss such examples.

Parsers

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 <u>highly</u> <u>readable</u> parser for those people. It would be a non-trivial exercise to encode this parsing scheme in another language without scarificing 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 <u>SCAN</u>. 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))
                                 ( \( (CONS (READ-REST)
                                             (READ-REST)))
                                 ( \) NIL)
                                 (OW (CONS OBJ
                                           (READ-REST))))
```

The actual parser in TLC-LISP is more complex. It performs error checking, allows backspacing and correction, 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 rediscover the structure of the text being input to the system (of course, one might ask "It seems backward to rediscover structure; the programmer knew the structure to begin with, yet she must present the linear text string to the machine, only to force the machine to uncover what was already known"). 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 <u>right</u> binding power of O1 is greater than the <u>left</u> 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 a given 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 phase:

z = x+y*z ? would parse as (= z (+ x (* y z))) with the appropriate delimiter tables set for =, +, and *; these tables are discussed in the Input-Output section of the manual. 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))))
           (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))))))
```

```
where:
(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, pp316-333, 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.

The TLC-LISP Manual

This section is a complete catalog of the built-in 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. <object> represents an element of the class <object>

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 expected to be atomic.
Examples: A A123 AGA-MEM-NON but not 1DERFUL.

<number> is expected to be numeric.

<fix> is expected to be an integer.
Integer values are 14-bit quantities, whose input and

output characteristics are determined by the values of INPUT-BASE and OUTPUT-BASE, respectively. These BASEs may take on values, two through ten, and sixteen. Undecorated numbers are always taken as decimal; if a number is preceded by "#" then it taken as "base INPUT-BASE"; if it is preceded by "#[<fix>] then <fix> is used as the base. OUTPUT-BASE is used when printing values; if that base is 10, then the undecorated form is printed; otherwise the prefix #[<n>] is used where <n> is the value of OUTPUT-BASE.

```
12       (base: 10)
#44      (base: current value of INPUT-BASE)
#[4]23   (base: 4)
```

<flt> is expected to be a floating point number.
For example, 1.23 and 2.718E-4 but not 1 or "1" or A.

<string> is expected to be a string.
For example, "abcABC" and "123ASD" but not A or \A.

<char> is expected to be a character object.
For example, \A and \1 but not A or "A".

<sexpr> is any well-formed LISP S-expression, atomic or composite.
For example, T, (A . B), and (A B C D) but not (A .).

<list> is expected to be a list object, empty or non-empty, but not atomic.
For example, (A B C D) but not (A . B) or A.

TLC-LISP encodes in tables, the information about what constitutes an atom, number, or string. For some advanced applications it may be convenient to change these tables. See the section in TLC-LISP's input and output.

4. It is also convenient to specify that an <object> be a member of a semantic LISP class.

<var> is expected to be an element which can be used as a variable. Therefore numbers are disallowed as well as the LISP reserved words: T, NIL; and the very basic primitives of LISP: CAR, CDR, CONS, EQ, ATOM, COND, and LAMBDA. Thus the other LISP "library" functions are available as variable name with the caveat that re-definition of library entries must be done with great care; the results are not guaranteed.

<fn> is expected to be an object which can be

interpreted as a LISP function; for example, <fn> may be a variable which is bound to a function object, or <fn> may be an anonymous LAMBDA expression.

<pred> is a LISP form that is expected to be used as a predicate; that is, its evaluation yields a LISP truth-value, NIL or non-NIL.

<form> is a LISP expression that is expected to be evaluated. That is, it meets LISP's syntax 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. Note <form> makes no claims about the evaluation; it could produce a value, cause an error message, or even fail to terminate.

These lists of LISP objects are not meant to be exhaustive, only indicative.

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 for a definitional description is:

(<name> {<arguments>}) <type>

where <name> is the name of the built-in 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. A few built-in functions are of type LSUBR, meaning they are call-by-value, but will take an arbitrary number of arguments.
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.
b. In the case of FSUBRs, as a pattern to be matched against the textual form of the argument.

For example given:
(FOO <atom> <number> <sexpr>) SUBR

a call (FOO (CAR X) (ADD1 22) 'A) would fit the constraints provided that (CAR X) evaluated to an atomic value since the _value_ of (ADD1 22) is a number and the _value_ of (QUOTE A) is a symbolic expression; the body of FOO would receive three values.
Whereas: (BAR <atom> <number> <sexpr>) _FSUBR_ could be called like (BAR X 22 'A). The body of BAR would see a single argument (X 22 (QUOTE A)) and would decompose it accordingly. Note: X _is_ an atom; 22 _is_ a number, and the list (QUOTE A) is a symbolic expression.

For a more detailed discussion of the LISP calling styles see the section on LISP evaluation.

Function 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 the basic strategies involved in call-by-value definitions and special forms. Here we introduce techniques for adding new functions to the LISP library; call-by-value functions introduced this way are called EXPRs; similarly, 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

Makes a call-by-value function out of the parameters and the forms; it then installs that definition as the value of the <var>. The value of DE is <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; see the section, Introduction to TLC-LISP.

However, for a simple example:

```
        (DE FACT (X) (IF (ZEROP X)
                         1
                         (MUL X (FACT (SUB1 X)))))
```

is a definition of the venerable factorial function.

For a more complex example, consider:

```
        (DE WHIZ (X &OPTIONAL (Y (CONS 5 X))) Y)
```

then

```
        (WHIZ 2 7) => 7, and (WHIZ "ab") => (5 . "ab")
```

(DF <var> (<param> {&AUX {<params>}}) {<form>}) FSUBR

Similar to DE, but for call-unevaluated functions (also called special forms or FEXPRs). Note the single <param>. When the special form <var> is applied, the list of unevaluated parameters is bound to <param>. For example:

If we make the following definitions:

```
        (DF FEXAMPLE (X) (CAR X))

        (DF FEXAM (X) X)

then    (FEXAMPLE 1 2 4) => 1
and     (FEXAM 1 2 3)    => (1 2 3)
```

```
      whereas (DE EXAM (X) X) results in:

                (EXAM (ADD1 2)) => 3,
      but       (FEXAM (ADD1 2)) => ((ADD1 2))
```

we could have defined QUOTE as

```
            (DF QUOTE (L) (CAR L))
```

Note too the possibility for AUX-parameters. Though a DF may have at most one required parameter, and no OPTIONALS or REST parameters, it may specify a set of local variables to be allocated at entry to the special form. For further information see the discussion in PART II, section a.

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. Such "DF"'s are used illustratively throughout this manual. Look at them; however, if the distinction between call-by-value functions and special forms is still confusing, see the section titled "Evaluation".

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. Thus:

(DM <var> (<param> {&AUX {<params>}}) {<form>})   FSUBR

Associate the macro definition, represented in (<param> {&AUX {<params>}}) {<form>}), with the name <var>. As with DF, DM may also specify auxiliary parameters.

A macro may be called with an arbitrary number of arguments since, when a macro is invoked, it is the <u>text</u> of the whole call that is bound to that single parameter. For example, if we define a macro TEST,   (DM TEST (L) ...),

the call (TEST (CAR X) 4 'NOW) will bind the <u>list</u>

(TEST (CAR X) 4 (QUOTE NOW)) to the variable L.

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 <u>new</u> expression; since, as we leave the macro call, that expression is evaluated by the interpreter 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). Here is a simple example:

Let NCONS be a macro defined as:
```
      (DM NCONS (L) (LIST 'CONS (CADR L) NIL))
```

Consider a call (NCONS 6):

The list (NCONS 6) gets bound to L; the evaluation of the body gives a list (CONS 6 NIL). Finally that list get evaluated and (NCONS 6) returns (6) as value.

Many of the traditional uses of special forms can be handled by macros. For example some LISP implementations which don't have LSUBRs define LIST as a macro:

```
(DM LIST (L) (COND ((NULL (CDR L)) NIL)
                   (T (CONS 'CONS
                            (CONS (CADR L)
                                  (CONS (CONS (CAR L)
                                              (CDDR L))
                                        NIL))))))
```

The alternative is to define LIST as a special form and require that the implementation of LIST handle all of the parameter evaluation.

Macros are able to express a complex behavior in terms of simple transformations which can be carried out on the program text. In the LIST example, we have a "function" which appears to the programmer as one which will take an arbitray number of arguments. Yet when LIST is called the evaluator expands the macro to a nest of CONSes. Thus macros can be used to obscure many implementation details; they are an exceptionally powerful technique for "information hiding". Learn to use them. For further examples of macros see the "Evaluation" section, and the discussion of RPLACB.

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. Note, however, that if these functions are used in a context where the atom name has been lambda-bound then the old lambda binding will reappear when we exit that context.

There are also two operations, LAMBDA and FLAMBDA, that are used to make more temporary function definitions.

(LAMBDA <parameters> {<form>}) FSUBR
makes a functional object whose formal parametrs 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 and therefore are often called anonymous lambdas. 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 LISP, even to the point of using them as argument to functions and returning them as values of functions. Currently, TLC-LISP supports only a subset of the full power of functional objects; future implementations will rectify that situation.

One application of lambda expressions appears within the implementation of DE. DE 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 (M) (LIST 'SET (CADR M)
                      (CONS 'LAMBDA (CDDR M))))
```

(FLAMBDA (<var> {&AUX {<params>}}) {<form>}) FSUBR
is similar to LAMBDA, but constructs an anonymous special form.

(MLAMBDA (<var> {&AUX {<params>}}) {<form>}) FSUBR
is used to construct a macro definition. MLAMBDA may not be used anonymously since part of the macro call is the name of the macro.

Finally, a "syntactically sugared" form of the LAMBDA-expression is provided in the LET-expression:

(LET ({(<var> <form-1>)}) {<form-2>}) FSUBR
abbreviates:

```
((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. For example our previous LAMBDA-example could be expressed as:

```
(LET ((X 3) (Y 5)) (ADD X Y))
```

Functions to Perform Evaluation

The actual 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 an expression is read, then evaluated; the result is printed and the top level waits for the next input. This top-level loop is called the "READ-EVAL-PRINT" loop. This gratutious evaluation often suffices, but sometimes it is convenient to impose other evaluation regimes.

One also needs to be able to 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
```

Assume that X has value A, and A has value 4;

then: (EVAL 'X) => A since the actual parameter passed to EVAL is the atom X.

and: (EVAL X) => 4 since the actual parameter passed to EVAL is the atom A.

```
        (EVAL '(CAR '(A . B))) => A

        (EVAL '(FIRST '(1 2 3))) => 1

        (EVAL (LIST 'CAR
                    (LIST 'CONS X 'X))) => 4  since the value
```

passed to EVAL is (CAR (CONS A X)).

(EVLIS ({<form>})) SUBR

Forms 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.

As an example of EVLIS we have:

```
(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 <exp>'s, returning the value of the first <form>.

For example:

```
(PROG1 (CONS 1 3) 4) => (1 . 3)

(PROG1) => NIL
```

(PROGN {<form>}) FSUBR

Similar to PROG1, but returns the value of the LAST Form.

For example:

```
(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 '. The effective definition is:

```
(DF QUOTE (L) (CAR L))
```

(TOPLEV) SUBR

TOPLEV is the name of the function that controls the user interface. It is initially defined to be approximately:

```
(DE TOPLEV (&AUX INP OUT)
     (DO ()
         (NIL)
         (PRINT ">" CONSOLE 2)  ;-- PRINT a prompt,
         (SETQ INP (READ CONSOLE))      ;-- READ an expression,
         (SETQ OUT (EVAL INP))  ;-- EVALuate that form,
         (PRINT OUT CONSOLE 0)  ;-- PRINT the value, and
                                 ))    ;-- loop back
```

NOTE: the body is expressible without the &AUX variables as:

```
        (PRINT ">"CONSOLE 2)
        (PRINT (EVAL (READ CONSOLE)) 0)
```

For a discussion of the parameters to READ and PRINT, see the section on Input and Output. Of course, the user may supply a different TOPLEV --simply redefine TOPLEV. A certain amount of caution should be exercise, however; bugs in a new TOPLEV might destroy the system.

Function Manipulating Functions

The functions in this section operate with one or more parameters being a functional object. Note: such parameters are expected to be functional objects, not objects which evalute to a functional object.

(APPLY <fn> <list>)   SUBR

Apply the function <fn> to the list of evaluated arguments represented in <list>.
For example:

```
        (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 (5 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)
```

Using the bindings: X has value 4,

```
        (APPLY CAR (LIST (CONS X 'X))) => 4
```

APPLY, like EVAL, seldom need be explicitly applied. In fact, though APPLY can be used with SUBRs and EXPRs, APPLY may not be used with a special form or macro in the <fn> position.

(MAP <fn> <list>)   SUBR

Apply the function <fn> successively to <list> and its tails. The value returned is ().

```
(DE MAP (FN L)
        (IF (NULL L)
            ()
            (FN L)
            (MAP FN (REST L)))))
```

(Note the implicit application of FN to L)

```
For example (MAP PRINT '(A (B C) D)) gives:
            (A (B C) D)
            ((B C) D)
```

```
                (D)
                NIL
```
where the final NIL is the value returned.

(MAPLIST <fn> ({<form>})) SUBR

Apply the function <fn> succesively to ({<form>}) 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)))))
```

and, for example, we could define EVLIS as:

```
(DE EVLIS (L)
       (MAPLIST (LAMBDA (X) (EVAL (FIRST X)))
                L))
```

(CLOSURE <fn> ({<var>})) SUBR

This is a simplified version of LISP's FUNARG. The list of <var>s and current values are associated with the functional object <fn> in such a way that they will be established as the current bindings whenever the CLOSURE-object is applied as a function.

```
(LET ((Y 2))
     (LET ((F (CLOSURE (LAMBDA (X) (CONS X Y))
                '(Y)))
           (X 4)
           (Y 'A))
          (APPLY F (LIST Y))))    =>  (A . 2)
```

```
whereas (LET ((F (LAMBDA (X) (CONS X Y)))
              (X 4)
              (Y 'A))
             (APPLY F (LIST Y)))    => (A . A)
```

Control Structure Functions

Call-by-value, recursion, and the parameter evaluation mechanism impose a order in which LISP computations are carried out. A programming language also needs a mechanism to control which computations are to be executed. This is done in LISP with the conditional expression.

Control structures are based 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 falsity, and any non-NIL value is taken as truth. See the section Recognizers and Predicates for further discussion.

TLC-LISP includes two forms of the conditional expression:

(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>}s is evaluated and the value of the IF is the value of the last <form2>.

```
        (IF (CAR X)
            1
            2)
```

gives value 1 if (CAR 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>s.

The most general conditional form in LISP is the "COND":

(COND (<pred1> {<form1>}) ... (<predn> {<formn>})) FSUBR

The object (<predi> {<formi>}) is called a clause. The evaluation of a COND-expression 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. In this case the <formn>'s are able to handle all exception cases. The use of T in this context is therefore read as "otherwise".

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 computationally convenient. 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.

Constructs like:

```
        (COND ((SETQ XX <pred>) XX)
                ...
                                    ) are cretinous.
```

This usage involves both a marginal LISP coding trick, and requires the use of a variable XX which must be specified globally to the COND. The effect is better described by:

```
        (COND (<pred>)
                ...
                        ).
```

Here is an example of COND usage:

```
        (COND ((BAR X Y) (WHIZ U X))
              ((BAZ X) (ZAM X) (MAZ U 2) (TLC 2 B))
              ((FROB X))
              (T (WALDO U)))
```

TLC-LISP also supplies forms of the Boolean operations AND and OR which can "short circuit" their evaluation.

(OR {<form>})   FSUBR
Evaluate the sequence of <form>s from left-to-right, terminating that process if one returns a non-NIL value; return that value as the value of the OR-expression. If no <form> gives a non-NIL value, then the value of the OR is NIL.
For example:

```
    (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"; further note that the expression (CAR 1) --which would yield an error-- never gets evaluated. A binary form, (OR X Y), could be considered an abbreviation for:

```
        (COND (X) (T Y))
```

(AND {<form>}) FSUBR

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.

```
For example, (AND (CONS 1 2) NIL (CAR 1)) => NIL
        and  (AND (CONS 1 2) T 4 (ADD1 2)) => 3
```

again, (AND X Y) abbreviates a conditional expression:

```
        (COND (X Y) (T NIL))
```

Finally, for completeness, we include the NOT function.

(NOT <form>) SUBR

Returns NIL if <form> is non-NIL, and T otherwise.

Though LISP is known for its penchant for recursion, every LISP has included control structures for describing computations in an iterative fashion. Indeed, even the first LISP, LISP 1 of 1960, had a construct which was identical to the later invented ALGOL "case-statement"; LISP called it SELECT. TLC-LISP includes a form of this construct:

(SELECTQ <form> {(<sexpr> {<formi>})}) FSUBR

The value of <form> is compared succesively against each <sexpr>; the <sexpr>s are not evaluated. The type of match is determined by the structure of <sexpr>. If <sexpr> is an atom other than T, the match uses the predicate EQ; if <sexpr> is a list then the match uses MEMQ; if the <sexpr> 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.

For example:

```
        (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.

(SELF {<form>})   LSUBR

SELF evaluates {<form>} in the context of the last (dynamically) surrounding lambda expression. This is a generalization of the LISP label-operator, allowing recursive definitions without explicit naming. For example:

```
        (LAMBDA (N) (IF (ZEROP N)
                        1
                        (MUL N
                             (SELF (SUB1 N)))))
```

expresses the factorial function.

(CATCH <atom> {<form>}) FSUBR

(THROW <atom> {<form>}) FSUBR

This pair of functions operates together to supply a non-structured type of function exit. These functions are a slight generalization of the MACLISP CATCH and THROW operators, which in turn is a generalization of the LISP 1.5 ERROR-ERRSET pair.

When a CATCH expression is entered, the <atom> is noted and the body, {<form>}, is evaluated as a sequence of expressions. If, during that evaluation, an expression (THROW <atom> {<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. For example:

```
(CATCH EXIT (MAP (LAMBDA (X) (AND (NUMBERP (FIRST X))
                                  (THROW EXIT 'YES)))
```

```
                    '(A B 2 C)) 'NO)
                                          => YES
```

If a THROW expression is encountered which does not have a dynamically surrounding CATCH expression with a matching <atom>, then an error is signalled.

The CATCH-THROW pair is particularly useful for effecting an immediate return from a sub-computation without requiring an explicit exits up through all the intervening levels of computation. Such a strategy would require all functions involved to include explicit tests for exit conditions and corresponding function-exit clauses.

TLC-LISP also offers iterative sequencing mechanisms which blend the traditional LISP style with many of the modern ideas of structured expression of programming concepts. Of particular note is the DO-expression.

```
(DO ({(<var> <init> <iter>)})
    ({(<exitp> {<exitval>})})
    {<form}}                        FSUBR
```

We will discuss the most general form of DO first, and follow that with an analysis of several useful degenerate subcases. There are four basic parts to the semantics of the DO expresion:

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 surounding 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 next.

4. The iterate phase. Following the body phase, we evaluate the <iter> forms; again, this is done in parallel. Only now, we assign these values to their corresponding <var> rather than lambda-bind

them. 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:

a (DO () ...): If there are no var-init-iter triples, we have no local variables. The execution of the DO involves only the body and the exit-tests.

b (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).

c (DO ... (NIL) ...): In this case the predicate will never be true; the DO will loop without end (unless it contains a THROW form.)

d (DO ... () ...): In this case the body is executed only once.

e (DO ... ...): If no body is present then we pass directly to the iterate phase.

Below are several other control structures expressed as equivalent DO formulations:

(LET ((var init)) body) is (DO ((var init)) () body)

(WHILE pred body) is (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.

Recognizers and Predicates

As we mentioned in the Control section, all LISP functions can be used as predicates; the truth-values in TLC-LISP (and most other LISP implementations) map 'true' and 'false' to non-NIL and NIL, respectively. This is more than 'just a programming trick', but 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 exits, 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 involve 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.

<u>(ASSOC</u> <u><atom></u> <u>({(<atomi> . <sexpri>)}))</u> SUBR
ASSOC searches the list ({<atomi> . <sexpri>}) for a match of <atom>. If one is found, the remainder of the list ({(<atomi> . <sexpri>)}) beginning with 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)))))
```

For example:

```
        (ASSOC 'TLC '((FOO . LOSE) (TLC . WIN) (NERD . LOSE)))

                => ((TLC . WIN) (NERD . LOSE))
```

<u>(MEMQ</u> <u><atom1></u> <u>({<atom2>}))</u> 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 (see the note at the end of MEMQ' discussion). MEMQ's definition follows:

```
(DE MEMQ (A L)
        (IF (OR (NULL L) (EQ (FIRST L) A))
            L
            (MEMQ A (REST L))))
```

For an example consider:

```
        (MEMQ 'A '(1 2 3 A B C)) => (A B C)
```

Though ASSOC and MEMQ are defined in terms of <atom>s, they may be applied with <expr>s in those positions. Note that both functions use EQ. Since EQ is defined to test only for identity of objects, EQ will respond with T for (EQ X X) regardless of the type of X. Care must be exercised since (EQ '(A) '(A)) will give NIL; if you don't understand this, dont't use <expr>s in the <atom> positions.

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 value, each LISP object has a distinguishable type. The LISP recognizers are predicates which the programmer can use to determine the type of a value.

(ATOM <sexpr>) SUBR

ATOM returns T if <sexpr> is not a composite object; it return NIL otherwise. Literal atoms, 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)
```

(LISTP <sexpr>) 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) => NIL
```
(even though NIL represents the empty list)

<u>(SYMBOLP</u> <u><sexpr>)</u>   SUBR

<u>(NUMBERP</u> <u><sexpr>)</u>   SUBR

<u>(FIXP</u> <u><sexpr>)</u>   SUBR,

<u>(FLOATP</u> <u><sexpr>)</u>   SUBR,

<u>(CHARP</u> <u><sexpr>)</u>   SUBR,

and
<u>(STRINGP</u> <u><sexpr>)</u>   SUBR

These recognizers check for an occurrence of a literal atom, number, a fixed point number, a floating point number, a character, or a string, respectively.

```
(SYMBOLP 4) => NIL
(SYMBOLP "BAC") => NIL
(SYMBOLP 'A) => T

(NUMBERP 4) => 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
```

<u>(PROCP</u> <u><sexpr>)</u>   SUBR

This recognizer returns the type of <sexpr> if <sexpr> is a functional object. Valid values are SUBR, LSUBR, FSUBR, EXPR, FEXPR, CLOSURE and MACRO. If <sexpr> is not a functional object, NIL is returned.

```
(PROCP PROCP) => SUBR

(PROCP COND) => FSUBR

(PROCP FOO) => NIL        (or an error)
```

<u>(BOUNDP</u> <u><atom>)</u>   SUBR

returns T if <atom> has a value other than UNBOUND.

```
(BOUNDP 'CONS) => T
```

(NULL <sexpr>) SUBR

NULL returns T just in the case that <sexpr> is the empty list.

```
(NULL '(A)) => NIL
(NULL (REST '(A))) => T
(NULL (NULL '(A))) => T
(NULL 3) => NIL
```

(EMPTY <sexpr>) SUBR

EMPTY returns T just in the case that <sexpr> is the empty string.

```
(EMPTY "ABC") => NIL

(EMPTY '(TRASH . CAN)) => NIL

(EMPTY "") => T
```

(TYPE <sexpr>) SUBR

This is a general type-extraction function, returning an atom that describes the type of the argument <sexpr>.

```
(TYPE 'TYPE) => ATOM

(TYPE TYPE) => SUBR

(TYPE (CONS 1 2)) => LIST

(TYPE (LAMBDA (X) 1)) => EXPR

(TYPE '(LAMBDA (X) 1)) => LIST
```

Besides the recognizers, TLC-LISP also includes some general predicates which implement forms of the equality relation.

(EQ <sexpr1> <sexpr2>) SUBR

EQ tests <sexpr1> and <sexpr2> to see if they are the same storage location. Since atoms are stored uniquely in LISP, EQ satisfies the 'eq' predicate as expected in LISP. EQ will also return T if <sexpr1> and <sexpr2> are the identical object. For example:

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

```
                (EQ 'A 'B)        => NIL

                (EQ "AB" "AB")    => NIL

                (EQ '(A B) '(A B)) => NIL

        but     (SETQ L '(A B))
        followed by:    (EQ L L) => T
```

(EQUAL <sexpr1> <sexpr2>) SUBR
This is the general equality predicate in LISP. returning T just in the case that <sexpr1> and <sexpr2> are the same tree-structure.

The definition of EQUAL can be sketched as:

```
        (DE EQUAL (X Y)
                (OR (EQ X Y)
                    (AND (EQUAL (CAR X) (CAR Y))
                         (EQUAL (CDR X) (CDR Y)))))
```

For example:

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

                (EQUAL '(A B) '(A B)) => T

                (EQUAL "ABC" "ABC") => T
```

Selection Functions

Given a composite data structure, we need tools for manipulating the components of that structure. This section deals with operations to select components; the next section discusses how to construct new structures, and two sections ahead we address the issue of modifying existing structures.

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. Some such tests are built into TLC-LISP --for example CAR and CDR of atoms is disallowed-- however, consistent with LISP's open nature, it is generally the programmer's responsibility to control the tool.

Selector Functions for Dotted Pairs

(CAR <sexpr>) SUBR

This function selects the first component of the dotted pair represented in <sexpr>.
For example:

```
(CAR '(A . B)) => A
```

also (CAR '(A B)) => A, since 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 '(A . (B . C))) => (B . C)
```

(C...R <sexpr>) SUBR

These (twelve) functions give the usual CAR-CDR chains of LISP selection operations.

```
(CADR '((1 . 2) . (3 . 4))) => 3

(CDAR '((1 . 2) . (3 . 4))) => 2

(CDDR '((1 . 2) . (3 . 4))) => 4

(CAAR '((1 . 2) . (3 . 4))) => 1
```

Selector Functions for Lists

To help reinforce the conceptual distinction between dotted pairs and lists, we have included selector functions which are supposed to be applied only to lists. Of course, LISP will not enforce the distinction between dotted pairs and lists; that restraint must come from within. Such restraint _must_ be cultivated early else, as programming tasks become _more_ audacious, the programmer will become mired in a sea of CARs and CDRs.

(FIRST <list>) SUBR

<list> is a non-empty list and FIRST selects its first component

```
(FIRST '(A B C D)) => A
```

(REST <list> &OPTIONAL (<fix> 1)) SUBR

<list> is a non-empty list; <fix> is a non-negative integer. REST returns the 'tail' of <list> beginning at the <fix>-th element.

```
(REST '(A B C D) => (B C D)

(REST '(A B C D) 2) => (C D)

(REST '(A)) => NIL
```

(NTH <list> <n>) SUBR

NTH returns n-th element of <list>; if there are less than n elements in the list, NIL is returned; if n is less than one, an error is signalled.

```
(DE NTH (L N)
        (IF (LE N 1)
            (FIRST L)
            (NTH (REST L) (SUB1 N))))

(NTH '(A N T I F R E E Z E) 4) => I
```

(LENGTH <list>) SUBR

This returns the length of the list <list>.

For example:

```
(LENGTH '(1 2 3 4)) => 4

(LENGTH NIL) => 0
```

LENGTH could be defined as:

```
(DE LENGTH (L) (LENGTH1 L 0))
        where:
(DE LENGTH1 (L N) (IF (NULL L)
                      N
                      (LENGTH1 (REST L) (ADD1 N))))

        or:

(DE LENGTH (L) (DO ((N 0 (ADD1 N))
                    (L L (REST L))
                    (((NULL L) N))   ))
```

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 <string> &OPTIONAL <fix1> <fix2>) SUBR

This function makes a new string EQ to the substring of <string> beginning with the <fix1>-th character and containing <fix2>-th succeeding characters. If <fix1> and <fix2> are missing, <string> is copied; if <fix2> is missing it defaults to the length of <string>.

```
(SUBSTRING "ABCDEF" 4) => "DEF"

(SUBSTRING "ABCDEFG" 4 3) => "DEF"

(SUBSTRING "1" 0) => ""     the empty string.
```

(GETCHAR <string> <fix>)

This selects the <fix>-th character from <string>.

```
(GETCHAR "ABC" 2) => \B
```

(STRSIZE <string>)   SUBR

This function returns the number of characters in <string>.

```
(STRSIZE "ABCD") => 4

(STRSIZE (SUBSTRING "ABCDEF" 4)) => 3

(STRSIZE "") => 0
```

Constructors

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 general name for such a function is a constructor.

Constructors for Dotted Pairs

(CONS <sexpr1> <sexpr2>) SUBR

This constructor makes a new dotted pair whose CAR-branch is <sexpr1> and whose CDR-branch is <sexpr2>.

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

(CONS "A" '(A . B)) => ("A" A . B)
```

where the printer has formatted the output in semi-list form.

```
(CONS (ATOM 'A) (ATOM '(A))) => (T)   i.e., (T . NIL)
```

(SUBST <sexpr1> <sexpr2> <sexpr3>) SUBR

This function substitutes <sexpr1> for every occurrence of <sexpr2> in (a copy of) <sexpr3>.

```
(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)))))

(SUBST 'C 'A '((1 . A) (A B) C)) => ((1 . C) (C B) C)
```

(COPY <expr>) SUBR

This function makes a copy of its argument; thus:

```
(DE COPY (X) (IF (ATOM X)
                 X
                 (CONS (COPY (CAR X))
                       (COPY (CDR X)))))
```

```
or:     (DE COPY (X) (SUBST 0 0 X))

note:   (EQ X (COPY X)) => T if X is atomic, otherwise => NIL

but,    (EQUAL X (COPY X)) => T, always.
```

Constructors for Lists

(CONCAT <sexprl> <list>) SUBR

This constructor expects a list in its second argument position; it makes a new list object with <sexpl> as its FIRST element, and has <list> as its REST-component. In terms of the traditional implementation of LISP, CONCAT and CONS are equivalent.

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

(CONCAT 'A NIL) => (A)
```

(LIST §<sexpr>†) LSUBR

This constructor makes a list out of the values of its arguments. This function can be expressed as a macro over CONS.

```
(LIST (CONS 1 2) (CAR '(A . B)) (REST '(A B)))

                  => ((1 . 2) A (B))
```

(APPEND <listl> <list2>) SUBR

This function makes a new list whose initial segment consists of the elements of <listl> and whose final segment is the list <list2>. APPEND will copy the elements of <listl>; 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)
```

Constructors for Strings

(STRING {<string> 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"
```

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 these operations can create circular list-structure, which can cause difficulty for a simple list-printer. A more subtle difficulty can arise in the "aliasing problem"; for details see the section titled "Evaluation."

(RPLACA <sexpr1> <sexpr2>) SUBR

RPLACA, from 'RePLace the CAr of', expects <sexpr1> to be a dotted-pair or a non-empty list; it replaces the CAR part of <sexpr1> with <sexpr2>. The value returned is the modified <sexpr1>.

```
For example,     (RPLACA  '(A B) 'C)  =>  (C B)

or consider,     (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 <sexpr1> <sexpr2>) SUBR

This operation replaces the CDR-part of <sexpr1> with <sexpr2>. As with RPLACA, RPLACD expects <sexpr1> 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-atomic.

```
(DE RPLACB (X Y) (RPLACA X (CAR Y))
                 (RPLACD X (CDR Y))))
```

This function is useful in defining 'self-destructive' macros or 'displacing' macros. For example, if we wanted to define (IS-DOG X) to be equivalent to
(EQ (CAR X) 'DOG), we could write:

```
   (DM IS-DOG (X) (RPLACB X (LIST 'EQ
                                  (LIST 'CAR
                                        (CADR X))
                                  '(QUOTE DOG))))
```

or we could define a destructive macro NEQ to mean NOT-EQ by:

```
(DM NEQ (L) (RPLACB L (LIST 'NOT
                            (LIST 'EQ
                                  (CADR L)
                                  (CADDR L)))))
```

Note: you should not use the functions in this section until you understand how these macros work!

(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
                      (LET (L L1)
                        (IF (NULL (REST L))
                            (PROGN (RPLACD L L2) L1)
                            (SELF (REST L))))))
```

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

or
```
(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)   !!
```

Notice that NCONC can be used to make circular list structure: (NCONC X X). Such structures must be printed, traversed and copied with great care.

(FREVERSE <list>) SUBR

This is a 'fast' 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; for example:

```
        (SETQ X '(A B C))  => (A B C)
        (SETQ Y (REST X))  => (B C)

now     (FREVERSE Y) => (C B)
and     Y => (C B),
but     X => (A) !
```

String Modifiers

(REPLACE <string1> <string2>) SUBR
<string2> replaces an equivalent number of character positions in <string1>.

Functions to Modify the Environment

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 (or LET or DO) expression 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, only as with all LISP forms, they return a value; therefore they are expressions rather than "statements".

(SETQ {<var> <form>}) FSUBR

Each <var> is bound to the value of its corresponding <form>; the evaluation proceeds sequentially, rather than in parallel as in the DO-expression. The binary form of this construct is analogous to the traditional 'assignment statement' of most programming languages. However, since every LISP construct is an expression, the value of the SETQ is the value of the last <exp>.

```
(SETQ X 4 Y 'A) => A
X => 4
Y => A   as expected.
```

```
Now evaluate: (SETQ X 6 Y (CONS X Y)) => (6 . A),  not (4 . A)

and                X => 6
                   Y => (6 . A)
```

(SET <form1> <form2>) SUBR

This is a generalized assignment expression; here, both <form1> and <form2> are evaluated. <form1> is expected to evaluate to a <var>; that atom is assigned the value of <form2>. For example (SET (QUOTE X) <exp>) is the same as (SETQ X <exp>).

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

Now     (SET (FIRST X) (CONS X 1)) => ((A B) . 1)
        A => ((A B) . 1)
        X => (A B)
```

Most common usages of the assignment operators involve SETQ, not SET.

(UNBIND <var>)   SUBR

This function sets <var> to the distinguished atom UNBOUND.

(POP <var>)   FSUBR

This function is used for destructive traversal of the list bound to <var>. Each call on POP returns the first element of the list while setting the list to REST of the list. For example:

```
          (SETQ X '(1 2 3 4))

Now       (POP X) => 1
          X => (2 3 4)

and another: (POP X) => 2
with         X => (3 4)
```

(PUSH <var> <form>)   FSUBR

This function is used in conjunction with POP; PUSH places the value of <form> on the front of the list bound to <var>.

```
        (SETQ SIMON  '(GEORGE  BERNARD))

                => (GEORGE BERNARD)

        SIMON => (GEORGE BERNARD)

now:    (PUSH SIMON 'SHAW) => (SHAW GEORGE BERNARD)

and:    SIMON => (SHAW GEORGE BERNARD)
```

Functions to Manipulate 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 literal atom. Therefore one can think of an atom as a 'dictionary entry' and the attribute-value pairs play the role of the various 'parts of speech' and associated meanings. For a more complete discussion of the role of property lists in LISP programming see the section titled "Property Lists".

(PUTPROP <atom> <ind> <expr>) SUBR

<atom> is a literal atom, <ind> is an atom, and <exp> is placed on the property list of <atom> under the attribute <ind>. Any previous value associated with <ind> is destroyed. The value returned is the value of <expr>.

```
            (PUTPROP 'WALDO 'AGE 47) => 47
```

(GETPROP <atom> <ind>) SUBR

<atom> is a literal atom; <ind> is an atom. The property list of <atom> is searched for the indicator <ind>; 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

now     (PUTPROP 'WALDO 'CHILDREN NIL) => NIL

and     (GETPROP 'WALDO 'MARRIED)  => NIL
        (GETPROP 'WALDO 'CHILDREN) => NIL
```

(REMPROP <atom> <ind>) SUBR

This function removes the latest attribute-value pair associated with <ind>; if none existed, NIL is returned. The value of REMPROP is the removed value.

```
        (REMPROP 'WALDO 'AGE) => 47
and now (GETPROP 'WALDO 'AGE) => NIL
```

(ADDPROP <atom> <ind> <expr>) SUBR

Similar to PUTPROP, except a previous value associated with <ind> is saved.

Consider the following sequence of evaluations:

```
        (PUTPROP 'WALDO 'CHILDREN '(LOUIE SAM)) => (LOUIE SAM)
        (ADDPROP 'WALDO 'CHILDREN '(NERD)) => (NERD)

Now     (GETPROP 'WALDO 'CHILDREN) => (NERD)
        (REMPROP 'WALDO 'CHILDREN) => (NERD)
        (GETPROP 'WALDO 'CHILDREN) => (LOUIE SAM)
```

(PLIST <atom>)   SUBR

PLIST returns a representation of the property-list associated with <atom>.

```
(PLIST 'WALDO) => ((CHILDREN LOUIE SAM))
```

(PUTPROP 'WALDO 'FOO '7) => 7, and now:

```
(PLIST 'WALDO) => ((CHILDREN LOUIE SAM) (FOO . 7)))
```

Functions for Atom Names and Strings

(CHARPOS <chr> <str>) SUBR

CHARPOS will return the position of the first occurrence of <chr> in <str>; if <chr> does not occur NIL is returned.

```
(CHARPOS \c "ABCDEF") => 3
```

(GENSYM) SUBR

Generates a new symbol name of the form Gnnn, where nnn is an integer.

```
(GENSYM) => G100
(GENSYM) => G101
```

(ASCII <arg>) SUBR

If <arg> is an integer, ASCII returns the character whose ascii code is that number. If <arg> is a character, then the ascii code for that character.

```
(ASCII \C) => 67

(ASCII 67) => \C
```

(INSERT <string>) SUBR

Find a literal atom with print name <string> and return that atom as value or, if no such atom exists, construct a new atom with that print name.

(LOOKUP <string>) SUBR

Like INSERT, except returns NIL if the desired atom is not in the symbol table; in this case a new atom is not constructed.

```
Assume   (LOOKUP "ABC") => NIL

then     (INSERT "ABC") => ABC,

and now (LOOKUP "ABC") => ABC
```

(PNAME <atom>) SUBR

Return a string which represents the print name of <atom>.

```
(PNAME 'ABC) => "ABC"
```

(REMOVE <atom>) SUBR

Removes <atom> from the symbol table; return <atom> as value. REMOVE is a dangerous function. For example,

```
        (SETQ Y (REMOVE 'X))  => X
```

removes X, and now type:

```
        (EQ 'X Y) => NIL !
```

This occurs because the act of reading 'X creates a new X which is not EQ to the old X. All input and computation which occurred before the REMOVE will access the old X, but all input after the REMOVE will access the new X; mystery can result!

(STRCOMP <string1> <string2>) SUBR

This function allows lexicographical comparison of the two strings, returning -1, 0, or 1 if <string1> is less than, equal to, or greater than <string2>, respectively.

```
        (STRCOMP "AB" "A") => 1

        (STRCOMP "A" "B") => -1
```

(OBLIST) SUBR

Returns a list of the atoms currently known to LISP.

(GETFN <proc>) SUBR

and

(PUTFN <proc> <sexpr>) SUBR

These functions allow us to manipulate the text of a defined function. GETFN extracts a list-respresenting the body of the function <proc> if <proc> is a user-defined function. PUTFN is used to re-install <sexpr> as a function definition of <proc>. These functions are most useful in writing system functions like editors and debuggers that must modify the representation of functions.

```
(DE FOO (X Y) (CONS X Y)) =>,  then

(GETFN FOO) => ((X Y) (CONS X Y)).

Note that     (TYPE FOO) => EXPR,

but    (TYPE (GETFN FOO)) => LIST
```

Arithmetic Functions

TLC-LISP supports both small integer and floating point arithmetic. We use <n>, <fix>, and <flt> to stand for numbers, fixed-point numbers, and floating-point numbers, respectively.

The examples in this section will assume decimal input and output; for a complete description of numbers and their representation, see the discussion of "Conventions" at the beginning of this section.

(ADD1 <n>) SUBR
returns <n>+1.

```
(ADD1 4) => 5

(ADD1 -1) => 0
```

(SUB1 <n>) SUBR
returns N-1.

```
(SUB1 4) => 3

(SUB1 0) => -1
```

(ABS <n>) SUBR
returns the absolute value of <n>; this function works for any type of number.

```
(ABS -1) => 1

(ABS 3.4) => 3.4
```

The following four arithmetic functions --ADD, SUB, MUL, and DIV--are all LSUBRS in their most general setting; they all use the convention that if any argument is a floating point number, then the result will be floating point. Variants of these four operations which are restricted to specific types of numeric arguments are only available in binary form.

We use MacLISP-like conventions for the arithmetic functions: using, ADD for addditions which may involve both fixed and floating point numbers; + and +$ for additions which are restricted to fixed and floating point numbers, respectively.

(ADD {<n>})
(+ <fix1> <fix2>)
(+$ <flt1><flt2>)

Return the sum of the arguments.

```
(+ 3 4) => 7

(ADD 1.2 4 4) => 9.2

(ADD) => 0
```

(SUB {<n>})
(-<fix1> <fix2>)
(-$ <flt1> <flt2>)

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

(- 1 2) => -1

(SUB 1 2 3) => -4
```

(MUL {<n>})
(* <fix1> <fix2>)
(*$ <flt1> <flt2>)

Returns the product of the arguments.

```
(MUL 2.0 3 4) => 24.0

(* 2 (ADD1 5)) => 12
```

(DIV {<n>})
(/ <fix1> <fix2>)
(/$ <flt1> <flt2>)

DIV returns its first argument divided by the rest of its arguments. If only one argument is given, the reciprocal is returned.

```
(DIV 4.0 2) => 2.0

(/ 4 2) => 2

(DIV 5.0) => 0.2
```

(REM <fix1> <fix2>) SUBR

Form the remainder upon division of <fix1> by <fix2>; the sign of the result is the sign of the dividend.

```
(REM -5 2) => -1
```

Two arithmetic conversion functions are provided:

(FIX <flt>) SUBR
(FLOAT <fix>) 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 <n>) SUBR
returns NIL if <n> is non-zero; returns <n> otherwise.

(GE <n1> <n2>) SUBR
returns NIL if <n1> is less than<n2>; returns <n1> otherwise.

(GT <n1> <n2>)

(LE <n1> <n2>)

(LT <n1> <n2>)
These are similar to GE.

(MINUSP <n>) SUBR
returns <n> if <n> is a negative number; returns NIL otherwise.

Logical Functions

The functions in this section perform bit-wise logical operations. They are restricted to integer parameters.

Note: the prefix # indicates that the number following it is base eight in these examples. For a complete discussion of the effect of # see the discussion of "Conventions" at the beginning of this section.

(LOGAND <fix1> <fix2>) SUBR
Perform the logical 'and' between <fix1> and <fix2>

For example: (LOGAND #27 #14) => #[8]4

(LOGOR <fix1> <fix2>) SUBR
Perform the inclusive or between <fix1> and <fix2>

For example: (LOGOR #27 #14) => #[8]37

(LOGXOR <fix1> <fix2>) SUBR
LOGXOR gives the exclusive or between <fix1> and <fix2>.

For example: (LOGXOR #27 #14) => #[8]33

(COMPL <fix>) SUBR
Form the complement of <fix>.
Equivalent to (LOGXOR <fix> #[8]37777).

General Error Functions

TLC-LISP supplies a collection of functions to examine the state of the LISP machine in case of an error. These functions may be used to create a sophisticated debugging system.

(ARGSFRAME &OPTIONAL (<fix> 0))

This function returns a list of the arguments passed to the <fix>-th pending function invocation.

(FCNFRAME &OPTIONAL (<fix> 0))

This function returns the function applied in the <fix>-th previous pending function invocation.

(RETFRAME <sexpr> &OPTIONAL (<fix> 0))

RETFRAME returns from the <fix>-th pending invocation, using <sexpr> as the returned value.

(TRACEFRAME &OPTIONAL (<fix> 0))

This function prints a "backtrace" of the pendent function invocations, beginning at the <fix>-th frame.

Finally, we supply a mechanism for signalling errors:

(ERROR {<sexpr>}) LSUBR

ERROR prints the list of arguments and returns to the toplevel of LISP. As with TOPLEV, the system supplied ERROR function can be replaced by the user. Simply re-define ERROR.

Besides calling ERROR explicitly, one may also invoke the error handler by typing <control>-G. When this key combination is pressed, the evaluator is interrupted and a "HALT" error is generated.

Input and Output

Though LISP was created in the era of batch-processing, LISP is a distinctly interactive language. Its programming style --exploratory and incremental--thrives on a calculator-like immediacy. An expanding part of LISP's interactive nature is its input and output. Some of the most successful LISP implementations have been on computing systems with sophisticated display systems. For example, one common complaint about LISP is the beginner's difficulty with parentheses: LISP --Lots of Irritating Single Parentheses. One common trick is to invoke a pretty printer to format the output such that the substructure of expressions is apparent from its positioning on the page (we have used pretty printing throughout the manual.) Incremental pretty printing of input is also most helpful. Of course, such techniques are not restricted to display systems; hard-copy devices can also use these ideas. However if we embed a LISP editor in a window-oriented editor, a whole new class of techniques becomes available. We could locate a matching parenthesis by pointing a cursor at one parenthesis and blinking its mate; we could edit LISP objects by manipulating atoms and lists on the screen as entities, rather than as simple character strings. We could finally begin to look upon program preparation as something more than the application of an ersatz keypunch.

In a somewhat more mundane note, the Achilles heel of every programming language is its input and output; LISP is no exception. In line with our goal to present a streamlined and strengthened LISP dialect for the 1980's, we have begun to unify the ideas of "sinks and sources" for output and input. One principle of a well-designed system is that anything that can be done from a terminal can be done within a program, and conversely. For example, a simple text editor can become quite powerful by including a macro facility which defines complex operations in terms of sequences of program-generated "keystrokes". The key to this behavior is the ability to redirect the input program to arbitrary (but compatible) sources.

To this end we allow the TLC-LISP readers and printers to specify a "file data type" which may either be a traditional input/output device, or may be a list of strings. The combination of strings as sinks and sources, and the string manipulation functions supplies the TLC-LISP programmer with elegant power. Elegance, in that string objects need not be represented as lists of atoms; power, since string items can be components of list manipulation.

All input and output functions have a &OPTIONAL parameter that specifies which sink or source is to be used in the operations. That parameter defaults to 'the last one'. Since it is also useful to know the name of 'the last one', the atoms CURRENT-SOURCE and CURRENT-SINK are bound to the currently selected input source and output sink, respectively.

An input operation must also handle the problem of "echoing" --whether to print the input stream on an output sink. The most common case involves interactive input, but one might also wish to echo input from prepared files. Of course, echoing may be desired on the console or on a disc file, or both; and, of course, many times echoing is not wanted. For example, the printing of passwords, or the reader's progress through very large files is seldom desirable. These varying demands are catered to by appropriate use of sinks and sources. Input from files is not gratuitously echoed; if echoing is desired, then one must specify the read-print behavior in a small LISP program. The solution to echoing the interactive source is given by splitting that source into two sources --one which echoes, named CONSOLE; and one which does not echo, named KEYBOARD. The default TOPLEV uses CONSOLE as its source; if different behavior is desired, TOPLEV may be redefined.

In conjunction with the input and output functions, we need to specify control information. A reader must be able to examine the state of the input stream with or without modifying it; a printer should be able to specify formatting information. Both of these expediencies are catered for in an additional optional argument. Future releases of TLC-LISP will exploit the generality further.

Input

The LISP reader recognizes various special characters. Later we will describe how to modify and extend these facilities, but now will discuss the default settings of standard TLC-LISP.

These characters include:

'	the QUOTE character
.	the dot character
"	the string character
;	the comment character
\	the char character
#	the number-base prefix character

A detailed description follows:

The QUOTE character: ' Instead of requiring the user to type (QUOTE Exp), TLC-LISP supports the abbreviation 'Exp. Thus:

```
'(A B) is the same as (QUOTE (A B))
''(A B) is the same as  (QUOTE (QUOTE (A B))).
```

The dot character: . This character is used in the representation of dotted pairs; thus: (A . B). This is not the same as the decimal point in decimal or floating point representation.

The string character: " String literals are presented to TLC-LISP as arbitrary character sequences of length less than 256, bracketed within a pair of ". Thus "ABCD" is a string as is "(foO". To include the character " in a string use a double ""; thus the string "a single "" mark" contains a single ".

The comment character: ; Comments are encouraged. The default comment character is ";". 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 the string "A" is not the same object as the character \A just as the list (A) is not the same object as the atom A.

The number-base character: # This character is used to prefix a number which is to be interpreted using INPUT-BASE as its base. Thus #[8]10 = 8

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.

The format of a read macro definition is:

(DMC <chr> <list> {<exp>}) FSUBR
<chr> 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 <chr>; 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; second, the entry in the character table for <chr> is modified to reflect its new position as a macro; this is done by TYPECH. The function TYPECH is used to examine and modify the table of character properties.

(TYPECH <chr> &OPTIONAL <n>) SUBR
If <n> is missing, TYPECH gives the current character-table value for <chr>. If <n> is given, TYPECH sets the character-table value for <chr> to <n>.

Acceptable values for <n> are the following:

0: totally ignore the character

1: the character is to function like the dot in "dot-notation".

2: the character begins a comment; ignore all input until a comment-end character is seen. (For example, ;)

3: the character ends a comment. (e.g. ; and <cr>)

4: the character is a separator (e.g. space and tab)

5: the character is a read-macro; its value cell contains the definition to be applied when this character is seen in the input stream.

6: the character is a string delimiter (e.g. ")

7: the character designates hex input (e.g. #)

8: these are normal characters

9: these are character-characters. e.g. \

10: these are left parenthesis characters

11: these are right parenthesis characters

12: these are backspace characters

Between read macros and TYPECH, the user can redefine the syntax accepted by the scanner at a very low level. A version of the scanner is also available to the user.

(SCAN &OPTIONAL <source>) SUBR

SCAN will return either a basic token --string, character, identifier, or number--or will return a single character representation of a delimiter. One can then use SCAN as a component of a parser; see the Examples section for several applications of SCAN.

<source>, if present, is control block for a list of strings or a disk file (this control block is set up by a call on OPEN); in either case, input is accepted from that source. For examples, see the file EXAMPLE.IO.

The default parser, supplied in TLC-LISP is named READ:

(READ &OPTIONAL <source> <read-info>) SUBR

This function is the main LISP parsing routine. It reads the next well-formed expression from the current input source, and returns that expression as value after establishing its internal form.

<source>, if present, is control block for a list of strings or a disk file (this control block is set up by a call on OPEN); in either case, input is accepted from that source.

The quantity of input accepted by READ is defined by a "status word" defined by <read-info>.

Currently this value is a three bit quantity defined in the following table where the "starred" values are the defaults.

bit position	value	meaning
0	1	Read the next character from the source.
	0*	Read the next object from the source.
1	1	Examine the next token in the input without moving the input pointer.
	0*	Accept the next input token.

This implementation does not support a <read-info> value of 2. The value of <read-info> local to the <source>, therefore subsequent READs on a <source> will use the previous value until it is superceeded. The value may be replaced either by calling READ with a new <read-info> word or by using TYPEREAD.

(TYPEREAD <source> &OPTIONAL <fix>) SUBR

If the optional <fix> argument is present, then it replaces the current <read-info> associated with the <source>; if only one argument is given, the current value of the <source>s <read-info> is returned. For example, one may examine the current setting of the <read-info> word by: (TYPEREAD CURRENT-SOURCE).

This function is useful for defining a "peek" function, for example:

```
        (DE PEEK (&OPTIONAL (SRC CURRENT-SOURCE)
                  &AUX (TEM (TYPEREAD SRC)))
                 (PROG1 (READ SRC #[2]11) ;peek a character
                        (TYPEREAD SRC TEM))) ; restore read-info, ar
```

Appropriate combinations of <source> and <read-info> cover a multitude of input functions usually supplied in LISP implementations. However when expecting input from the terminal, it is frequently desirable to discover whether a key has been struck without accepting the input or, if no key has been struck, allow the program to continue until input appears. The function TYS serves this purpose.

<u>(TYS)</u> 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 useful in interactive programs that can be interrupted by striking a key.

Output

As with READ, the output functions are controlled by a status word; here it is named <print-info>. The values for <print-info> are given below, again with the default values starred.

bit position	value	meaning
0	1	don't print a space after the output.
	0*	print a trailing space.
1	1	Print strings and characters without surrounding string delimiters.
	0*	Print strings and characters with surrounding string delimiters.

PRINT, like READ, has a primitive to mainpulate the status word; in this case, it is called TYPEPRINT, however its action is analogous to TYPEREAD.

As with input, TLC-LISP accomodates the traditional class of LISP output functions as variations on a simple theme. The kernel function is:

(PRINO <exp> &OPTIONAL <sink> <print-info>) SUBR

Print the value of the <exp> to the sink referenced by <sink>.

(TERPRI &OPTIONAL <sink>) SUBR

Print a carriage return-line feed sequence on the current output device.
An abbreviation for (PRINO "
" <sink> #[8]3)

(PRINT <exp> &OPTIONAL <sink> <print-info>) SUBR

This function has the effect of:

```
(PROG1 (PRINO <exp> <sink>) (TERPRI <sink>))
```

One may also control the field width in which information is printed; this is accomplished by two integer-valued variables, LEFT-MARGIN and RIGHT-MARGIN. The output is printed from LEFT-MARGIN through RIGHT-MARGIN. The initial settings are LEFT-MARGIN at 1 and RIGHT-MARGIN at 80.

(CHARCT &OPTIONAL <sink>) SUBR

This function returns the number of character positions left in the current output line of <sink>.

File specification

A file name in TLC-LISP is a string specified in the following format:

```
   "d:xxxxxxxx.yyy", where:
        d is the device, (A, B, C, D)
        xxxxxxxx.yyy is the name and extension.
```

The three disk related functions in TLC-LISP are:

(OPEN <string> <status>) SUBR

<string> designates a file name as described above. The <status> is either OLD or NEW. The value returned is a file data type suitable as an argument to READ or PRINO.

The "<string>" may also be a list, in which case it designates a stream, described as a list of strings. If the argument is NIL, an empty list of strings is built; in either case the list of strings is prepared for input and output. In case the stream is exhausted on input, an optional character-valued function may be applied to realize more input. The system default function simply supplies an end-of-file character to the reader.

(CLOSE <file> <status>) SUBR

<file> is closed and if <status> is PURGE the file is deleted; otherwise it is saved.

(RENAME <string1> <string2>) SUBR

The file <string2> is renamed to <string1>.

Disk Utility Functions

Two functions are supplied that load text into TLC-LISP.

(TYPEFILE <string>)   SUBR

and

(LOAD <string>)   SUBR

In both cases, <string> is a file name. TYPEFILE executes a READ-LOOP on the file; LOAD executes a READ-EVAL-PRINT loop.

Sink and Source Controls

Given the ability to open several (kinds of) sinks and sources, we also need to select these objects as input and output targets. This is handled by the atoms CURRENT-SOURCE and CURRENT-SINK. These atoms are initially bound the the console FCB, but may be rebound to select alternate input and output. The value associated with these variables should be an object created by OPEN.

One also has access to the OPEN-created objects through the variable FILE-LIST which contains an entry for each open file; this list is automatically maintained by OPEN and CLOSE.

(GC) SUBR

This function makes an explicit call on the garbage collector.

(EXIT &OPTIONAL <string$>) SUBR

This function returns control to CPM, such that the resulting memory image can be SAVEd on a disk file for later restart. If String$ is present this message will be displayed when the SAVEd file is run. Note: String$ must be terminated with '$', for example (EXIT "--Welcome to the LISP data base --$").

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. One way to forestall this difficulty is to "virtualize" large programs that may only be needed for short durations. Of course, one could explicitly expunge functions, thereby reclaiming their space. Rather than resort to this rather ugly solution, TLC-LISP recognizes an "auto-load" value in the VALUE cell of a symbol. When an attempt is made to fetch an autoload value, the TLC-LISP interpreter retrieves the actual value from the appropriate disk. The information available to the interpreter is the file name, record, and relative byte in the record that indicates the beginning of the LISP object; since the disk operation occurs as a random access, it is reasonably rapid.

Two types of autoload are available: "smash" and "no-smash". A "smash" object is loaded in and replaces the contents of the value cell; subsequent references to that symbol will retrieve the value without accessing the disk. The system also saves the autoload information so that the value may be "unsmashed" and the space reclaimed. This is done by a call

(UNSMASH symbol).

UNSMASH is defined on the file AUTO.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.

An autoload file consists of two parts: a directory file ?.ato which contains calls to (the SUBR) AUTO, like:

(AUTO <smash indicator> <name> <file name> <rec> <pos>)

where the indicator is SMASH or NO-SMASH <name> is the symbol that will have the autoload object, and the last three arguments contain the file information as described above. The user calls LOAD with the ?.ato files that are to be needed in the application.

The second part of an autoload file system is, of course, the file --<file name>--that contains the LISP text. This file is never explicitly loaded; it is accessed through the autoload mechanism. Since the determination of <rec> and <pos> is non-trivial, TLC-LISP includes utilities to make a pair of autoload-able files. These utilities are included on the file AUTO.LSP. The major component is a function named ALOAD, which takes a file name as argument and converts it to an autoload-able file. A more complete description of these function is included on the AUTO.LSP file.

The EVAL-APPLY pair

The interpreter given below is meant only to be indicative of the behavior of TLC-LISP, not to be definitive.

```
(DE EVAL (X) (COND ((SYMBOLP X) (COND ((GETVAL X))
                                      (T (ERROR "unbound atom")
                   ((LISTP X) (SELECTQ (TYPE (EVAL (FCN X)))
                                       (SUBR (DOIT (FCN X)
                                                   (EVLIST (ARGS X))))

                                       (FSUBR (DOIT (FCN X)
                                                    (LIST (ARGS X))))

                                       (EXPR (APPLY (EVAL (FCN X))
                                                    (EVLIST (ARGS X))))

                                       (FEXPR (APPLY (EVAL (FCN X))
                                                     (LIST (ARGS X))))

                                       (MACRO (EVAL (APPLY (EVAL (FCN X))
                                                           (LIST X))))

                                       (OW (ERROR "undefined function")))
                   (T X)))

;if the expression is a symbol get its value; in the symbol is
;       unbound we call ERROR.

;if the expression is a list, it represents a function application,
;       a special form, or a macro call; act accordingly.
;       Note that the function position is always evaluated, and
;       must be a functional object.

;otherwise, return the object; numbers, strings, etc. fit here.


(DE APPLY (FN L &AUX VAL)   (BIND (FORMALS FN) L)
                            (SETQ VAL (EVAL (BODY FN)))
                            (UNBIND (FORMALS FN))
                            VAL)

;This APPLY has an easy job; it does not handle  &OPTIONAL, &REST, or
;       &AUX. Since this is a shallow binding interpreter, BIND
;       saves the old values of the formals, moves the new values
;       into the value-cells, and evaluated the body of the functional
;       object in this new environment. The value is saved as the
;       old values of the formals are restored.

;We can give suggestive definitions for some of the subfunctions
;       too:
```

```
(DE EVLIST (L) (MAPLIST (LAMBDA (X) (EVAL (FIRST L)))
                        L))

;i.e. generate a list of evaluated arguments.


(DE BIND (FORMALS VALS)
               (IF (NULL FORMALS)
                   NIL
                   (SAVE (FIRST FORMALS))
                   (PUTVAL (FIRST FORMALS)
                           (FIRST VALS))
                   (SELF (REST FORMALS)
                         (REST VALS))))

;Of course, BIND should  make sure that the number of (required)
;      formals is equal to the number of supplied values.
;SAVE will store the contents of a symbol's value-cell.
;PUTVAL will smash a  value into the value-cell.


(DE UNBIND (FORMALS)
               (IF (NULL FORMALS)
                   NIL
                   (PUTVAL (FIRST FORMALS)
                           (RESTORE (FIRST FORMALS)))
                   (SELF (REST FORMALS))))

;RESTORE locates the symbol's saved value.
;UNBIND simply undos BIND's work.
```

This definition, though not complete, gives a concise description of the action of EVAL and APPLY.

This appendix gives some examples of LISP programming extracted from the "Artificial Intelligence Programming" book. TLC-LISP and the LISP discussed in that book differ in both inessential and essential ways. The inessential differences involve the LISP library: one LISP will have some functions that the other lacks. These inessential differences can be easily remedied by defining the missing functions, by redefining existing library functions to your liking, or by incorporating the changes into your programming. For example, TLC-LISP uses GETPROP rather than GET to name the function that extracts a value from a property list; the name GETPROP fits better with the other property-list function (ADDPROP, REMPROP, and PUTPROP) than "GET". Similarly, in TLC-LISP these functions always have <name> and <property> as their first two arguments.

The essential differences require more care and are outlined below.

First, TLC-LISP does not have the "PROG"-feature. PROG tends to be used for (1) initialization of local variables, or (2) programming iteration. Instead, use the "&AUX"-facility for initialization, and a form of "DO" to express iteration. If PROG really is desired, it can be expressed as an appropriate collection of "CATCH/THROW" expressions.

So for example, figure 1.1 of "AIP" can be expressed as

```
(DE ADD (N) (DO ((N N (SUB1 N)) ;initialize a local N to the actual paramete
                 (SUM 0         ; decrement that value each time aound the
                      (PLUS SUM N)))   ; replace SUM by (PLUS SUM N) on each
                (((EQUAL N 0) SUM)) ))   ; exit the DO with SUM when N=0.
```

Note that the comment conventions are different between the two LISPs. Be warned that the "super-bracket", /, is not implemented in TLC-LISP; parenthesis balancing are better accomplished by an understanding LISP editor.

Also, TLC-LISP does not have LEXPRs. These constructs are basically ugly; use &REST instead. FEXPRs are supported in both LISPs, but are seldom really necessary; usually macros supply what is desired.

By the time you have reached Chapter five of the AIP book -- "Flow of Control"--, you should have sufficient familarity with TLC-LISP and macros that this chapter can be digested easily.

Chapter six, "I/O in LISP", begins with a discussion of character strings. Since TLC-LISP supplies first-class string objects, much of this discussion is out-dated. As with most languages, the subject of input and output is very implementation dependent. TLC-LISP supplies a comprehensive input/ output package; we recommend that you understand and use these facilities rather than map TLC-LISP into Chapter six's facilities.

Chapter seven, "Editing LISP Expressions", is an interesting example of how one can use LISP as a systems implementation language. We encourage you to implement this chapter and compare your TLC-LISP code with that in AIP.

Examples

The remainder of this appendix discusses some generally useful techniques for adding structure to your LISP programming style. These examples come from the AIP book, and are included on your disk as "EXAMPLE.AIP".

First, are the basic read-macro facilities of section 3.3. ; note we used $ rather than " for quasi-quote. " can be used if desired.

```
(dmc \! (&aux (char(readch)))
  (cond ((eq char \@)  (list '*splice-unquote* (read)))
        ((eq char \$) (quasi-quote (read)))
        (t (list 'error char))))
;NOTE the super-bracket hack, ], does not exist. From the console,
; (T . (L . C)) LISP will react when parentheses "count-out. From a file
; one can simply build a right-parentheses "fence": ))))))).
; a good display editor will handle the parenthesis-balancing problem,
; and [-] are reserved for the array extension of (T .(L . C)) LISP.

(dmc \@ () (list '*unquote* (read)))


(de readch ( &OPTIONAL (src current-source)
             &AUX (tem (typeread src)))
        (prog1 (read src 1) (typeread src tem)))
;NOTE we define readch as an instance of read

(de quasi-quote (skel)
  (cond ((null skel) nil)
        ((atom skel) (list 'quote skel))
        ((eq (car skel) '*unquote*) (cadr skel))
        ((and (listp (car skel))(eq (caar skel) '*splice-unquote*))
                (list 'append (cadar skel)(quasi-quote (cdr skel))))
        (t (combine-skels (quasi-quote (car skel))
                        (quasi-quote (cdr skel))
                        skel))))


(de combine-skels (lft rgt skel)
  (cond ((and (isconst lft)(isconst rgt))(list 'quote skel))
        ((null rgt)(list 'list lft))
        ((and (listp rgt)(eq (car rgt) 'list))
                (cons 'list (cons lft (cdr rgt))))
        (t (list 'cons lft rgt))))


(de isconst (x) (or (null x) (eq x t)(numberp x)
                    (and (listp x) (eq (car x) 'quote))))


;--------------------------
; Here's the record package of section 4.3

(dm record-type (l) (let ((*type* (cadr l))
                          (slots (caddr l)))
        (list 'de *type* (slot-funs-extract slots nil)
                         (struc-cons-form slots))))
;NOTE the let syntax is slightly different

(de slot-funs-extract (slots path)
  (cond ((null slots) nil)
        ((atom slots) (eval  ¦$(dm @(insert (string (pname slots)
                                                     \:
                                                     (pname *type*)))
                                 (l)
                                 (list  ' @(insert (apply string
                                                          ¦$(\c ¦@path \r))
                                        (cadr l))))
                (list slots))
        (t (nconc (slot-funs-extract (car slots)(cons \a path))
                (slot-funs-extract (cdr slots)(cons \d path))))))

;NOTE: several differences occur in slot-funs-extract. (T . (L . C)) LISP
; has better string facilities, so this function is simpler, and takes
; fewer cons-es.
; Note also the use of apply. This occurs because string takes an arbitrary
; number of arguments.


(de struc-cons-form (struc)
  (cond ((null struc) nil)
        ((atom struc) struc)
        (t (list
            'cons
            (struc-cons-form (car struc))
            (struc-cons-form (cdr struc))))))

; an example

(record-type goalnode (char state . plans))

(setq xx (goalnode 'macbeth '(eq macbeth king) '((murder )(treason))))

(char:goalnode xx)
```

```
(state:goalnode xx)

(plans:goalnode xx)


;-----------------------------------
; The := macro hack to allow gneralized assignments.

(DM := (EXPRESSION)
        (LET ((LFT (CADR EXPRESSION))
              (RGT (CADDR EXPRESSION)))
             (COND ((ATOM LFT) {$(SETQ @LFT @RGT))
                   ((GETPROP (CAR LFT) 'SET-PROGRAM)
                        (CONS (GETPROP(CAR LFT) 'SET-PROGRAM)
                              (APPEND (CDR LFT) (LIST RGT))))
                   (T (ERROR)))) )

(PUTPROP 'CAR 'SET-PROGRAM 'RPLACA)

(PUTPROP 'CDR 'SET-PROGRAM 'RPLACD)

(PUTPROP 'GETPROP 'SET-PROGRAM 'PUTPROP)

;NOTE the different order in the arguments to the property-list function,
; PUTPROP.
```

INDEX