# CLISP - Conversational LISP

Warren Teitelman
Xerox Palo Alto Research Center
Palo Alto, California 94304

## Abstract

CLISP is an attempt to make LISP programs easier to read and write by extending the syntax of LISP to include infix operators, IF-THEN statements, FOR-DO-WHILE statements, and similar ALGOL-like constructs, without changing the structure or representation of the language. CLISP is implemented through LISP's error handling machinery, rather than by modifying the interpreter: when an expression is encountered whose evaluation causes an error, the expression is scanned for possible CLISP constructs, which are then converted to the equivalent LISP expressions. Thus, users can freely intermix LISP and CLISP without having to distinguish which is which. Emphasis in the design and development of CLISP has been on the system aspects of such a facility, with the goal in mind of producing a useful tool, not just another language. To this end, CLISP includes interactive error correction and many 'Do-What-I-Mean' features.

* * *

The syntax of the programming language LISP[2,3,9] is very simple, in the sense that it can be described concisely, but not in the sense that LISP programs are easy to read or write! This simplicity of syntax is achieved by, and at the expense of, extensive use of explicit structuring, namely grouping through parenthesesization. For the benefit of readers unfamiliar with LISP syntax, the basic element of LISP programs is called a form. A form is either (1) atomic, or (2) a list, the latter being denoted by enclosing the elements of the list in matching parentheses. In the first case, the form is either a variable or a constant, and its value is computed accordingly. In the second case, the first element of the list is the name of a function, and the remaining elements are again forms whose values will be the arguments to that function. In Backus notation:

Figure 1
Syntax of a LISP Form

```
<form> ::= <variable>|<constant>|
    (<function-name> <form> ... <form>)
```

For example, assign X the value of the sum of A and the product of B and C is written in LISP as (SETQ X (PLUS A (TIMES B C))).

The syntax for a conditional expression is correspondingly simple:*

Figure 2
Syntax of a Conditional Expression

```
<conditional expression> ::=
      (COND <clause> ... <clause>)
<clause> ::= (<form> <form>)
```

Note that there are no IF's, THEN's, BEGIN's, END's, or semi-colons. LISP avoids the problem of parsing the conditional expression, i.e., delimiting the individual clauses, and delimiting the predicates and consequents within the clauses, by requiring that each clause be a separate element in the conditional expression, namely a sublist, of which the predicate is always the first element, and the consequent the second element. As an example, let us consider the following conditional expression which embodies a recursive definition for FACTORIAL:**

Figure 3
Recursive Definition of Factorial

```
(COND
  ((EQ N 0)
    1)
  (T (TIMES N (FACTORIAL (SUB1 N)))))
```

* Actually, a clause can have more than two consequents, in which case each form is evaluated and the value of the last form returned as the value of the conditional. However, for the purposes of this discussion, we can confine ourselves to the case where a clause has only one consequent.

** The expression in Figure 3 is shown as it would be printed by a special formatting program called PRETTYPRINT. PRETTYPRINT attempts to make the structure of LISP expressions more manageable by judicious use of indentation and breaking the output into separate lines. Its existence is a tacit acknowledgment of the fact that LISP programs require more information than that contained solely in the parenthesization in order to make them easily readable by people.

The first clause in this conditional is the list of two elements ((EQ N 0) 1), which says if (EQ N 0) is true, i.e., if N is equal to 0, then return 1 as the value of the conditional expression, otherwise go on to the second clause. The second clause is (T (TIMES N (FACTORIAL (SUB1 N)))), which says if T is true (a tautology-the ELSE of ALGOL conditionals), return the product of N and (FACTORIAL (SUB1 N)). The latter is evaluated by first evaluating (SUB1 N), and then calling FACTORIAL (recursively) on this value.

As a result of the structuring of conditional expressions, LISP does not have to search for words such as IF, THEN, ELSE, ELSEIF, etc., when interpreting or compiling conditional expressions in order to delimit clauses and their constituents: this grouping is specified by the parentheses, and is performed at input time by the READ program which creates the list structure used to represent the expression. Similarly, LISP does not have to worry about how to parse expressions such as A+B*C, since (A+B)*C must be written unambiguously as (TIMES (PLUS A B) C), and A+(B*C) as (PLUS A (TIMES B C)). In fact, there are no reserved words in LISP such as IF, THEN, AND, OR, FOR, DO, BEGIN, END, etc., nor reserved characters like +, -, *, /, =, ←, etc.* This eliminates entirely the need for parsers and precedence rules in the LISP interpreter and compiler, and thereby makes program manipulation of LISP programs straightforward. In other words, a program that "looks at" other LISP programs does not need to incorporate a lot of syntactic information. For example, a LISP interpreter can be written in one or two pages of LISP code.[3] It is for this reason that LISP is by far the most suitable, and frequently used, programming language for writing programs that deal with other programs as data, e.g., programs that analyze, modify, or construct other programs.

However, it is precisely this same simplicity of syntax that makes LISP programs difficult to read and write (especially for beginners). 'Pushing down' is something programs do very well, and people do poorly. As an example, consider the following two "equivalent" sentences:

> "The rat that the cat that the dog that I owned chased caught ate the cheese."

versus

> "I own the dog that chased the cat that caught the rat that ate the cheese."

Natural language contains many linguistic devices such as that illustrated in the second sentence above for minimizing embedding, because embedded sentences are more difficult to grasp and understand than equivalent non-embedded ones (even when the latter sentences are somewhat longer). Similarly, most high level programming languages offer syntactic devices for reducing apparent depth and complexity of a program: the reserved words and infix operators used in ALGOL-like languages simultaneously delimit operands and operations, and also convey meaning to the programmer. They are far more intuitive than parentheses. In fact, since LISP uses parentheses (i.e., lists) for almost all syntactic forms, there is very little information contained in the parentheses for the person reading a LISP program, and so the parentheses tend mostly to be ignored: the meaning of a particular LISP expression for people is found almost entirely in the words, not in the structure. For example, the expression in Figure 4

Figure 4
Careless Definition of Factorial

```
(COND (EQ N 0) 1)
      (T TIMES N FACTORIAL ((SUB1 N)))
```

is recognizable as FACTORIAL even though there are five misplaced or missing parentheses. Grouping words together in parentheses is done more for LISP's benefit, than for the programmer's.

CLISP is designed to make LISP programs easier to read and write by permitting the user to employ various infix operators, IF-THEN-ELSE statements, FOR-DO-WHILE-UNLESS-FROM-TO-etc. expressions, which are automatically converted to equivalent LISP expressions when they are first interpreted. For example, FACTORIAL could be written in CLISP as shown in Figure 5.

Figure 5
CLISP Definition of FACTORIAL

```
(IF N=0 THEN 1 ELSE N*(FACTORIAL N-1))
```

Note that this expression would be represented internally (after it had been interpreted once) as shown in Figure 3, so that programs that might have to analyze or otherwise process this expression could take advantage of the simple syntax.

CLISP also contains facilities for making sense out of expressions such as the careless conditional shown in Figure 4. Furthermore, CLISP will detect those cases which would not generate LISP errors, but are nevertheless obviously not what the programmer intended. For example, the expression (QUOTE <expression> <form>) will not cause a LISP error, but <form> would never be seen by the interpreter. This is clearly a parentheses error. CLISP uses both local and global information to detect, and where possible, repair such errors. However, this paper will concentrate primarily on the syntax extension aspects of CLISP, and leave a discussion of the semantic issues for a later time.

---

* except for parentheses (and period), which are used for indicating structure, and space and end-of-line, which are used for delimiting identifiers.

There have been similar efforts in other LISP systems, most notably the MLISP language at Stanford.[4] CLISP differs from these in that it does not attempt to replace the LISP language so much as to augment it. In fact, one of the principal criteria in the design of CLISP was that users be able to freely intermix LISP and CLISP without having to identify which is which. Users can write programs, or type in expressions for evaluation, in LISP, CLISP, or a mixture of both. In this way, users do not have to learn a whole new language and syntax in order to be able to use selected facilities of CLISP when and where they find them useful.

CLISP is implemented via the error correction machinery in INTERLISP*. Thus, any expression that is well-formed from LISP's standpoint will never be seen by CLISP (e.g., if the user defined a function IF, he would effectively turn off that part of CLISP). This means that interpreted programs that do not use CLISP constructs do not pay for its availability by slower execution time. In fact, the interpreter does not 'know' about CLISP at all. It operates as before, and when an erroneous form is encountered, the interpreter calls an error routine which in turn invokes the Do-What-I-Mean (DWIM) analyzer[5,7,8] which contains CLISP. If the expression in question turns out to be a CLISP construct, the equivalent LISP form is returned to the interpreter. In addition, the original CLISP expression, is modified so that it <u>becomes</u> the correctly translated LISP form. In this way, the analysis and translation are done only once.

Integrating CLISP into the LISP system (instead of, for example, implementing it as a separate preprocessor) makes possible Do-What-I-Mean features for CLISP constructs as well as for pure LISP expressions.[5,7] For example, if the user has defined a function named GET-PARENT, CLISP would know not to attempt to interpret the form (GET-PARENT) as an arithmetic infix operation. (Actually, CLISP would never get to see this form, since it does not contain any errors.) If the user mistakenly writes (GET-PRAENT), CLISP would know he meant (GET-PARENT), and not (DIFFERENCE GET PRAENT), by using the information that PRAENT is not the name of a variable, and that GET-PARENT is the name of a user function whose spelling is "very close" to that of GET-PRAENT. Similarly, by using information about the program's environment not readily available to a preprocessor, CLISP can successfully resolve the following sorts of ambiguities:

1) (LIST X*FACT N), where FACT is the name of a variable, means (LIST (X*FACT) N).

2) (LIST X*FACT N), where FACT is not the name of a variable but instead is the name of a function, means (LIST X*(FACT N)), i.e., N is FACT's argument.

3) (LIST X*FACT(N)), FACT the name of a function (and not the name of a variable), means (LIST X*(FACT N)).

4) cases (1), (2) and (3) with FACT misspelled!

The first expression is correct both from the standpoint of CLISP syntax and semantics and the change would be made without the user being notified. In the other cases, the user would be informed or consulted about what was taking place. For example, to take an extreme case, suppose the expression (LIST X*FCCT N) were encountered, where there was both a function named FACT <u>and</u> a variable named FCT. The user would first be asked if FCCT were a misspelling of FCT. If he said YES, the expression would be interpreted as (LIST (X*FCT) N).* If he said NO, the user would be asked if FCCT were a misspelling of FACT, i.e., if he intended X*FCCT N to mean X*(FACT N). If he said YES to this question, the indicated transformation would be performed. If he said NO, the system would then ask if X*FCCT was to be treated as CLISP, since FCCT is not the name of a (bound) variable.** If he

---

* INTERLISP (formerly BBN-LISP[6]) is implemented under the BBN TENEX timesharing system[2] and is jointly maintained and developed by Xerox Palo Alto Research Center and Bolt, Beranek, and Newman, Inc., Cambridge, Mass. It is currently being used at various sites on the ARPA Network, including PARC, BBN, ISI, SRI-AI, etc.

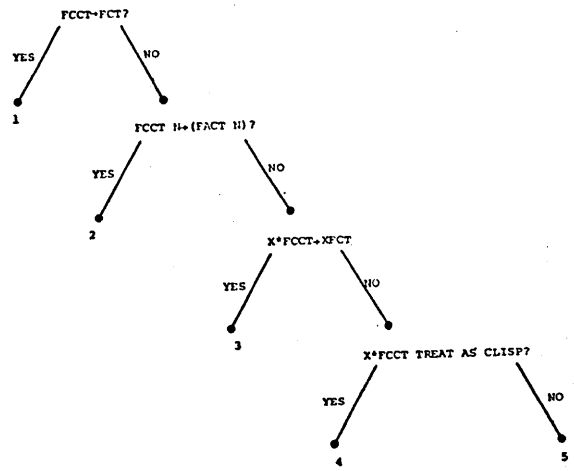
* Through this discussion, we speak of CLISP or DWIM asking the user. Actually, if the expression in question was typed in by the user for immediate execution, the user is simply informed of the transformation, on the grounds that the user would prefer an occasional misinterpretation rather than being continuously bothered, especially since he can always retype what he intended if a mistake occurs, and ask the programmer's assistant to UNDO the effects of the mistaken operations if necessary.[5] For transformations on expressions in his programs, the user can inform CLISP whether he wishes to operate in CAUTIOUS or TRUSTING mode. In the former case (most typical) the user will be asked to approve transformations, in the latter, CLISP will operate as it does on type-in, i.e., perform the transformation after informing the user.

** This question is important because many of our LISP users already have programs that employ variables whose names contain CLISP operators. Thus, if CLISP encounters the expression A/B in a context where either A or B are not the names of variables, it will ask the user if A/B is intended to be CLISP, in case the user really does have a free variable named A/B, but has mistakenly used A/B here in a context where it was not bound.

said YES, the expression would be transformed, if NO, it would be left alone, i.e., as (LIST X*FCCT N). Note that we have not even considered the case where X*FCCT is itself a misspelling of a variable name, as with GET-PRAENT. This sort of transformation would be considered after the user said NO to X*FCCT N -> X*(FACT N). The complete graph of the possible interpretations for (LIST X*FCCT N) where FCT and XFCT are the names of variables, and FACT is the name of a function, is shown in Figure 6.

Figure 6

Possible interpretations of (LIST X*FCCT N)




The final states for the various terminal nodes shown in Figure 6 are:

```
1:   (LIST (TIMES X FCT) N)
2:   (LIST (TIMES X (FACT N)))
3:   (LIST XFCT N)
4:   (LIST (TIMES X FCCT) N)
5:   (LIST X*FCCT N)
```

CLISP can also handle parentheses errors caused by typing 8 or 9 for '(' or ')'. (On most terminals, 8 and 9 are the lower case characters for '(' and ')', i.e., '(' and '8' appear on the same key, as do ')' and '9'.) For example, if the user writes N*8FACTORIAL N-1, the parentheses error can be detected and fixed before the infix operator * is converted to the LISP function TIMES. CLISP is able to distinguish this situation from cases like N*8*X meaning (TIMES N 8 X), or N*8X, where 8X is the name of a variable, again by using information about the programming environment. In fact, by integrating CLISP with DWIM, CLISP has been made sufficiently tolerant of errors that almost everything can be misspelled!* For example, CLISP can successfully translate the definition of FACTORIAL:

```
(IFFN=0 THENN 1 ESLE N*8FACTTORIALNN-1)
```

to the form shown in Figure 3, while making 5 spelling corrections and fixing the parenthesis error.*

This sort of robustness prevails throughout CLISP. For example, the iterative statement permits the user to say things like:

```
(FOR OLD X←M TO N
      DO (PRINT X) WHILE (PRIMEP X))**
```

However, the user can also write OLD (X←M), (OLD X←M), (OLD (X←M)), permute the order of the operators, e.g., DO (PRINT X) TO N FOR OLD X←M WHILE (PRIMEP X), omit either or both sets of parentheses, misspell any or all of the operators FOR, OLD, TO, DO, or WHILE, or leave out the word DO entirely! And, of course, he can also misspell PRINT, PRIMEP, M, or N!***

CLISP is well integrated into the INTERLISP system. For example, the above iterative statement translates into an equivalent LISP form using PROG, COND, GO, etc.**** When the interpreter subsequently encounters this CLISP expression, it automatically obtains and evaluates the translation. Similarly, the compiler "knows" to compile the translated form. However, if the user PRETTYPRINTS his program, at the corresponding point in his function, PRETTYPRINT "knows" to print the original CLISP. Similarly, when the user edits his program, the editor makes the translation invisible to the user. If the user modifies the CLISP, the translation is automatically discarded and recomputed the next time the expression is evaluated.

---

* Where misspelling includes running adjacent words together, as shown in example.

* CLISP also contains a facility for converting from LISP back to CLISP, so that after running the above definition of FACTORIAL, the user could 'CLISPIFY' to obtain:
(IF N=0 THEN 1 ELSE N*(FACTORIAL N-1)).

** This expression should be self explanatory, except possibly for the operator OLD, which says X is to be the variable of iteration, i.e., the one to be stepped from M to N, but X is not to be rebound. Thus when this loop finishes execution, X will be equal to N.

*** In this example, the only thing the user could not misspell is the first X, since it specifies the name of the variable of iteration. The other two instances of X could also be misspelled.

****
```
(PROG NIL
      (SETQ X M)
 $$LP (COND
        ((OR (IGREATERP X N)
             (NOT (PRIMEP X)))
         (RETURN)))
      (PRINT X)
      (GO $$LP)).
```

In short, CLISP is not a language at all, but rather a system. It plays a role analagous to that of the programmer's assistant.[5] Whereas the programmer's assistant is an invisible intermediary agent between the user's console requests and the LISP executive, CLISP sits between the user's programs and the LISP interpreter.

Only a small effort has been devoted to defining a core syntax for CLISP. Instead, most of the effort has been concentrated on providing a facility which 'makes sense' out of the input expressions using context information as well as built-in and acquired information about user and system programs. Just as communication is based on the intention of the speaker to produce an effect in the recipient, CLISP operates under the assumption that what the user said was intended to represent a meaningful operation, and therefore tries very hard to make sense out of it. The motivation behind CLISP is not to provide the user with many different ways of saying the same thing, but to enable him to worry less about the syntactic aspects of his communication with the system. In other words, it gives the user a new degree of freedom by permitting him to concentrate more on the problem at hand, rather than on translation into a formal and unambiguous language.

CLISP has just become operational and the expected reactions and suggestions from users will do much towards polishing and refining it. However, the following anecdote suggests a favorable prognosis: after being cursorily introduced to some of the features of CLISP, two users wanted to try out the iterative statement facility, but neither of them were sure of the exact syntax. The first user thought that if they just typed in something "reasonable", the system would figure out what they meant. And it did!

## References


1 Berkeley, E.C. "LISP, A Simple Introduction," in The Programming Language LISP, its Operation and Applications, Berkeley, E.C. and Bobrow, D.G. (editors), MIT Press, 1966.

2 Bobrow, D.G., Burchfiel, J.D., Murphy, D.L. and Tomlinson, R.S. "TENEX, a Paged Time Sharing System for the PDP-10." Communications of the ACM, March 1972, Vol. 15, No. 3.

3 McCarthy, J. et al. LISP 1.5 Programmer's Manual. MIT Press, 1966.

4 Smith, D.C. MLISP User's Manual, Stanford Artificial Intelligence Project Memo AI-84, January 1969.

5 Teitelman, W. "Automated Programmer - The Programmer's Assistant." Proceedings of Fall Joint Computer Conference, December 1972.

6 Teitelman, W., Bobrow, D.G., Hartley, A.K., Murphy, D.L. BBN-LISP Tenex Reference Manual, Bolt, Beranek, and Newman, Inc., August 1972.

7 Teitelman, W. "Do What I Mean," Computers and Automation, April 1972.

8 Teitelman, W. "Toward a Programming Laboratory," Proceedings of First International Joint Conference on Artificial Intelligence, Walker, D. (editor), May 1969.

9 Weissman, C. LISP 1.5 Primer, Dickenson Press, 1967.