WORKING PAPER 121

CGOL - an Alternative External Representation For LISP users

Vaughan R. Pratt

Massachusetts Institute of Technology
Artificial Intelligence Laboratory

March 1976

Abstract

Advantages of the standard external representation of LISP include its simple definition, its economical implementation and its convenient extensibility. These advantages have been gained by trading off syntactic variety for the rigidity of parenthesized prefix notation. This paper describes an approach to increasing the available notational variety in LISP without compromising the above advantages of the standard notation. A primary advantage of the availability of such variety is the extent to which documentation can be incorporated into the code itself, decreasing the chance of mismatches between code and documentation. The approach differs from that of MLISP[4], which attempts to be a self-contained language rather than a notation available immediately on demand to the ordinary LISP user. A striking feature of a MACLISP implementation of this approach, the CGOL notation, is that any LISP user, at any time, without any prior preparation, and without significant compromise of storage or speed, can in mid-stream change to the CGOL notation merely by typing (CGOL) at the LISP he is presently using, even if he has already loaded and begun running his LISP program. Another striking feature is the possibility of notational transparency; a LISP user may ask LISP to read a file without needing to know the notation(s) used within that file.

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the laboratory's artificial intelligence research is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-75-C-0643.

CGOL - an Alternative External Representation For LISP users

V. R. Pratt

11/25/75

*****WARNING*****
CGOL IS AN EXPERIMENTAL FACILITY. ALTHOUGH IT IS HOPED THAT THE LANGUAGE WILL SETTLE DOWN FAIRLY SOON, IT IS AT PRESENT TOO EARLY TO COMMIT THE DESIGN TO STONE TABLETS. ALSO, SOME DIFFERENCES PRESENTLY EXIST BETWEEN THIS MANUAL AND THE IMPLEMENTATION. THE KNOWN DIFFERENCES ARE DOCUMENTED IN SECTION 9.

*****COME-ON*****
BUT DON'T LET THAT STOP YOU TRYING IT OUT. USER FEEDBACK WILL BE MOST APPRECIATED.

1. PHILOSOPHY

1.1 Representational Nonsense

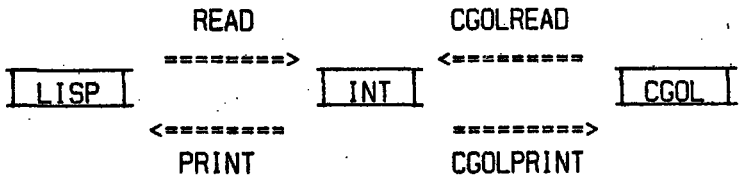
LISP S-expressions ("abstract" objects in a domain that contains atoms and is closed under the pairing function CONS) require an internal and an external representation ("concrete" objects). The former is for the convenience of the programmer, the latter for that of the machine. The functions (READ) and (PRINT) provide maps between the two concrete representations.

We use the term "LISP" principally to denote the abstract objects; out of respect for usage, however, we shall also use it to denote "the" standard external representation, which varies mainly in detail from one implementation to the next. We rely on context to disambiguate these usages. We use "INT" to denote whatever internal representation obtains in a particular implementation.

1.2 CGOL as an alternative external representation.

This document describes an alternative external representation, CGOL, for those LISP S-expressions that encode procedural information. The representation is modeled on McCarthy's M-expression notation; as such it has Smith and Enea's MLISP language as a predecessor. See section 8 for a discussion of similarities and differences between MLISP and CGOL.

In an environment that supports both CGOL and LISP external representation, we envisage facilities for mapping between representations as diagrammed below:



This diagram of course generalizes to any set of external representations.

1.3 Usage

If LISP's top-level, a cut-down version of which might be
(PRINT (EVAL (READ))) ,
is replaced by
(CGOLPRINT (EVAL (CGOLREAD))) the top level should now expect CGOL notation, and reply in like notation. (At present CGOLPRINT is implemented in MACLISP as PRINT, so the user must put up with LISP replies.)

In MACLISP,[2] the function (CGOL) will set up your top and break levels as above. Both (CGOLREAD) and (CGOL) are autoloading. Thus if at any time while talking LISP you suddenly want or need to use CGOL notation, you need merely type (CGOL) at LISP and LISP will then expect CGOL expressions. To revert to standard notation, type LISP♦ . Because of the autoloading feature, no other action on your part is needed. This implies that you may have a mixture of CGOL and LISP programs in a single file, provided the appropriate heading information is given. The overhead of executing (CGOL) and LISP♦ is negligible.

It is hoped at a later date to lambda-bind NOTATION to LISP, CGOL, MAPL or whatever, and to have top-level use NOTATION to choose the appropriate versions of READ and PRINT. This will facilitate proper nesting of environments. An immediate application of such a scheme is to do all file reading with LISP as the default NOTATION for the duration of that file. Thus a user using notation X can read a file without asking what notation it uses, and on exit from the file find that he is still using notation X. People preparing files need merely prefix non-LISP notation with the appropriate instruction to rebind NOTATION.

1.4 Design Considerations.

The two principles that serve respectively as lower and upper bounds on the choice of CGOL notation are:

(i) The notation should fairly match what the non-LISP community regards as a reasonable notation. In particular, users of

ALGOL, FORTRAN, PL/I etc, should not experience difficulty in guessing the meanings of those constructs common to those languages, e.g. assignment, application, arithmetic and relational operations, and the more familiar control constructs. This requirement need not apply to constructs peculiar to LISP, such as LIST, CONS, LAMBDA, EVAL, QUOTE and so on.

(ii) The notation should restrict itself to being a notation for LISP abstract objects, and not try to be a full-blown programming language with its own useful set of constructs. (This represents a point of departure from the MLISP philosophy[4].) Useful constructs should be added to LISP via the same channels (modulo choice of notation) that are used regularly by all LISP users to augment LISP, e.g. (DEFUN ...) or its CGOL equivalent.

There is a tension between these two requirements that is at present not appreciated by the bulk of the computer science community. That tension is brought about by the two very different techniques for specifying the syntax of ALGOL-like languages and the semantics of LISP programs. The former is phrase oriented, the latter is token oriented. ("Phrase" and "token" may be replaced respectively by "non-terminal" and "terminal", to use the formal language terminology.) A typical syntax specification uses BNF, i.e. a context-free grammar, whereas LISP programs are specified in terms of the meaning of atoms. The rich non-terminal structure of, say, the syntax of ALGOL is replaced by a trivial non-terminal structure for LISP, namely all non-atomic programs are of the form (FUNCTION ARGUMENTS).

Not all ALGOL-like languages are specified by their phrase structure. From time to time attempts are made to use the powerful macro facilities of assembly languages to implement "higher level" languages[1]. Since macros are identified by single tokens, they constitute a token oriented specification. A limitation of the approach is that the macro identifier must usually appear before its arguments. Also a macro interpreter that allows nested calls must be used.

The CGOL notation uses a token oriented approach[3] to fit comfortably with constraint (ii). Unlike macro based approaches, a given token may have either zero or one argument preceding it, in addition to any number following it (suitably delimited). This gives rise to the familiar association problem, which we deal with using Floyd's notion of precedence functions[6]. Each argument position of a token has an associated "binding power"; association is resolved in favor of the higher binding power. The binding power idea is due to Floyd, who called the binding powers "operator functions"; the term we use appears to have originated with CPL. The parsing algorithm we use differs from Floyd's in that ours is top-down whereas Floyd's is

bottom-up. The original version of the CGOL parser was implemented at Stanford in July 1970; its use of binding powers was adapted soon after by Smith for the MLISP system.[4] We discuss the details of the association problem in section 3.

2. EXAMPLES OF CGOL EXPRESSIONS

The following examples of CGOL expressions, together with the effect of doing (PRINT (CGOLREAD)) on them (i.e. their LISP translations), are given below. To aid the eye we shall use upper case for LISP and lower case for CGOL. Note that CGOLREAD demands an ALTMODE (not shown) after each CGOL expression.

```
1+1
(PLUS 1 1)

[1, '2+2', sin(.37*x+1)]
(LIST 1 '(PLUS 2 2) (SIN (PLUS (TIMES .37 X) 1)))

\x,y; 1/sqrt(x**2 + y**2)
(LAMBDA (X Y) (QUOTIENT 1 (SQRT (PLUS (EXPT X 2) (EXPT Y 2)))))

sstatus(toplevel, 'print *; eval read')
(SSTATUS TOPLEVEL '(PROG2 (PRINT *) (EVAL (READ))))

car m ~ & car m ← cdr m
(PROG2 NIL (CAR M) (RPLACA M (CDR M)))

'father' of x ← 'brother' of relative of y
(PUTPROP X (GET (GET Y RELATIVE) 'BROTHER) 'FATHER)

a(i,j) ← 3
(STORE (A I J) 3)

if i isnum and -j<i<j then !!1 else print i
(COND ((AND (NUMBERP I) (LESSP (MINUS J) I J)) (ABS I)) ((PRINT I)))

a.(b@c) = (a.b)@c
(EQUAL (CONS A (APPEND B C)) (APPEND (CONS A B) C))

for i in a@b do if 7<i<13 then return "In range"
(MAPC (FUNCTION (LAMBDA (I) (COND ((LESSP 7 I 13)
                                   (RETURN 'I/n/ /r/a/n/g/e)))))
      (APPEND A B))

f(x,y)(u,v,w)(i)
(((F X Y) U V W) I)

if j remainder 6 isin !'(1 5) then print j
```

```
                    else badlist ← j . badlist
        <note: - ! is a no-op unary function that expects its argument
         in LISP notation>
(COND ((MEMBER (REMAINDER J 6) '(1 5)) (PRINT J))
      ((SETQ BADLIST (CONS J BADLIST))))

while (a;b) do c
        <a handy way to put the stopping condition b in the middle of
         a loop body a;b>
(DO NIL ((NOT (PROG2 A B))) C)

define a "TO" b; if not a>b then a.((a+1) to b)
(DEFUN TO (A B) (COND ((NOT (GREATERP A B)) (CONS A (TO (PLUS A 1) B)))))
```

3. GRAMMAR

A CGOL expression is a string of sub-expressions and tokens. For example the expression "if a=b then 0 else i+1" has six constituents corresponding to the six items in the "construct"

```
        if a then b else c
```

In this example those constituents are:

- the token "if"
- the sub-expression "a=b"
- the token "then"
- the sub-expression "0"
- the token "else" and
- the sub-expression "i+1".

In turn, the sub-expression "a=b" has three constituents:

- the sub-expression "a"
- the token "=" and
- the sub-expression "b".

And the sub-expression "a" has one constituent, the token "a".

For those accustomed to BNF, the grammar of CGOL might look like:

```
<expression> ::= if <expression> then <expression> [else <expression>]
<expression> ::= <expression>=<expression>
<expression> ::= (<expression>)
<expression> ::= <expression>;<expression>
```

and so on. Since <expression> will be the only non-terminal, the left hand side of productions may be omitted without loss of information. Substituting variables for <expression> , we can then write CGOL rules as:

```
        if a then b [else c]
        a = b
        (a)
```

a ; b

and so on.

As they stand, such rules are ambiguous. Does a=b;c mean (a=b);c or a=(b;c) ? The problem is that each of "=" and ";" could take b as its argument. We say that b associates with the one that takes it as argument. Thus if "print a+b" means "print(a+b)", "a" has associated with "+" in preference to "print". In CGOL, all such disputes are resolved by binding powers, a sort of syntactic version of atomic valences. Thus if the right binding power (rbp) of "=" is 10 and the left binding power (lbp) of ";" is 1, then a=b;c is read as (a=b);c because the right hand argument slot of "=" pulls harder on b than does the left hand argument slot of ";" . (Ties are broken by associating to the left.)

Left and right binding powers are completely independent. Each is relevant when the expression can have a sub-expression at the left and right hand ends respectively. Thus the left binding power of "car" is irrelevant because "car a" does not have a sub-expression at the left. However, it has one at the right, so its right binding power is relevant. The opposite obtains for the suffix operator "isatom", which has a left binding power but no right binding power. In an expression like "if a then b else c", the right binding power applies to all three arguments even though only the last may actually be fought over by another operator to its right.

In addition to the left-right association problem there is the "dangling else" problem, named after an instance of the problem: does "if a then if b then c else d" mean "if a then (if b then c else d)" or "if a then (if b then c) else d" ? This problem is just a variant of the left-right association problem; the argument "else d" could associate with either the first or the second "if". In CGOL, all such disputes are resolved by associating to the closer operator. (For those who liked the atomic-valence analogy, imagine an inverse (square?) law for distance holds.)

The above two rules deal with all ambiguities that might arise in the CGOL grammar given below.

The following table lists the explicitly defined CGOL constructs together with their translation into standard notation. Except where otherwise noted, a,b,...,z denote CGOL expressions and A,B,...,Z their corresponding standard forms. The table has three columns, the CGOL form, the left and right binding powers when relevant (only given once when they are the same, or when only one is relevant), and the translation.

---------BRACKETING OPERATORS---------

(a)	0	A
f(a,b,...,z)	25 0	(F A B ... Z)
[a,b,...,z]	0	(LIST A B ... C)
---------QUOTING OPERATORS-------		
'a'	0	(QUOTE A)
"a" (where a is a string)		'a'
?a (where a is a character)		/a
#a (where a is any CGOL token)		A
!A (where A is a LISP S-expr)		A
---------DECLARATIVE OPERATORS---------		
\a,b,...,z;f	0	(LAMBDA (A B ... Z) F)
prog a,b,..,p; q;r;..;z	0	(PROG (A B ... P) Q R ... Z)
new a,b,..,p; q;r;..;z	0	(PROG (A B ... P) Q R ... (RETURN Z))
special a,b,...,z	1	(DECLARE (SPECIAL A B ... Z))
---------CONTROL OPERATORS--------		
eval a	1	(EVAL A)
a;b	1 0	(PROG2 A B)
a&b	1 0	(PROG2 NIL A B)
if a then b [else c]	2	(COND (A B) [(C)])
return a	1	(RETURN A)
while a do b	2	(DO NIL ((NOT A)) B)
for i in l do f	2	(MAPC (FUNCTION (LAMBDA (I) F)) L)
--------STORAGE OPERATORS-------		
a of b ← c	25 1	(PUTPROP B C A)
a←b (a is atomic)	25 1	(SETQ A B)
x(a,b,...,z)←y	25 1	(STORE (X A B ... Z) Y)
a of b	25 24	(GET B A)
a assoc b	25 24	(ASSOC A B)
---------LIST OPERATORS---------		
a.b	14 13	(CONS A B)
a@b	14 13	(APPEND A B)
---------RELATIONAL OPERATORS--------		
a=b	10	(EQUAL A B)
a ne b	10	(NOT (EQUAL A B))
a eq b	10	(EQ A B)
a<b<...<z	10	(LESSP A B ... Z)
a>b>...>z	10	(GREATERP A B ... Z)
a isin b	10	(MEMBER A B)
a isatom	10	(ATOM A)
a isnum	10	(NUMBERP A)
---------LOGICAL OPERATORS-------		
not a	9	(NOT A)

```
a and b                          8       (AND A B)
a or b                           7       (OR A B)

---------ARITHMETIC OPERATORS-------
|a|                              0       (ABS A)
+a                               20      A
a+b                              20      (PLUS A B)
-a                               20      (MINUS A)
a-b                              20      (DIFFERENCE A B)
a*b                              21      (TIMES A B)
a/b                              21      (QUOTIENT A B)
a remainder b                    21      (REMAINDER A B)
a**b                             22      (EXPT A B)

---------I/O OPERATORS-------
print a                          2       (PRINT A)
princ a                          2       (PRINC A)
write a                          2       (PROG2 (TERPRI) (PRINC A))
uread a b ... z (a-z are tokens) (UREAD A B ... Z)
uwrite a b ... z                 (UWRITE A B ... Z)
ufile a b ... z                  (UFILE A B ... Z)
load a b ... z (a-z are tokens)  (FASLOAD A B ... Z)
newline                          (TERPRI)
```

In addition to the above, CGOL "knows" about all the unary functions in LISP. It does this by testing (ARGS token) when said token is undefined. Thus although "car" does not appear in the above table, CGOL knows that CAR is a LISP function with one argument. CGOL treats all such functions f as though they were defined as

```
f a                              25      (F A)
```

When in doubt you can always drop back into LISP by using "!". However, that should be rarely necessary - it is intended mainly for non-procedural items such as lists for doing MEMBER and ASSOC in. If you can't recall the CGOL form of an expression (F A B ... Z) you won't go wrong by writing f(a,b,...,z). Thus if you forget the form "1+1", you can write "PLUS(1,1)" and it will translate correctly. By the same token, any LISP function not catered for in the above table can be written as "f(a,b,...,z)", e.g. "sstatus(toplevel,nil)".

At first sight the binding powers may seem a lot to learn. However, they have been chosen on the basis of the data types their operators take as arguments and return as results in order to minimize the need for parenthesization.[3] If you want to use CGOL notation but don't want to have anything to do with binding powers, simply parenthesize every CGOL expression as though you were writing in LISP. However, if you omit all parentheses (apart from those needed in constructs of the form f(a,b,...,z)) you will not often go wrong.

Most often you will want parentheses for grouping in arithmetic expressions when the default priority ranking (|| + - * / mod **) gives the wrong grouping, and when procedural expressions occur as arguments to non-procedural expressions, e.g. "if a then (b;c)", "(print a) + b", "a←(b←read)+1" and the like.

Some operators have a right binding power one less than their left binding power. This is to make those operators right associative. For example, "a of b of c" is parsed as "a of (b of c)" (since oftener than not that's what was intended), and "a@b@c" as "a@(b@c)" (for efficiency). An interesting pair of right associative operators is ";" and "&". These are duals of each other. They both evaluate their arguments in the same order, but differ in the value they return: a&b returns a, a;b returns b. By making both of them right associative and giving them the same priority, they interact in an elegant way. Suppose you want to evaluate a,b,...,z and to return the value of k. Then the expression a;b;...k&...;z has the desired effect. That is, follow every argument but the last with ";", except for the one you want the value of, which if it is not the last should be followed by "&". A common use for "&" is in tidying up after computing some value, e.g. "x & x←2" will set x to 2 and return the old value, "a ← (b & b←a)" will swap the contents of a and b without using a temporary variable, and so on. Note that all this is really a feature of LISP rather than of CGOL; however, the notation makes it easier to see at a glance the intent of what would be relatively difficult to follow in LISP.

4. THE DEFINE FACILITY

The CGOL analog of DEFUN is "define". In addition to allowing the user to attach a lambda expression to some functional property of an atom, it gives him some syntactic capabilities as well.

The basic form is
define <pattern> [,bp [,bp]]; body
For example, the following could serve as a definition of "@":
define a "@" b, 14, 13; if a then car a . cdr a @ b else b
(For readability, one would normally put in more parentheses than we have here.)
In this example, the pattern gives the rule for this operator, and the two numbers give the two binding powers. The body is what would normally appear as the body of a DEFUN.

At present the allowable forms of patterns are few in number. You may write a sequence of variables separated by one or more tokens in string quotes (letters inside string quotes must be capitalized - this is the one place where a distinction is drawn between upper and lower case, in the sense that the reader maps all lower-case letters not in string quotes to upper-case before thinking about what they

mean). The variables stand for CGOL expressions and the strings for tokens (recall that a CGOL expression is a sequence of sub-expressions and tokens). The sequence may start and end with either tokens or variables.

The first token in the pattern is called the operator, and the remaining tokens are called delimiters. The operator is said to have been defined. An operator may be defined twice only if it takes a left argument in one case and no left argument in the other, this being a criterion CGOL uses when deciding what a particular token in a program means. In the above table, the operators "(", "+" and "-" all have such dual meanings. The delimiters may appear in arbitrarily many definitions, and arbitrarily often in each. However, note that a delimiter's binding power is set to the minimum of its previous binding power and the right binding power of the operator being defined. If the delimiter has already been defined to be an operator with a left argument (the term is LED, for LEft Denotation - NUD is the case when the left argument is missing, or NULL), and this new binding power is less than its old, an error message is given.

The default binding power is 25 if none is specified, and applies to both left and right binding powers. If one binding power is given it is the left and right binding powers. If both are given, they are respectively the left and right ones.

Like LISP, CGOL is a one-pass system. This is so that a user can type in a definition and have it take effect immediately. This conflicts with the requirement in any system offering sophisticated syntax that it be possible to use the syntax of an operator before it has been defined. This requirement is nice in general, and essential for mutually recursive function definitions. To get around this problem, you may define the syntax of an operator at any time without giving its semantics, simply by omitting the body of the define command. This is not an elegant solution, and a later version of CGOL may deal with this. (A possible solution is to keep around pieces of unparsable text until they become parsable, and then parse them. Even more dramatic is the solution of not parsing anything until it is to be evaluated, i.e. dropping the unparsability condition from the previous solution.)

5. EXTENDING CGOL

It is possible to add to or change the definitions of the rules of CGOL (the ones in Section 2). To see examples of such definitions, look under the heading BASE COMPONENT in the file AI:PRATT;CGOL > . Given the above table, you should be able to get a rough idea of how to write such definitions.

The meaning of

```
infix "+" 20 is "PLUS"
```

is as follows. The infix operator "+" is being defined with left and right binding power 20. (Had I said "infixr" it would make "+" right associative by making its right binding power 19, one less than its left.) The translation of "a+b" is then (PLUS A B) . Since the argument positions are "standard" for infix operators, the only information you really need to supply is what the LISP function name is, so you say "is "PLUS"" . You don't have to use "is" - if you want you can spell it out by saying ["PLUS", left, right] , which is a CGOL program to build a list whose three elements are the atom "PLUS", the left hand argument and the right hand argument. Notice the use of this technique when defining

```
infixr "&" 1 ["PROG2", nil, left, right]
```

We can't say 'is "PROG2"' because that would mean '["PROG2", left, right]'.

6. COMPILATION

Temporarily there is some awkwardness in compiling which will hopefully go away soon. In the meantime, a CGOL program is compiled by saying

```
maklisp <filename>
```

to CGOL, using the same syntax as for the other file manipulating commands (uread, etc). CGOL will then produce a file with the same FN1 (if using MIT's ITS) and with FN2 = LISP , and will return the area the file is on, e.g. (DSK SMITH). This file may then be compiled in the usual way. This also provides a convenient way of exporting CGOL programs to sites without CGOL - just send them the LISP translation.

Sometime it will be possible to have one's CGOL file compiled by NCOMPL without any action on your part provided your file begins with the incantation (CGOL). See the discussion at the end of section 1.3.

7. IMPLEMENTATION

CGOL is not resource-hungry. It consumes about 1K of binary program space and 1.5K of list space. It uses the LISP reader for lexical analysis, and so loses no more time on this account than does standard notation. The parser executes about ten instructions per lexical item, a negligible amount. The semantics of most CGOL operators is trivial enough that they take little time to execute. Unlike systems based on BNF grammars, you can extend the CGOL notation

on-line with none of the overhead associated with systems that have to consider the whole grammar before admitting a new rule.

8. COMPARISON WITH MLISP

The similarities between CGOL and Smith and Enea's MLISP[4] are:

(i) the use of ALGOL-like notation for LISP S-expressions;

(ii) the use of numeric operator precedence functions to resolve association problems;

(iii) the ability to export LISP translations of MLISP/CGOL programs to sites supporting LISP but not MLISP/CGOL.

The differences are:

(i) MLISP is a sophisticated programming language offering many facilities not appearing in LISP. These facilities are only visible to the speaker of MLISP, and vanish if he wants to use them while speaking LISP. (Due to the ubiquity of LISP's oblist, the user can get at them from LISP, though they are undocumented and have names starting with & to identify them as sytem names not for general consumption.) Assignment to S-expressions is a particularly complex example. In contrast, CGOL offers nothing but an alternative notation for things already meant for consumption by LISP users. This enables CGOL to be very small, both with respect to its implementation and its manual.

(ii) MLISP is a system that the user must call from the monitor, whereas CGOL is a package that can be loaded into LISP when the need arises. Hence a non-CGOL user can read a CGOL file without having to commit himself to a CGOL-oriented system when he loads LISP. In fact, when the I/O details are worked out as in Section 1.3 he may never know that he was reading a CGOL file.

9. KNOWN DIFFERENCES BETWEEN THE MANUAL AND THE IMPLEMENTATION

a<b<c not implemented. (Only a<b). Similarly for a>b.

Can't use GO in the body of "new". Use
prog(a, b, m, c, go m, d)

Use ^ for exponentiation

Use mod for remainder (not really mod since the PDP-10 treats negative moduli incorrectly).

Delimiters presently get lbp 0.

Bibliography

0. Floyd, R. W., Syntactic Analysis and Operator Precedence. 3, 316-333 (1963).

1. Leavenworth, B.M., Syntax macros and extended translation. CACM, 9, 11, 790-793 (1966).

2. Moon, D. MACLISP Reference Manual, Project MAC, MIT, December 1975.

3. Pratt, V. R., "Top-Down Operator Precedence". SIGACT/SIGPLAN Symposium on Principles of Programming Languages, Boston, 1973, 41-51.

4. Smith, D. C., "MLISP." STAN-CS-70-179, Stanford University, 1970.