STANFORD ARTIFICIAL INTELLIGENCE PROJECT August 19, 1964
Memo No. 21

LISP 2 SPECIFICATIONS PROPOSAL

by R. W. Mitchell

Abstract: Specifications for a LISP 2 system are proposed. The source language is basically ALGOL 60 extended to include list processing, input/output and language extension facilities. The system would be implemented with a source language translator and optimizer, the output of which could be processed by either an interpreter or a compiler. The implementation is specified for a single address computer with particular reference to an IBM 7090 where necessary.

Expected efficiency of the system for list processing is significantly greater than the LISP 1.5 interpreter and also somewhat better than the LISP 1.5 compiler. For execution of numeric algorithms the system should be comparable to many current "algebraic" compilers.

Some familiarity with LISP 1.5, ALGOL and the IBM 7090 is assumed.

The research reported here was supported in part by the Advanced Research Project Agency of the Office of the Secretary of Defense (SD-183)

LISP 2 SPECIFICATIONS PROPOSAL

by R. W. Mitchell

INTRODUCTION:

An algorithmic language and an implementation is specified. The language provides for numeric and list processing algorithms and includes facilities for user extension of the language. The basic language is very similar to ALGOL 60 with extensions for the facilities and data types which were not available. For this reason, the basic language is presented in reference to the revised ALGOL 60 report.[1] Some familiarity with LISP 1.5 and the IBM 7090 is also assumed.

The facilities for extending the language allow for definition of new data types, new operators and new syntactic constructions, all in terms of the basic language and previously defined extensions. The resulting compiled code for extended language items may be expressed in terms of symbolic machine code when desired, or it will be the result of source language manipulations.

The system would basically be implemented as four processors. These are a source language translator, a source language optimizer, an interpreter and a compiler. The source language translator would translate the source language statements to s-expressions. These would be similar to the s-expression input of LISP 1.5. Most syntax analysis would be in the translator.

The source language optimizer woudl convert the s-expressions into an optimized, packed form of s-expressions, hereafter called p-expressions. The p-expressions would occupy approximately one-third the space of s-expressions and could be interpreted in approximately one-fifth the time of the LISP 1.5 interpreter. There would be associative and commutative law, for loop, and common sub-expression optimization and the effects of declarations for formal parameters, local variables and data types would be propagated into the expressions and statements.

The interpreter and compiler would accept p-expressions as input and execute or translate the code to machine language, respectively. The compiler would perform optimization which is dependent on the particular machine language.

The control and communication stack is significant in obtaining efficient execution and the information on the stack (e.g. parameters, local variables and return instructions) would be directly usable. The same stack format is used for both interpreted and compiled code, but the format is oriented toward optimal execution of compiled code.

The implementation is specified for a single address computer, with particular reference to the IBM 7090 where necessary, and the specifications should be applicable for most such computers. The only major changes should be with regard to the compiler optimization.

Throughout the implementation, the design philosophy has been toward optimum speeds for the simpler and more frequently used features of the language. Added time cost for the use of more complicated or infrequently used features would, to a large extent, be paid only when such features were actually used.

LANGUAGE:

BASIC LANGUAGE:

The basic language is similar to ALGOL 60. Features of ALGOL 60 which cause significant difficulty have been altered for efficiency or deleted if the feature was not that important to the programmer.

Data Types

The new data types are atom and s-expr. An atom datum may have an associated value of any other type, general property lists, and are the basic units for constructing data or type s-expr. An s-expr datum is an s-expression as defined in LISP 1.5, i.e., a binary tree structure.

An atom in an s-expression may be a constant, variable or procedure of any type. If the type is other than atom, an atom type datum is constructed with the appropriate associated value. The null atom nil is an added basic symbol.

The basic symbol Boolean is replaced by the basic symbol logical.

Operators and Functions

The operators which have been added for the above data types are l, r, ', ⌜ ⌝ , a and n. (These are car, cdr, cons, list, atom, and null of LISP 1.5).

The following standard functions are defined:

(to be specified later)

Statements

The for statement has been extended in two respects. A <for clause> may be

for <s-expr variable> ϵ <list datum> do

(e.g. for j ϵ m do, where j is a variable to type s-expr and m is some expression yielding a list value).

Execution of a for statement with such a for clause will cause assignment of each successive element of the list to the variable and execution of the statement following do for each assignment. The statement is concluded when the variable is given the value nil.

The basic symbol and is added and one may express concurrent for loop controls with statements such as

for i:= 1 step 2 until n and j:= f(x) do S_1;

for j ϵ r and k ϵ m, n and i:= 1 step 1 do S_2;

The for control variables are assigned and tested from left to right and the for statement is concluded when any variable is assigned a value which concludes its < for list>.

To allow full flexibility of the conditional if statement, two forms are allowed.

1) if p then S_1 else S_2;

2) if p do S;

Also another statement is defined which one might call a conditional statement. This is

while < logical expression > do S;

This statement will cause S to be executed if the logical expression is true and control cycyles to re-analysis of the entire while statement. If the logical expression is false then control passes to the next statement without executing S.

The return statement

return;

or

return (<expression>);

causes exit from this procedure to the calling procedure. The second form is used when returning from a function and the value of the expression is the returned value of the function.

Procedures

In procedure declarations the specifications for mode of parameter transmission may be value or name, and value specification is assumed if neither is present. The name specification is not the ALGOL 60 "call by name", but rather a "call by address". Thus a name parameter may be assigned a new value in the procedure and that value assignment is effected in the calling procedure, but there is not the effective literal substitution and its complications.

Complete specifications are required in procedure declarations if the item is other than a simple variable of the same type as the procedure.

Single statement procedure declarations are permitted. The parameter specifications are included in the formal parameter list and global and local variables are analogously declared in a second list. These are followed by the assignment symbol (:=) and the defining expression. For example:

integer procedure step (real u):= if $o \leq u \wedge u \leq 1$ then 1 else 0;

LANGUAGE EXTENSION FACILITIES

These facilities are included to allow programmer specification of new data types, operators and syntactic constructions. The specifications are normally in terms of the basic language and previously defined extensions, but alternatively supplemental routines for the language processors may be included. These facilities should permit convenient tailoring of the language to a particular applications area and in general permit the programmer some degree of freedom in mapping the programming language to the problem.

Data Types

New data types may be defined by Cartesian product ($\otimes$) and direct union ($\oplus$).[2] For example:

define type 'complex' = real $\otimes$ real;

define type 'number' = real $\oplus$ integer;

(The apostrophe (') delineates the effective basic symbols which are defined by these statements and all uses of the identifier as a type must be so delineated.)

Since operators must be defined for new data types, some functions are necessary for accessing the constituents of such data and forming such data from other data. The following are defined.

For 'X' = A ⊗ B

'Y' = C ⊕ D

and x ∈ X, a ∈ A, etc.

1 E (x) = a

r ∈ (x) = b

∈ 'X'(a,b) = x

∈'Y'(c) = y

∈'Y'(d) = y

∈(y) = c or do whichever is the case

∈pC(y) = if ∈(y) = c then true else false

∈pD(y) = if ∈(y) = d then true else false

Operators

Operations on new data types may be defined as procedures or current operators of the language may be defined for application to elements of the new data types. For example:'

```
'complex' procedure compplus(a,b) =
     ∈'complex'(1 ∈ (a) +1 ∈ (b), r ∈ (a) + r ∈ (b))
define 'complex' operator + (a,b) =
    ∈ 'complex' (1 ∈ (a) + 1 ∈ (b), r ∈ (a) + r ∈ (b))
define 'complex' operator + (a, real b) =
    ∈ 'complex' (1 ∈(a) + b, r ∈ (a))
```

Syntactic Constructions

New syntactic constructions must be defined with created (") basic symbol as a prefix. The defined constructions are primarily a notational convenience and the programmer must not create ambiguous syntax as a result of their use. As an example let us define the LISP 1.5 COND in terms of
if -- then -- else -- expressions.

define expression 'cond'($(p_1,e_1)(p_2,e_2)$ ... (p_n,e_n) e_{n+1})

= if p_1 then e_1 else

if p_2 then e_2 else

... if p_n then e_n else e_{n+1};

The programmer could then write

x:= 'cond'$((q_1,a_1)(q_2,a_2)(q_3,a_3)\ a_4)$

and the processing would result in

x:= if q_1 then a_1 else if q_2 then a_2, else

if q_3 then a_3 else a_4;

as the effective statement.

Currently expression and statement are the definable constructions.

P-EXPRESSIONS

The p-expressions are packed s-expressions and thus have the same logical structure but a different storage convention. A p-expression is a contiguous sequence of six bit bytes. There are sixty-four basic bytes and these are then followed by up to two bytes to form a single part of an s-expression. The determination of the basic bytes was the result of a crude application of information and coding theory.

The chart on the following page defines sixty-three of the basic bytes. The other three are open for later definition. The term "general expression" means the following item is any valid p-expression and the result of its evaluation is the desired input for this operation. All 3X bytes are followed by general expressions.

The 2X, 4X and 70 bytes are followed by one byte, n, per necessary element. If the byte, n, is less than 40 then the n^{th} parameter is referenced. If the byte, n, is equal to or greater than 40 then the $(n-40)^{th}$ local variable is referenced.

The 00 byte is the) of s-expressions and the 76 and 77 bytes are the (, but a count is included which, when added to the current byte pointer, will set the pointer to the byte following the corresponding).

The l and r composition bytes, 23 and 33, are followed b one byte which gives the composition. This byte is interpreted as follows:

00	ll
01	lr
02	rl
03	rr
04	lll
05	llr
.	
.	
.	
76	rrrrl
77	rrrrr

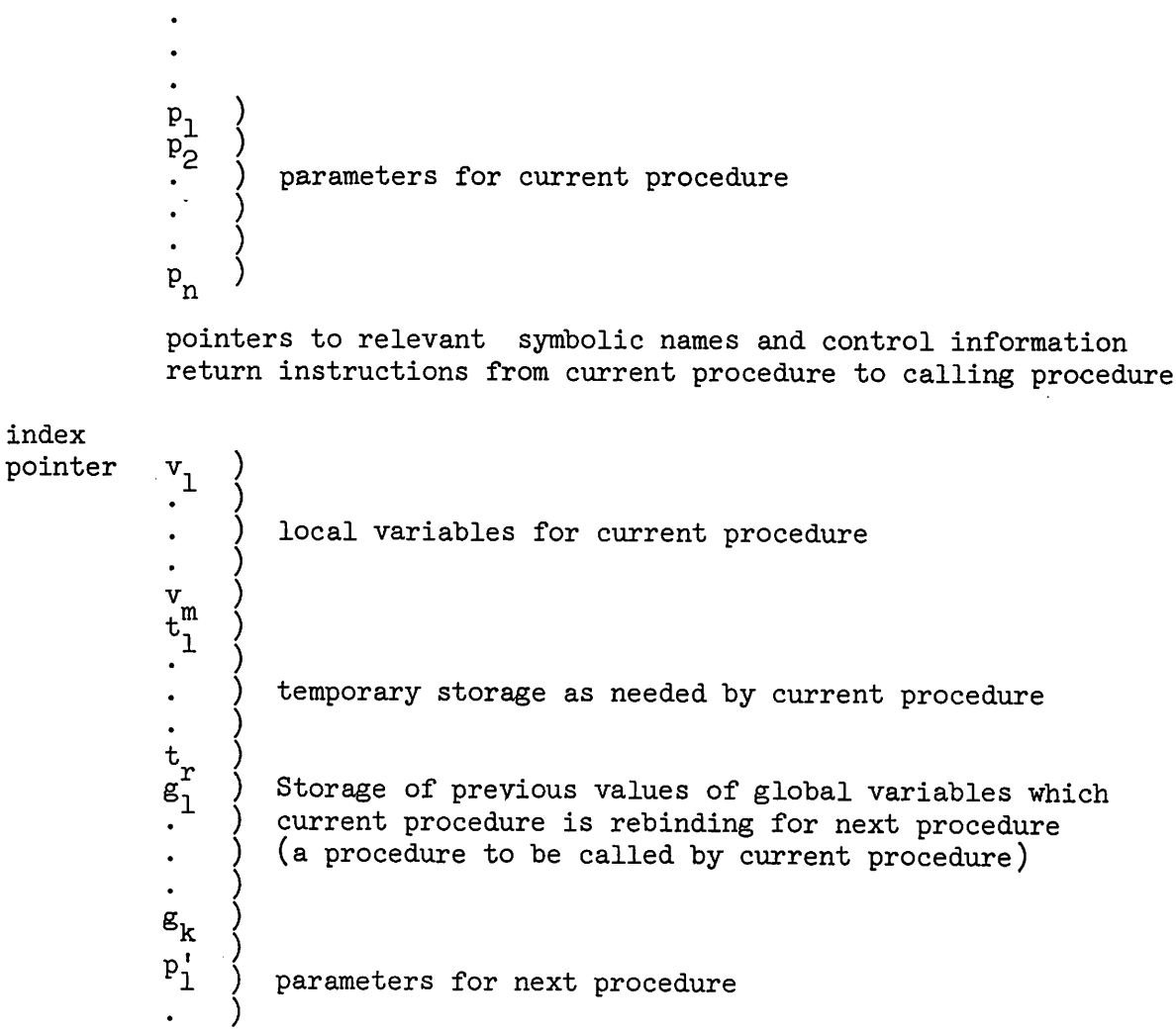
The temporary variable byte, 16, is used for optimizer generated local variables for which the compiler should use high speed registers whenever possible.

IMPLEMENTATION

CENTRAL STORAGE STACK

The central storage stack provides a mechanism for parameter communication, local variable storage, procedure linkages and control information. The stack is a contiguous block of memory of variable length. An index register always points to the first available cell upon entry to a procedure and a counter is maintained of the current number of stack cells used in the procedure for local storage.

The general format of the stack is:



BASIC INTERPRETER BYTES

	.0	.1	.2	.3	.4	.5	.6	.7
0	end	if expression	if statement	for statement	:=	return	goto relative (one byte)	goto (general) (general expression)
1	parameter (value)	parameter (reference)	local variable	local array	global variable (2 bytes for name)	global array (2 bytes per name	temporary variable	
2	ℓ	r	.	ℓ and r composition	a	u	= (value)	= (sexpr)
	← parameter or local variable →							
3	ℓ	r	.	ℓ and r composition	a	u	= (value)	= (sexpr)
	← general expressions →							
4	t_i	$-_i$	x_i	÷	t_r	$-_r$	x_r	/
	← parameters or local variables →							
5	subscript	functional parameter		s-expr subelement (must be translated)	ϵ sexpr	step	until	while
6	constant (2 bytes for "value")	0	1	true	false	nil	set global	restore global
7	procedure call 1 byte for name-parameter are local parameters or variables	procedure call 2 bytes for name-parameters are general expressions		convert integer local parameter or variable value to real value	statement label	procedure head (2 bytes for name)	subelement w/ 1 byte for length (begin)	subelement w/ 2 bytes for length (begin)

,)
.)
p_2')

pointers to relevant symbolic names and control information.
return instructions from next procedure to current procedure.

α empty
.
.
.

At the time that control is passed to the next procedure, the index pointer will be set to α.

Each of the two locations marked "pointers to relevant symbolic names and control information" contains two pointers - one to the name of the calling (current) procedure and one to the name of the current (next) procedure. With the symbolic names is information giving the type, number, names, etc. of parameters and local variables.

With this format, simple variables which are value parameters and local variables may be referenced by appropriate positive and negative displacements from the index pointer. The addresses of name and array parameters and entries to procedural parameters are available on the stack and call by value arrays are stored as a local array by the procedure initialization.

SOURCE LANGUAGE TRANSLATOR

The source language translator will convert the source language to s-expressions. The syntactic heirarchy will be represented by the s-expression heirarchy with all syntactic constructions in prefix form. All character scanning and syntax recognition routines will be in this processor.

OPTIMIZER

The optimizer will perform source language optimization. Necessary features of the processor will be:

a) The propagation of all type declarations into the program body so that operation symbols (e.g. +, /) whose operands can be of several types will be converted to the required type operation for each case.

b) The conversion of all parameter and local variable references to relative references (e.g. the i^{th} parameter, the j^{th} local variable).

c) The preparation of a property list for each procedure giving the symbolic names of parameters, local and global variables, all type information, etc.

d) The conversion of operations on data of programmer defined types to operations on basic types (if the programmer has not supplied routines for the operation).

e) The conversion of programmer defined syntactic constructions to basic constructions (if the programmer has not supplied routines).

Optimizer features that are optional are such asctions as:

a) for loop optimization;

b) single calculation of common expressions.

INTERPRETER

Processing of p-expressions for direct answers will be performed by the interpreter. The interpreter will be a transfer tree with all nodes using a common byte accessing routine and the terminal nodes will perform actual desired operations. The interpreter processing could be expressed in source language but machine language should be used, since efficient sequential byte processing and transfer trees are quite machine dependent. A short internal stack will be necessary for most of the basic byte forms but all general procedure calls will use the central stack as will recursive calls of the interpreter.

The byte accessing routine will keep a pointer to the current byte and allow calls requesting skipping any number of bytes or accessing up to six bytes. (Normally, only one or two bytes will be desired by any request). The initial node of the transfer tree will access one byte and transfer to the proper basic byte processor which will then access the following bytes and transfer to the next node or perform a specified operation, whichever is the case.

COMPILER

Translation of p-expressions to optimal machine language for immediate or later execution will be performed by the compiler. The compiler could be expressed in source language with reasonable efficiency. (Special l and r routines for p-expressions would use the byte accessing routine). Data of type set would be particularly useful for coding the compiler since most aspects of the first level compiler processing are very similar to a transfer tree. However, since the main work of the compiler is determing optimum machine code generation for a given case, it is not recommended that the compiler be written in machine language.

In generating the optimal code the compiler must consider use of high speed registers for temporary variables; the calculation of a result into a particular type of register, depending upon its use; and the ordering of commutative operations.

LANGUAGE EXTENSIONS

The language extensions should be implemented in two manners. The system will contain straightforward routines in the Optimizer for converting to the equivalent base language for any defined operation or construction. Secondly, the programmer may specify routines to be executed when the compiler or interpreter encounters such items.

ADDITIONAL FEATURES FOR CONSIDERATION

DATA TYPES

Two additional data types have been suggested and should be considered. These are string and set. The for statement should be extended for both.

The string data would be like ALGOL 60 strings, but string values could be assigned to variables and the following basic functions would be defined:

first (n, s)	-	first n basic symbols in string s
last (n, s)	-	last n basic symbols in string s
length(s)	-	number of basic symbols in string s
concat(s_1,s_2)	-	concatenate the strings s_1 and s_2 in that order

The set data would be similar to arrays but sets would not be ordered. Basic set operation would be defined. The set data could be implemented as tables with entries ordered by "numeric" value, then binary search, merge, merge equal entries, etc. would provide efficient processing for basic operations. Whenever a table is established the odd cells could be set with an "empty" marker, thus allowing for efficient growth of the table.

SEPARATE PROCEDURE HEADS

Since the compilation of an efficient procedure call is very dependent on the procedure specifications, the preparation of separate procedure heads containing the parameter, local variable and global variable specifications would allow more efficient processing and is almost essential for reasonable separate compilation of procedures with free variables or which reference routines with free variables.

SUBSTITUTION PARAMETERS

The ALGOL 60 "call by name" results in effective source language substitution of actual parameters for formal parameters. This has not been included because of the variable name binding and storage reclaimer

problems and since functional parameters provide most of the facility (as well as additional facilities). These problems could be overcome, but the gain has not yet been justified.