



STANFORD RESEARCH INSTITUTE  
Menlo Park, California 94025 · U.S.A.

November 1969

A LISP-FORTRAN-MACRO INTERFACE FOR THE PDP-10 COMPUTER

by

John H. Munson

Artificial Intelligence Group

Technical Note 16

(Supersedes Technical Note 12)

SRI Project 8259

This research was sponsored by the Advanced Research  
Projects Agency and the National Aeronautics and  
Space Administration under Contract NAS 12-2221.

## SUMMARY

An interface has been devised for use on the PDP-10 computer that allows FORTRAN (or FORTRAN-compatible MACRO) subroutines and functions to be run under the LISP operating system (specifically, the LISP written at Stanford University, and described in Stanford Artificial Intelligence Project Note SAILON 28, by Lynn Quam). A considerable effort has been made to endow the interface with generality and ease of use, as much as could be achieved without tampering with the FORTRAN and LISP operating systems and compilers. The operating features of the interface are as follows:

(1) The interface and the various FORTRAN subprograms are loaded with the regular loader for relocatable programs that is available under the LISP system.

(2) FORTRAN subprograms may be called at will from within LISP. To the LISP programmer, these subprograms look exactly like LISP functions. Up to four arguments may be passed. The arguments and the returned value may be integers, floating-point numbers, LISP atoms, or other S-expressions.

(3) An optional call-by-name mechanism is incorporated: after the execution of the FORTRAN subprogram, LISP may access the calling arguments, which may have been changed by the subprogram.

(4) At any point or points in the FORTRAN subprogram structure, a function call may be made back into LISP. Again, the arguments may be of various types. The recursion ends here, however. Neither the interface nor the FORTRAN system, as presently constituted, can handle a second recursive entrance into FORTRAN.

The current form of the interface represents two major improvements over the earlier version, namely, the handling of non-numeric arguments and the ability to call back into LISP. By thus allowing FORTRAN to "dip into" LISP conveniently, we reap at least three benefits. The first is simply a vast improvement in intercommunication, with FORTRAN now able to access information within LISP data structures. The second is the ability of FORTRAN to invoke LISP functions for those activities (notably input/output) that are presently missing because the FORTRAN operating system

is not in operation. (We may be driven, however, to remedy this situation more directly in the future.) The third benefit is the use of the improved interface as a link for two-way communication in which large structures or quantities of data are explicitly shared by the LISP and FORTRAN subsystems, for example in the form of EXARRAY's.

## TERMINOLOGY

The term "FORTRAN" is used herein as a shorthand for any and all program codes that can be loaded by the PDP-10 relocatable program loader operating under the LISP system. In general, these will include subprograms written in MACRO as well as in FORTRAN. Top-level subprograms called directly through the interface, and those called from FORTRAN subprograms, must be FORTRAN-compatible in their calling sequences. The same applies to subprograms calling back to LISP through the interface. Elsewhere within the "FORTRAN" side, FORTRAN compatibility need not be maintained, as long as consistency is maintained at each point.

Throughout this paper, we use the term "subprogram" to refer generically to SUBROUTINES and FUNCTIONS in FORTRAN. FUNCTIONS and SUBROUTINES can be used interchangeably if the returned value, or lack thereof, is of no consequence. At present (contrary to the manuals) FORTRAN FUNCTIONS do store back into their calling arguments and thus allow the call-by-name mechanism. Furthermore, listings indicate that SUBROUTINES may not save and restore all arguments. Thus, it is perhaps preferable to write everything as functions.

## ARGUMENT TYPES AND CONVERSIONS

Both FORTRAN and LISP recognize two types of numeric quantities, integers and floating-point (or real) numbers. Since the FORTRAN and LISP systems represent each of these differently, the interface performs a conversion on every numeric argument (LISP functions NUMVAL and MAKNUM

are invoked to do these). Pointers to non-numeric atoms and S-expressions in LISP are transmitted unchanged to FORTRAN, where they appear as single-word variables, with the pointer in the right half word and zero in the left half.

FORTRAN has a limited data type called "literal," used for symbolic information. Literal constants such as 'ABC123' may be coded into a FORTRAN program and are stored as consecutive words of 7-bit ASCII characters, five to a word, left justified, with the last word filled out with blanks if necessary, and followed by a word of all zeroes. Since there is no literal variable type, any variable literal must be made up within a variable or array of some other type. The interface will convert a FORTRAN literal into the LISP pointer to the atom whose print name matches the literal. (The interface scans the literal, character by character, until it encounters either an ASCII blank or zero. The maximum length allowed is 25 characters.) How the interface is aware of a literal is described below.

Table I summarizes the conversions that are applied by the interface to every argument (when LISP calls FORTRAN or vice versa) and to every result value when the corresponding returns are made. Also, when a call-by-name argument reference is made by LISP, the argument is converted from a FORTRAN data type back to a LISP data type.

TABLE I

CONVLF and CONVFL Conversions

<u>LISP Quantity</u>	<u>FORTRAN Quantity</u>	
	<u>Type</u>	<u>Code</u>
Integer (INUM or FIXNUM)	Integer*	0
Real (FLONUM)	Real*	2
Pointer to a non-numeric S-Expr. (machine address)	Logical*	3
	Octal	4
Pointer to Atom (machine address)	Literal	5
	Dbl. Prec.*	6
	Complex*	7

—————> Indicates LISP-to-FORTRAN conversion (CONVLF)

←———— Indicates FORTRAN-to-LISP conversion (CONVFL)

\* Indicates existence as FORTRAN variable type

The LISP-to-FORTRAN conversion routine in the interface, CONVLF, determines the nature of each LISP quantity by examining it (with NUMBERP). If numeric, the quantity is converted with NUMVAL; if not, it is passed unchanged. The FORTRAN type-code is also generated and stored (even though called FORTRAN routines seem to ignore the codes), so that in the case of a call-by-name argument the reverse conversion can be performed properly later. The "octal" code is generated for all non-numeric pointers, because intuitively it seems most appropriate.

The reverse, or FORTRAN-to-LISP conversion routine (CONVFL), determines the type of the FORTRAN quantity by examining the type code as shown in Table I. These type codes are taken from the standard FORTRAN subprogram calling sequence (see the PDP-10 FORTRAN IV manual, Appendix 4). If a FORTRAN routine is to call LISP with an "octal" or "literal" variable quantity, that quantity must be stored in (or EQUIVALENCE'd to) a logical variable (for an octal quantity) or a complex or double-precision variable

(for a literal quantity) so that the latter may be used in the calling sequence to establish the type for the conversion. This dodge requires a slight amount of care, but it should not impose any real hardship. On the other hand, an octal or literal constant may be coded into the calling sequence without further ado.

#### CALLING FORTRAN FROM LISP

A call to a FORTRAN subprogram appears in LISP exactly as would a call to a LISP function of the same name; thus, the FORTRAN function

```
FUNCTION ITEST (I,J,X,Y)
LOGICAL Y
etc.
```

might be called from LISP as

```
(ITEST 3 JJ 4.0 NIL) .
```

To prepare LISP for this, we define a dummy function ITEST in LISP as follows:

```
(DE ITEST (I J X Y) (IFORT4(FORTREF(QUOTE ITEST))I J X Y)).
```

This causes ITEST as just defined to occur as a real LISP function, so it can be called in LISP. When it is called, FORTREF takes the name "ITEST" to the loader symbol table, and returns with the address of the FORTRAN function ITEST (which is then stored on the property list of ITEST under "FORTFUNC"). IFORT4 is an entry into the interface itself. The interface takes the function address (which it receives as its first actual argument, in accumulator 1) and prepares a FORTRAN calling sequence to that address. The interface then performs the CONVLF conversion on the remaining arguments, stores them in the calling sequence, and enters the FORTRAN subprogram.

When the FORTRAN subprogram does a RETURN, control comes back to the interface. The value of the function (or garbage in the case of a subroutine) is converted back by CONVFL, and control returns to LISP.

For each FORTRAN subprogram to be thus called from LISP the only necessary preliminary is a dummy function definition similar to that above. The identifier IFORT4 is varied in two ways, as appropriate to the subprogram called. The terminal digit tells the number of arguments (limited to 4 by a constraint of LISP), and the initial letter is I for an integer result to be returned, null for a floating-point (real) result, and P for a pointer result. Thus, the interface provides the following entries:

<u>Number of Arguments</u>	<u>Integer Result</u>	<u>Real Result</u>	<u>Pointer Result</u>
None	IFØRT0	FØRT0	PFØRT0
1	IFØRT1	FØRT1	PFØRT1
2	IFØRT2	FØRT2	PFØRT2
3	IFØRT3	FØRT3	PFØRT3
4	IFØRT4	FØRT4	PFØRT4

(Pardon our previous laxness about slashed O's.)

Upon return to LISP, accumulators 0 and 6-17 are restored to the values they had when LISP called the interface. Accumulator 1 holds the result.

As in most FORTRAN programming, it is the programmer's responsibility to ensure that the types of the arguments and the result are agreed upon between the called program and the calling program, given the transformation performed by the interface.

#### THE CALL-BY-NAME MECHANISM

A distinction is made in FORTRAN between call-by-name and call-by-value. In a call by name, the called subprogram is given access to the

location of the argument in question where it resides within the calling program; hence, if the called program changes the value of the argument, it changes its value within the calling program as well. In a call by value, on the other hand, the called subprogram is given access only to a copy of the value of the argument and cannot change its value within the calling program. PDP-10 FORTRAN SUBROUTINES and (contrary to the manual) FUNCTIONS operate on a call-by-name basis at present.

An analogous distinction may be made in LISP. LISP usually operates on a call-by-value basis, in that if a function (FOO ATOM) is executed, FOO receives the value of ATOM and cannot change the fact that ATOM is bound to that value. Only if (FOO (QUOTE ATOM)) is executed is FOO able to get at ATOM and change its binding.

The interface allows the LISP programmer the option of accessing the arguments of a called FORTRAN subprogram after the subprogram has been executed, thus creating the effect of a call by name. To do this, one of the following functions is executed in LISP:

(ARG1) (ARG2) (ARG3) (ARG4) ,

corresponding to the first through fourth arguments of the earlier calling sequence. Each function returns the current value of the argument, as it was left by the most recently called FORTRAN subprogram. (The interface properly converts the returned argument back to the LISP type that it had when the subprogram call was made.)

For example, if the called FORTRAN subprogram

FUNCTION ITEST (I,J,X,Y)

executed the statement

J = J + 1 ,

*and*  
~~as~~ if it were called from LISP as

(ITEST 3 JJ 4.0 NIL)

where JJ had been SETQ'ed to 5, this would be a call by value, and the value of JJ in LISP would be unchanged. But the form

```
. . . (ITEST 3 JJ 4.0 NIL)(SETQ JJ (ARG2)) . . .
```

would change the value of JJ in LISP to 6.

The call-by-name mechanism affords a convenient means for passing back to LISP information in addition to the function value, without bothering to create EXARRAY's or other mechanisms.

#### CALLING LISP FROM FORTRAN

At any point within the FORTRAN subprogram structure, a function call may be made back into LISP by invoking the following "FORTRAN function":

```
LISP(lisp-fn, arg1,arg2, . . . ,argn) .
```

This behaves as a regular function in FORTRAN, i.e., it can be coded into an arithmetic assignment statement and it returns a value.

The argument "lisp-fn" must convert, under CONVFL conversion, either to a pointer to the atom LISPFN or to a pointer to the executable compiled code for LISPFN, where LISPFN is the name of the function to be evaluated in LISP. Therefore, "lisp-fn" in FORTRAN must be either the pointer itself (in which case it must be given a logical or octal type) or the FORTRAN literal 'LISPFN' (in which case it must be given a literal, double precision, or complex type). In normal usage, of course, where it is merely desired to call a specific LISP function from a specific point in the FORTRAN structure, it suffices to code the literal constant (of up to 25 characters) directly into the calling sequence:

```
LISP('LISPFN',arg1,. . .).
```

The number of arguments for the lisp function, n, may range from zero through five. (The interface automatically determines the length

of the calling sequence, so only the single entry point "LISP" is needed.) The interface performs CONVFL conversion on the arguments, with their types as specified in the FORTRAN calling sequence. On return from LISP, the interface examines the result to determine its LISP type and performs CONVLF conversion on it.

The values of all of FORTRAN's accumulators are saved by the interface and restored to FORTRAN before returning. Whenever LISP is entered (either by a call or a return from FORTRAN), the values of LISP's accumulators are restored to those that were saved at the last exit from LISP. Thus, both LISP and FORTRAN see the necessary continuity of storage in their accumulators.

#### MISCELLANY

The interface, the FORTRAN functions, and DDT if desired, are loaded under LISP function (LOAD) by naming the files in which they reside. When the loader is terminated with the escape key, any called-for FORTRAN library functions are loaded. The status of the FORTRAN operating-system routines is unclear; in any case, they are not operative in the current arrangement because their UJO's conflict with those of LISP.

Other preliminaries amount to making all the xFORTn and ARGn entries of the interface available in LISP (through GETSYM), making certain LISP functions available to the interface (through PUTSYM), and establishing the dummy LISP functions and FORTREF.

The programmer must be aware that when control passes to LISP, list structures are subject to garbage collection under the rules of LISP. Thus, any pointers into LISP that are stored in FORTRAN across a time when LISP is entered must refer to structures that are protected from garbage collection.

The interface uses the LISP UJO "CALL" (see SAILON 28, Appendix 3) to enter a called LISP function. CALL accepts a pointer to an atom that has a SUBR or EXPR function definition; otherwise, it assumes the pointer is to compiled code. This means that other functional forms, such as

MACRO's, cannot be called directly this way. To get at these, one must shield the MACRO with a dummy function definition, or go through EVAL, or fetch pointers to the appropriate codes and use them directly, or perform some other trick. The result of calling a MACRO directly, or calling a named function that is not established in LISP, is generally to enter the atom header in LISP or in the interface and cause a crash.

Listings of the interface and other routines involved are available from the author.