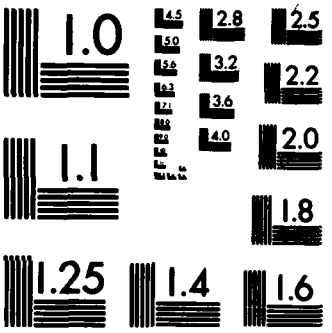
AD-A127 718 NOTES ON THE CONVERSION OF LOGLISP FROM RUTGERS/UCI-LISP TO INTERLISP(U) ROME AIR DEVELOPMENT CENTER GRIFFISS AFB NY R C SCHRAG JAN 83 RADC-TM-83-1 1/1

UNCLASSIFIED F/G 9/2 NL



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

RADC-TM-83-1
In-House Report
January 1983

NOTES ON THE CONVERSION OF LOGLISP FROM RUTGERS/UCI-LISP TO INTERLISP

Robert C. Schrag

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, NY 13441

This report has been reviewed by the RADC Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RADC-TM-83-1 has been reviewed and is approved for publication.

APPROVED:

DEAN F. BERGSTROM
Chief, Computer & S/W Engineering Branch
Command & Control Division

APPROVED:

JOHN J. MARCINIAK, Colonel, USAF
Chief, Command & Control Division

FOR THE COMMANDER:

JOHN P. HUSS
Acting Chief, Plans Office

If your address has changed or if you wish to be removed from the RADC mailing list, or if the addressee is no longer employed by your organization, please notify RADC (COES) Griffiss AFB NY 13441. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document requires that it be returned.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER: RADC-TM-83-1	2. GOVT ACCESSION NO.: AD-A127 718
	3. RECIPIENT'S CATALOG NUMBER:
4. TITLE (and Subtitle): NOTES ON THE CONVERSION OF LOGLISP FROM RUTGERS/UCI-LISP TO INTERLISP	5. TYPE OF REPORT & PERIOD COVERED: In-House Report
	6. PERFORMING ORG. REPORT NUMBER: N/A
7. AUTHOR(s): Robert C. Schrag	8. CONTRACT OR GRANT NUMBER(s): N/A
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Rome Air Development Center (COES) Griffiss AFB NY 13441	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: 62702F 55811912
11. CONTROLLING OFFICE NAME AND ADDRESS: Rome Air Development Center (COES) Griffiss AFB NY 13441	12. REPORT DATE: January 1983
	13. NUMBER OF PAGES: 16
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): Same	15. SECURITY CLASS. (of this report): UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE: N/A

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

Same

18. SUPPLEMENTARY NOTES

None

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

LogLisp	Artificial Intelligence Programming Languages
Lisp	Programming Language Translation
InterLisp	User's Guide
Rutger/UCI-Lisp	(logic programming in Lisp)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Conversion of the LogLisp Artificial Intelligence programming environment from its original Rutgers/UCI-Lisp (RUCI-Lisp) implementation to an InterLisp implementation is described. This report may be useful to researchers wishing to convert LogLisp to yet another Lisp dialect, or to those wishing to convert other RUCI-Lisp programs into InterLisp. It is also intended to help users of the InterLisp version of LogLisp to understand the implementation.

(Cont'd)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

The conversion process is described at a level aimed toward potential translators who might benefit from approaches taken and lessons learned. General issues of conversion of Lisp software between dialects are discussed, use of InterLisp's dialect translation package is described, and specific issues of non-mechanizable conversion are addressed. The latter include dialect differences in function definitions, arrays, integer arithmetic, i/o, interrupts, and macros. Subsequent validation, compilation, and efficiency enhancement of the InterLisp version are then described. A brief user's guide to the InterLisp version and points of contact for information on LogLisp software distribution are also provided.

Acknowledgements

Nort Fowler conceived and initiated the effort under which this work was done. Sharon Walter helped perform the conversion. Carl Engelman helped us with problems we had in using InterLisp. Phil Tarbell installed RUCI-Lisp on Tops-20 to allow us to validate our InterLisp implementation, and provided us with insight into machine-level execution. Darrel VanBuer provided the set of RUCI-Lisp-to-InterLisp transformations we used, as well as advice on compiler macros. Sanjai Narain utilized a preliminary, experimental InterLisp version of LogLisp, and provided useful bug reports. LogLisp's designers, Alan Robinson, Ernie Sibert, and Ken Bowen, have graciously helped us to understand the system software.

Thanks to all.

1. Introduction

This report documents the conversion of LogLisp (logic programming in Lisp), an Artificial Intelligence programming environment developed by Syracuse University under contract to Rome Air Development Center (RADC) [Robinson and Sibert 80, 81a, 81b], from its original Rutgers/UCI-Lisp (RUCI-Lisp) [LeFaivre 78] implementation to an InterLisp implementation. This work was performed as part of Lisp Implementation Baseline Investigation, an RADC in-house effort to experiment with and evaluate LogLisp. It served as an important preliminary to the effort's main task of evaluating the effectiveness of LogLisp for implementing a simple knowledge-based system, using its original InterLisp implementation as a baseline. Two versions of LogLisp were actually converted: LogLisp V1M1 delivered to RADC in December 1980, and LogLisp V2M3 delivered in December 1981. Version V2M3 has a more developed user interface and is more efficient than the previous delivery. References to LogLisp without a version number in this report pertain to the later version. The translation work was performed on the USC Engineering Computer Laboratory's System A (ECLA), a DEC PDP machine running Tenex and later Tops-20.

This report, which documents RADC in-house work, may be helpful to researchers who wish to convert LogLisp to still another dialect of Lisp, as many of the same problems--if not the same solutions--will be involved. We have tried to include enough information to point such individuals to the problem areas in the actual code. It may also be helpful to researchers who need to convert other programs from RUCI-Lisp to InterLisp. To this end we have tried not to be too verbose about the LogLisp code. It is further intended to provide a user of the InterLisp version of LogLisp with some insight into the implementation. However, it is not intended as program documentation to allow users to do their own bug fixes. Please report any bugs or perceived bugs to us, and we will try to respond quickly, and to keep other users informed.

The remainder of this report consists of six sections. In section 2 general issues of inter-dialect conversion are discussed and background on each dialect is provided. Section 3 describes our use of InterLisp's automated translation program. Section 4 describes non-mechanizable translation done by hand. Section 5 describes the validation of the translation, and subsequent compilation and efficiency enhancement. Section 6 provides a consolidated description of the differences between the InterLisp version and the original RUCI-Lisp version that could serve as a brief postscript to the LogLisp manual. Section 7 covers distribution of the LogLisp software.

2. Dialect Differences

The conversion of software from one dialect of Lisp to another involves many issues. Most Lisp's include a core of common functions corresponding roughly to those found in Lisp 1.5 [McCarthy et al. 62]. The exact syntax of these functions and their behavior in boundary cases (e.g. when one argument is NIL or zero) vary among the dialects. In addition to these more standard functions, each Lisp has its own peculiar functions which, though not absolutely necessary, implementors decided to include for reasons such as programming convenience or run-time efficiency. While there is less correspondence between dialects among the latter type of functions, some composition of functions in the target dialect will usually behave appropriately. Inevitably, some utility functions may correspond to operations which are not defined in the target dialect. In this case, it is necessary to determine what effects of the function must be reproduced in order to successfully convert the code.

2.1. InterLisp and RUCI-Lisp

InterLisp [Teitelman 78] is an extensive Lisp programming environment. It includes a sophisticated structure-oriented list editor, trace and break facilities for debugging, a comprehensive file package, a DWIM (Do What I Mean) facility for correcting certain kinds of errors in user input, a CLISP (Conversational Lisp) facility that allows some syntax similar to that of conventional programming languages such as Fortran or Algol, a facility for modifying functions by advising them of tests or actions to be performed before or after execution, and many other advanced facilities for Lisp programming. It represents one of the most sophisticated program development environments in existence.

RUCI-Lisp is an extended version of UCI-Lisp that runs under the Tops-10 operating system on the DEC-10 computer. UCI-Lisp (University of California, Irvine) [Bobrow et al. 76] is itself an extension of Lisp 1.6 [Quam and Diffie 76]. It includes the edit and break packages of 1972 InterLisp. (See [Teitelman 78] for background on the evolution of InterLisp.) To be fully informed, a user of the original implementation of LogLisp must have manuals for these three Lisp's besides the LogLisp manual.

3. Mechanical Conversion

While some of the problems that arise due to different techniques employed in the dialects' implementations can be interesting, the repetitive manual conversion between corresponding syntactic forms is a tiresome task. Fortunately, this process can be automated.

3.1. TRANSOR

One of the advanced programming facilities of InterLisp, called TRANSOR, is designed to assist in converting Lisp software from one dialect to another. It reads Lisp function definitions from a source file and, for each function for which they are provided, replaces its calls with translations resulting from the execution of transformations expressed as Lisp editor commands, writing the "transored" results onto a separate file.

The transformations can include "notes"--descriptions of the differences between the behaviors of the source function and the transored code and warnings of possible consequences. Or the "transformation" for a source function whose operation is undefined in the target dialect may simply be a note to that effect. Translation note output goes to another separate file on which the individual notes and the code portions to which they apply are cross-referenced.

We found a fairly complete set of RUCI-Lisp-to-InterLisp transformations at ECLA. Some trial and error was necessary to get the TRANSOR package to work properly. It failed unless the file of transformations was loaded before the TRANSOR package itself. We transored a file containing the source code for all of LogLisp. (The files containing the LogLisp system code are described in [Robinson and Sibert 80, 81b].)

3.2. Building an InterLisp File

The resulting transored code, like the RUCI-Lisp code, included the "'" read macro character, rather than QUOTE as a function call. While InterLisp allows the "'"-read macro in terminal input, it expands it immediately, and expects loaded files to contain the expanded form. Loading the transored code into InterLisp would not cause the macro to be expanded, or the forms in which it occurred to be correctly evaluated. To fix this problem, we made a temporary change to the read-table for files, setting the syntax of "'" to the same syntax it has in the read-table for terminals. Making the file (see below) made the code change permanent. (Alternatively, we could have set the syntax before transoring.)

Another problem appeared when we loaded the transored code into InterLisp. Two LogLisp functions had to be renamed because of clashes with InterLisp system functions. RESTORE is called RESTORR in our implementation, and FIND is called FYND.

Mechanical Conversion
Building an InterLisp File

We created an InterLisp file containing the transored functions, variables and properties, by constructing appropriate InterLisp file package commands. The most worrisome part of this task was to construct a list of names of the some 300 functions included in the LogLisp system code without having to type them all in by hand. The InterLisp function MAPATOMS, which "maps" over all known atoms, was useful for this purpose. We identified the function types in the transored file, and mapped over an empty InterLisp to isolate the InterLisp system functions of the same type. Then we loaded the transored file, mapped again over all known atoms for these function types, and took the difference of the two lists as our result. This resulting list also worked for saving the documentation (DOC) properties of functions that the designers provided. Other file package commands were small enough to be written by hand. We saved the file using MAKEFILE. Subsequent conversion was performed using this file and its future generations.

4. Non-mechanizable Translation and Manual Conversion

Examination of the translation notes generated by TRANSOR showed several areas in which problems remained. In some cases, it was only the need for knowledge of context which precluded the implementation of a syntactic correspondence in a transform. This was true for array references and arithmetic operations. It is conceivable that a translation program could learn about these contexts by remembering code it has already seen, or by being told, but TRANSOR does not have this capability. In other cases, while direct syntactic correspondences existed, no transformations had been written to cover them. This was true for many input/output (i/o) functions and atom manipulating functions. Finally there were cases where the operation of the function was not defined in InterLisp. This was true for some interrupt and error-handling functions and for those functions designers implemented as macros.

Attempting to run the code revealed calls to RUCI-Lisp functions which had slipped through TRANSOR because neither translations nor notes for them were included in the set of transformations we used. This happened for a variety of functions, and we usually dealt with it by defining InterLisp functions with the same names and including them with the code for our implementation.

Finally, there were problems resulting from assumptions in the LogLisp code about issues which are implementation specific. This was true for code which accesses stored function definitions.

We will examine the last problem first, as the storage issue is relevant to other problems, including array and macro expressions. Except when logic dictates other sequencing, we will examine problems and describe our approaches to them in the order in which they were encountered.

4.1. Atoms, Functions, and Property Lists

An atom is the Lisp identity with which objects such as values, function definitions, property lists, and print names are associated, and by which they are accessed. In InterLisp, separate memory cells are associated with each of these objects, and, except for print names, primitive functions are provided with which they may be accessed. In RUCI-Lisp, only the property list is directly associated with an atom. Values, function definitions, and print names are stored under and accessed via appropriate properties on the atom's property list.

In some of the functions (#REDUC1, #VALRED, and #SETOF) comprising LogLisp's reduction mechanism, there is code which tests whether an atom has a function definition, to determine its semantic class in a LogLisp expression. The RUCI-Lisp code checks whether an atom has a property among the function types defined in RUCI-Lisp. We translated this to InterLisp code of the form (FNTYP FOO), which returns the type of the function definition associated with the atom FOO, or NIL if FOO is not a function name.

We are glossing over a subtlety which the direction of the translation allowed us to ignore. In RUCI-Lisp, the result of predicate evaluation is usually T or NIL, while in InterLisp, most (non-numeric) predicates either return NIL or one of their arguments. Consistent with these interpretations, the InterLisp COND form accepts test arguments with non-logical values, while RUCI-Lisp requires that the test argument evaluate to T or NIL. We can therefore ignore that our InterLisp translation branches on non-logical values, as returned by FNTYP.

An unusual sort of bug was born when we neglected to include the compiled function type CFEXPR* among those checked for. While the code still seemed to work in interpreted form (since we weren't using any CFEXPR*'s in the place of LogLisp procedure calls), as soon as we compiled it, the LogLisp forms ANY and LISP ceased to work. Tracing execution in such situations gave no indication of the offending functions. Because broken functions become embedded in interpreted code and CFEXPR*'s become FEXPR*'s, traced code worked properly. In retrospect this was a fairly glaring error of omission.

A related problem is that InterLisp expects any atom appearing at the head of an unquoted list to have a function definition, while RUCI-Lisp will allow an atom that is merely bound to a value that is a function definition. In InterLisp, this sort of construct requires the use of APPLY. Calls to APPLY were inserted in #VALRED, #REDUC1, and PRINTFACTS.

4.2. Arrays

InterLisp treats arrays as data, while RUCI-Lisp treats them like functions, placing an array referencing function on the atom's property list. In InterLisp an array FOO is created by binding FOO to an array pointer as in (SETQ FOO (ARRAY N P V)), where N is used to indicate size, P to determine integer versus s-expression entries, and V to initialize entries. Array bounds always run from 1 to N. Array elements of index I are referenced by (ELT FOO I), and set to a value V by (SETA FOO I V). In RUCI-Lisp such an array is created by (ARRAY FOO P (1 . N)), where P determines entry type and initialization, and the dotted pair indicates the subscript range. Array elements of index I are referenced by (FOO I), and set to a value V by (STORE (FOO I) V).

The LogLisp system uses four arrays: #HPWT and #HPCLS used in heap management, and #ENVS and #ENVX used in environment management. We made the appropriate changes in array-creating code for entry type and initialization, and converted the subscript range information to size information. RUCI-Lisp STORE expressions were easily converted to InterLisp SETA expressions, but TRANSOR ignored RUCI-Lisp array reference expressions as unidentified function calls. We changed these to ELT expressions.

This was all that was necessary for #HPWT and #HPCLS, because they are subscripted from 1 upwards. However, #ENVX is subscripted from 0

and #ENVS is subscripted from -1. We needed to preserve these subscript ranges in order to preserve the interface with other environment management code, so our translation adds 1 to the subscript for #ENVX whenever it is referenced or set, and adds 2 to #ENVS's subscripts.

4.3. Integer Arithmetic

The only arithmetic in LogLisp operates on integer expressions. Because RUCI-Lisp has only general arithmetic operators, TRANSOR was forced to make the conservative translation to general arithmetic operators in InterLisp. We used a text editor to replace the general arithmetic functions with more efficient integer arithmetic functions. The differences in function name length caused InterLisp to rewrite the file map when the file was loaded, but that is of no major consequence.

A related problem is that the RUCI-Lisp code showed all integer constants with decimal points to distinguish them from octal (the default). At first we just left the decimal points in, but that didn't work, because a decimal point signifies the floating point data type in InterLisp, and some parts of the code test these constants against valid integers which will never be EQ. We used a text editor to remove the decimal points.

4.4. I/O and Atom Manipulation

I/o is always implementation and usually operating system dependent, and it is tedious to translate (which is probably why no transformations were written for the functions in this section). The only code we actually changed was that used to access files in the functions that save and restore logic programming knowledge bases (SAVE and LOADLOGIC). For the terminal i/o we defined InterLisp functions (PRINA, PRINAC, PRINL, and MSG) that seem to behave properly and included them with the code for our implementation. We used the same approach for the atom manipulating functions AEXPLODE, AEXPLODEC, ANTHCHAR, and MAKNAM. The interested reader can consult the code for the details of the translations.

There is one terminal i/o function whose operation is worthy of note. The RUCI-Lisp function LINEREAD was converted by TRANSOR to the InterLisp function READLINE. The functions indeed have compatible input and output specifications, reading in an arbitrary line of text and returning it as a list, but READLINE is very impatient, returning NIL if the input buffer is empty when it is called. This is suitable in LOADLOGIC, which reads from a file, but not in FACTS, MONITOR, or #ASK, where terminal input is required. We took advantage of InterLisp's advice facility to advise the latter functions to call the function WAITFORINPUT before executing. WAITFORINPUT simply does nothing until the input buffer is ready. We included file package commands for this advice among those for our InterLisp file, so that it is saved from generation to generation.

4.5. Error/Interrupt Handling Functions

RUCI-Lisp includes the function INITFN which establishes its argument as the function to be evaluated whenever an interrupt to return to top level is encountered. This serves two purposes in LogLisp. During search the function #INITSEARCH resets two global variables to their proper pre-search values. During editing, the function #EDITTRAP prevents abnormal exit from the editing routine, since assertions are erased before editing and reasserted afterwards.

Unfortunately, INITFN is one of those functions whose operation is not defined in InterLisp. We were able to simulate its behavior in LogLisp by defining our own INITFN. It uses the function INTERRUPTCHAR to cause control-D and control-E, which normally call the function RESET, to call INITFN's functional argument instead. We then modified #INITSEARCH, but not #EDITTRAP, to call RESET.

4.6. Macros

In Lisp, in general, macros are like function definitions and have the same status, except that instead of consisting of code which is simply evaluated, the macro code manipulates the calling s-expression, perhaps recursively, with the resulting "macro-expansion" being evaluated in place of the original s-expression. Macros are thus doubly evaluated, first for expansion, then for effect.

While this is essentially the way macros are implemented in RUCI-Lisp, InterLisp deviates from the Lisp norm in this regard. In InterLisp, macros do not have function status and are only legitimately available for use by the compiler. The RUCI-Lisp-to-InterLisp transformations we used with the TRANSOR package translated macros into fexprs that were rather ugly. We found that once we understood the macros' intents we were able to write relatively straightforward fexprs to do the same jobs.

InterLisp compiler macros reside on an atom's property list under the property MACRO. While the normal function access mechanism will not find them there, InterLisp's designers have provided a package called MACROTRAN which interfaces with the DWIM facility. Whenever an undefined function is encountered, MACROTRAN is used to look up the atom's MACRO property and perform the expansion, with the expanded code then being evaluated. If no MACRO property exists, an error results. The MACROTRAN facility means that macros don't have to be converted to fexprs, but we did convert them for use with the version of our InterLisp translation of LogLisp that we run interpreted, in the belief that this would be more efficient.

When Lisp programs are compiled, the inclusion of macros rather than fexprs generally leads to greater run-time efficiency, because expansion eliminates the need for a function call. We translated the RUCI-Lisp macros to InterLisp compiler macros for the compiled version

of our InterLisp translation of LogLisp. InterLisp includes three kinds of macros, one of which (the "computed" macro) is similar to RUCI-Lisp's. Rather than overwork this correspondence, though, we took advantage of the fact that some constructs (the PMACRO and SUBST constructs) used in macros by LogLisp's designers produce the same effects as produced by InterLisp's "substitution" macro. Although the code generated is the same, the expansion is more efficient than if we had used computed macros in these cases.

4.7. Miscellaneous Transformless Functions

The RUCI-Lisp functions INCR and DECR, which increment and decrement integer arguments, respectively, were implemented as macros, as they are in RUCI-Lisp.

The RUCI-Lisp functions INITSYM and NEWSYM, which are used to generate subscripted atoms, were implemented as fexprs, with an atom's subscript counter being stored under the SCNTR property on its property list, as in RUCI-Lisp.

RUCI-Lisp includes a macro called DO, which provides a variety of iterative forms. Types of calls to DO in LogLisp include DO WHILE, DO UNTIL, DO FOR ... IN, and DO FOR ... ON. For each of these calls to DO (though not for all DO forms), there is an equivalent InterLisp CLISP form that behaves identically and is identically constructed except for the omission of the atom DO.

5. Testing and Tuning the InterLisp Implementation

During the process of solving the problems outlined in the previous section, we were continually referring to manuals, and listings of the original and partially converted code. Functions in InterLisp files are always listed alphabetically, but the RUCI-Lisp file we transored from was not. We numbered functions on the RUCI-Lisp file and cross-referenced this with the InterLisp file, for convenience. A better approach may have been to save the RUCI-Lisp function definitions as properties, making them accessible on-line.

After the obvious errors (those causing breaks) were eliminated, we tested and debugged our translation by comparing results obtained in it to those obtained for the same computation in the RUCI-Lisp version. When we were satisfied with the faithfulness of the translation, we commenced compiling as efficient a version of it as we could.

5.1. Validation

When we needed a reference point for our V1M1 translation, we used a version of RUCI-Lisp which runs under Tenex. Later, when we did the V2M3 translation and ECLA had changed its operating system to Tops-20, we used an implementation of LogLisp in RUCI-Lisp described in [Tarbell 83]. We validated the translations and their compiled versions with an example knowledge base called Places (an intelligent data base system containing world geographic, economic, and political information) and an associated variety of prepared queries from the software delivered by Syracuse University. Use of this knowledge base required the translation of ancillary files containing Lisp functions which are called by its procedures. When a version was capable of executing all of the prepared queries and gave the same results as the corresponding RUCI-Lisp version, we tentatively accepted it as valid.

5.2. Compilation

When we believed we had achieved a valid translation, we compiled it using TCOMPL, the standard InterLisp function for file compilation. At first we had no version of the translation which included compiler macros, and the efficiency of the compiled version was limited. LogLisp's designers have included a global variable in LogLisp V2M3 that serves as a counter for the number of resolvents generated. We took advantage of this counter and used InterLisp's TIME routine to measure the efficiency of various versions of our InterLisp translation. We chose a query requiring only logical inference (no Lisp reduction), used with a relatively small LogLisp knowledge base, as our standard for measurement. The first pass at compilation produced a version which performed about 16 resolutions per cpu second.

At this point we decided to implement RUCI-Lisp macros as InterLisp compiler macros rather than fexprs. When the resulting file was compiled, performance improved to about 34 resolutions per cpu second, about twice as fast as the fexpr-only version.

Testing and Tuning the InterLisp Implementation
Compilation

In compiled Lisp functions, references to variable names do not generally appear as they do in interpreted functions. Appearing instead are references to fast memory locations (accumulators) containing the values which they represent. These references are in effect local--when when another function is called from within the defining function, the accumulator contents are pushed onto a stack, in order that the same registers can be used by the function being entered. This stack implementation means that the variables of a given compiled function will not be accessible to other functions it calls (directly or indirectly) which reference them by name, as they would be if the calling function were not compiled.

Most dialects of Lisp provide a mechanism to allow variables which are bound in a compiled function to be used free in a function called by that compiled function. When variables are used in this way, they are called "special" variables. Variables declared special are treated by the compiler differently than non-special or "local" variables. Their names do appear in the compiled code, and refer to memory locations which are not local (contents don't change when a new function call is entered). When a variable in a compiled function is made special, it is accessible to called functions in which it occurs free, but efficiency of access from within the binding function is then degraded compared to efficiency of access to local variables. For this reason variables are usually assumed local unless otherwise declared.

Special variables in RUCI-Lisp are handled essentially as described above. The InterLisp compiler deviates from the norm by treating all variables as special, to the end of allowing interpreted and compiled functions to be freely intermixed. This is a default which can be overridden.

Since the special variable declarations were already available to us (in the original RUCI-Lisp code) it was a simple matter to create file package commands that would assure only these variables be treated special when the file was compiled. The resulting compiled version performed 53 resolutions per cpu second when tested with our standard query, or approximately half again as fast as without local variables. This is the version that we currently use, since we feel that those described here are the only efficiency enhancements we could make while faithfully preserving the intentions of LogLisp's designers.

We performed similar efficiency analyses running the code in interpreted form, to compare the effect on performance of having InterLisp's error handling machinery call MACRUTRAN, as cpposed to using code which simply included fexprs. Contrary to expectations, the timing routine indicated that the macro-containing code was about 10 per cent faster than the fexpr-containing version. (Both performed about 4 resolutions per cpu second.) We were unable to determine the source of this anomaly. It might be due to some overhead associated with fexprs which macros do not incur, or to some unknown feature in InterLisp's timing routine which subtracts out the time required for macro expansion.

We finally decided that maintaining fexpr-containing files consistent with the macro-containing files was more trouble than it was worth, and deleted them. The differences in elapsed time for query execution between the two interpreted versions did not seem to be great, and in practice (now that the compiled version handles CFEXPR*'s properly) we only run LogLisp interpreted when making changes.

In comparison with our InterLisp version, Syracuse University's LogLisp V2M3 running in the Tops-20 RUCI-Lisp installation described in [Tarbell 83] performs about 12 resolutions per cpu second interpreted, and about 230 resolutions per second compiled, when tested with our standard query. The InterLisp versions of LogLisp are somewhat more efficient than would be indicated by the folklore that RUCI-Lisp is six times faster than InterLisp.

6. Users' Guide

This section describes the differences between the original RUCI-Lisp version of LogLisp and our InterLisp translation. One difference has already been mentioned. Because of clashes with InterLisp system variables, RESTORE has been renamed RESTORR and FIND has been renamed FYND.

The interactive documentation facility described in [Robinson and Sibert 80, 81a] has been all but done away with because of dialect incompatibility. We have included only the function DOC, which returns the DOC property of its argument.

After we had made our translated implementation available for initial experimentation, an incompatibility between the InterLisp and RUCI-Lisp versions appeared. A user trying to utilize a knowledge base he had developed with the RUCI-Lisp version complained that it caused an error when loaded in the InterLisp version. Examination showed that the offending clause included an argument with the string data type. Strings are not considered atoms in InterLisp as they are in RUCI-Lisp.

We first decided to accommodate the string data type by modifying the LogLisp code to call a predicate which accepted either an atom or a string wherever a call to ATOM appeared in the original code. We regarded this to be in the spirit of [Robinson and Sibert 81a], in which atoms are identifiers, strings, or numerals (pp 3,4). We have since reconsidered this decision, as the phrase in the LogLisp manual was meant to apply both to the LogLisp code and to RUCI-Lisp, and while we could make it apply to our InterLisp translation of the LogLisp code, we could not make it apply to InterLisp. We believe that most usages of strings in LogLisp knowledge bases can be replaced by the appropriate usage of identifiers bound to strings. Dissatisfied users can complain, or make the substitution of an atom-or-string predicate for the ATOM predicate themselves.

7. Distribution

For information on obtaining a copy of the InterLisp version of LogLisp, or to report bugs in that version, contact:

Robert C. Schrag
Rome Air Development Center
Software Sciences Section (RADC/COES)
Griffiss AFB, NY 13441
(315) 330-2748 [Autovon 587].

For information on LogLisp running in RUCI-Lisp under Tops-10, contact:

Dr. J. Alan Robinson
Logic Programming Research Group
Link Hall, Syracuse Universtiy
Syracuse, NY 13210.

Future development of LogLisp by Syracuse University will be done on Lisp machines manufactured by Lisp Machine, Inc.

For information on LogLisp running in RUCI-Lisp under Tops-20, contact:

Capt. Phillip B. Tarbell
Rome Air Development Center
Software Sciences Section (RADC/COES)
Griffiss AFB, NY 13441.

References

[Bobrow et al. 76] R.J. Bobrow, R.R. Burton, J.M. Jacobs, and D. Lewis. UCI-Lisp Manual, on-line documentation at ECLA, 1976.

[LeFaivre 78] R. LeFaivre. Rutgers/UCI-Lisp Manual, on-line documentation at ECLA, 1978.

[McCarthy et al. 62] J. McCarthy, P.W. Abrahams, D.J. Edwards, T.P. Hart, and M.I. Levin. Lisp 1.5 Programmer's Manual, The MIT Press, 1962.

[Quam and Diffie 76] L.H. Quam and W. Diffie. Stanford Lisp 1.6 Manual, on-line documentation at ECLA, 1976.

[Robinson and Sibert 80] J.A. Robinson and E.E. Sibert. Logic Programming in Lisp, RADC Technical Report 80-379, volume 1, 1980.

[Robinson and Sibert 81a] J.A. Robinson and E.E. Sibert. The LogLisp User's Manual, unpublished interim technical report, 1981.

[Robinson and Sibert 81a] J.A. Robinson and E.E. Sibert. LogLisp Implementation Notes, unpublished interim technical report, 1981.

[Tarbell 83] P.B. Tarbell. "Notes on the Installation of Rutgers/UCI-Lisp on Tops-20," RADC Technical Memorandum, (in preparation).

MISSION
of
Rome Air Development Center

RADC plans and executes research, development, test and selected acquisition programs in support of Command, Control Communications and Intelligence (C^3I) activities. Technical and engineering support within areas of technical competence is provided to ESD Program Offices (POs) and other ESD elements. The principal technical mission areas are communications, electromagnetic guidance and control, surveillance of ground and aerospace objects, intelligence data collection and handling, information system technology, ionospheric propagation, solid state sciences, microwave physics and electronic reliability, maintainability and compatibility.

END

FILMED

6-83

DTIC