REPORT ON THE IMPLEMENTATION OF LISP FOR MULTICS: SUMMARY OF THE RESEARCH CONDUCTED IN CONJUNCTION WITH COURSE 6.681

Adviser: Prof. R. M. Graham
Student: John Linderman

The goal of this project was the development of a general strategy for the implementation of the LISP programming language under the MULTICS operating system.

I felt that one reasonable approach to the project was to

(1) Identify what constitutes a "LISP system", independent of machine; then

(2) Determine what impact the peculiarities of MULTICS would have on this ideal LISP system.

To this end, I first studied the formal LISP language, extending the introduction to the language I had received earlier in the term via course 6.539. When I felt that I understood the essentials of the language per se, I turned to the study of several operational LISP systems as described in "The Programming Language LISP: Its Operation and Applications" [1].

My introduction to MULTICS was through a series of FJCC reprints,[2] the "ACM Symposium on Operating System Principles"[3] and the first three chapters of a guide for MULTICS subsystem writers.[4] Before detailed implementation can be initiated, further research into the MULTICS system and the GE 645 will be required. However, I suspect that the "spirit of MULTICS" is well represented in the above references, and it is this spirit that should most influence the development of a general strategy.

The following is a report of my conclusions about the effect of the MULTICS system [which must be considered untouchable] on the implementation of a LISP system [which can, and really must, be tailored to suit our needs.]

"LISP", of course, is an acronym for list processing. It should come as no surprise, then, that fundamental to any LISP system is the structure called a list [or, more generally, an S-expression.] A list is actually a tree of substructures which I shall refer to as LISP cells. The basic property of a LISP cell is its ability to identify two other such cells, the so-called CAR and CDR of the cell. In theory, any configuration of memory registers which enables this pointing could lead to an operative LISP system. In practice, LISP will be devoting much of its effort to the construction and processing of lists, so it is important that the configuration be, in some sense, "natural" for the machine with which we are dealing.

While the choice of internal representation for LISP cells will be vital importance in any detailed implementation, I would like to put off the decision, at least for the time being. I do this (1) because most of the interesting problems in the implementation arise quite independently of the decision, and (2) since my exposure to GE 645 machine language has been limited to the subsystem writer's guide, I doubt my qualifications to make a good choice. I will assume in the following, then, that some adequate choice has been made.

The virtual memory feature of MULTICS appears at first to be a big bonus. In most LISP systems, a fixed supply of LISP cells are initially available. If and when this supply is

exhausted, processing must be suspended. A reclaimer or "garbage collector" is called to restock the supply with cells that are no longer being used. If our representation of LISP cells enables us to specify segment as well as word number, MULTICS provides us with an effectively limitless supply of cells. Perhaps we can do without the garbage collector, the writing of which is usually a major part of the implementation process.

Unfortunately, this is too good to be true. Paging, the mechanism which lets us pretend we have unlimited space, introduces some new problems. To see this, suppose we have a "cell manager" whose job it is to honor all requests for LISP cells. The manager presumably supplies cells by preparing contiguous areas from a data segment. Over a period of time, the manager is likely to receive a sequence of requests like

```
40 cells to put a labeled function on the argument list.
30 cells for some new atomic symbols just read.
 6 cells for bindings on the argument list.
 1 cell for a CONS
 8 more cells for bindings on the argument list.
...
```

The consequence of this manner of cell distribution is that cells in a single S-expression may lie in a multitude of pages. While this may present no difficulties in theory, it may considerably degrade performance. As a list is processed sequentially, missing pages must be retrieved from secondary storage. This is a relatively slow operation and will become particularly painful if the retrieved pages must displace other pages to be referenced.

One hope for reducing this loss of efficiency is a process called "unraveling". Basically, unraveling an S-expression consists of copying it so that the LISP cells in the expression occupy a minimal number of pages. Future passes through such compacted S-expressions should thus be more efficient. The basic simplicity of this process belies a number of difficulties. One problem is "What do we unravel?" If we unravel an S-expression that is contained in a larger S-expression, we are wasting time. Assuming we eventually get around to unraveling the larger expression, we must recopy the subexpression. We are also wasting time if we unravel an S-expression which will not be used again or will be used only once. Unraveling is most profitable when applied to lists constucted over a long period of time and subject to frequent scanning. The argument list in an interpreted system is such a list. It is relied on so heavily that serious consideration should be given to supplying it with a storage manager and data segment of its own. Allowing the user to identify such lists could aid the unraveler in choosing wisely.

Another problem is "When do we unravel?" The timing is not so obvious as it was with a garbage collector. If we unravel too often, we will get no more processing done than we would if we suffered the missing page delays. One intrigueing possibility opened up by MULTICS is the parallel operation of LISP and the unraveler. This would mean that some care would be necessary to assure that LISP did not

alter the structure of a partially unraveled S-expression. If we have a few unused bits in our representation of LISP cells, this could be accomplished by setting flags in partially compacted expressions.

The actual processing of lists can take place in either or both of two modes. Processing can be directed by functions which are themselves S-expressions, interpreted by the LISP system. When a function is invoked, its variables are paired with their current values on a large list called the argument list or A-list. Evaluation involves searching this list to find the most recent "binding" of the variables involved. Interpretation and A-list searching, while they are wonderfully general, are quite slow. It is also possible to translate functions from their S-expression form to machine language subroutines. Variables "local" to such functions are assigned to specific locations so the A-list need not be searched. This, combined with the direct execution of functions, considerably hastens processing.

If one is going to have a compiler in the LISP system, it is certainly tempting to observe MULTICS standards for translators. Doing so facilitates such advantages to the user as the development of a LISP library of functions. The implementer also profits. Not only are pure procedures "automatically" recursive, the use of the process stack relieves the LISP storage manager of the burden that the pushdown list represents in common implementations. By preserving standard linkage conventions, we may even call procedures written in other

languages. If this is to be a useful ability, we may wish to choose the internal representation of a LISP cell so that it corresponds to a valid argument for more conventional translators. Since it is the function of the LISP cell to "point", this might lead us to represent cells as pairs of PL/I-style absolute pointers [ITS pairs].

Implementing an interpreter along with the compiler adds considerably to the complication. In a mixed interpreter/ compiler system, we cannot assume that every function is compiled. This means that compiling the standard CALL/RETURN sequence is not adequate. Some means of determining the nature of a function [such as checking the property list of the atom which is the function name] is necessary to decide the mode of execution, and an interface between the compiled routines and the interpreter must be supplied.

There are some reasons for putting up with this extra work. For example, without the interpreter, it is not clear how to enable a computation to modify the definition of a function. Whether such bizarre uses of LISP are worth defending may be questionable, but any reduction of generality in the system is sure to cause dissatisfaction among LISP "purists".

In view of the version of LISP proposed and the strategies employed by others implementing LISP systems, I can envisage the following plan for implementation.

1) Decide, using considerations of time, space, compatibility with other MULTICS systems, etc., the exact representation of LISP cells. The ability to flag cells in some manner is very desirable. In fact, if cells do not include a few bits whose use we do not currently anticipate, we may be paying a high price in later flexibility for economy of storage. Conventions on the identification of atoms, representation of numerical atoms, atomic property lists and such like can then be established.

2) Design the garbage collector/unraveler. The problem of what to unravel will depend on other phases of the implementation, but the operation should largely defined. This phase will provide us with a check on the adequacy of our choice for cell structure.

Special consideration in the design should go toward making the unraveler able to act in parallel with LISP, even if this cannot currently be realized under MULTICS.

3) Write the basic system fuctions such as CAR, CDR, EQ etc. Because these functions are basic, they should be coded in machine language.

4) Any time after LISP cell conventions are set, the read and print routines can be written. Knowing very little about I/O under MULTICS, my only suggestion is

that these [and all other] routines be written in higher level languages to whatever extent possible, calling basic machine language subroutines when detail demands.

5) Write the compiler. The common approach to this job is to write a compiler in LISP which outputs an intermediate language to be translated into machine language. A translator must then be written, but LISP can again be used for much of the job. The compiler can then "compile itself" on an operational system. The output is translated to get a compiler for the new system.

The prime concern of our compiler should be the ultimate generation of pure procedures observing MULTICS standards. The nature of the intermediate language will no doubt reflect this. It is conceivable that the compiler could output a higher level language instead of special S-expressions. Unless our LISP structures can be conveniently manipulated within the higher language, this level of generality may be hard to justify.

6) Write the interpreter and the interface between the compiler and the interpreter. The interpreter could be compiled, but efficiency is a worthwhile goal here, so careful coding may be warrented.

7) Write the "overlord" to put all the components together.

Bibliography

1. Information International, Inc.: The Programming Language LISP: Its Operation and Applications, MIT Press

2. General Electric Company reprints of FJCC, Nov 30, 1965: A New Remote-Accessed Man-Machine System

3. MAC: ACM Symposium on Operating System Principles, Association for Computing Machinery.

4. Elliot Organick: A Guide to MULTICS for Subsystem Writers, MAC.

5. Computation Center and RLE: LISP 1.5 Programmer's Manual, MIT Press.