STANFORD ARTIFICIAL INTELLIGENCE PROJECT
Memo No.10

December 18, 1963

IMPROVEMENTS IN LISP DEBUGGING

by S. R. Russell

Abstract: Experience with writing large LISP programs and helping students learning LISP suggests that spectacular improvements can be made in this area. These improvements are partly an elimination of sloppy coding in LISP 1.5, but mostly an elaboration of DEFINE, the push down list backtrace, and the current tracing facility. Experience suggests that these improvements would reduce the number of computer runs to debug a program by a third to a half.

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

Introduction

The ability to write and debug large programs quickly is a very valuable one. We are still in the unfortunate situation that a well-informed programmer can design in a few weeks a program that cannot be written in less than several months or debugged in less than several years. LISP can claim virtues in the actual programming, but in debugging it has no great virtue. It is to more effective debugging that these proposals are addressed.

The two effects I hope to achieve are:-

1. Protecting the student and casual user from needless mystery when it is not too hard to do so.

2. Giving everyone, especially the expert with a large program, the relevant information about his bugs, producing as few irrelevancies as possible.

In both these respects current versions of LISP are sadly lacking.

These features are worth a noticeable time sacrifice in the running speed of programs although they do not necessarily demand it, as debugging of LISP itself and LISP programs have consumed far more time than all the "production" runs of LISP. There is no indication that this situation will change in the near future.

1. Changes in error recognition

Very few people who have debugged a LISP program have escaped from the mysteries of LISP's subconscious. It takes only a small bug to take car or cdr of an atom and this is sufficient to introduce invalid list structure.

Until quite recently, I, as one of the proprietors of the code that permitted this easy access to LISP's subconscious, maintained that this was the way it should be. Reflection, however, has shown that it would have taken less of my time to have prevented easy entry into the subconscious originally than it has subsequently taken to explain away its appearance to baffled users.

Taking car or cdr of an atom should give an explicit error message. This is common occurrence, but currently it merely gives strange list structure and strange results which needlessly compound the mystery.

To access property lists special functions must exist, but these can be either a special version of cdr or some more elaborate functions such as FLAG or PROP which give more freedom of arranging and rearranging atom storage.

The cost in running time of recognizing this error will vary with storage conventions. When it is high, an artifact may be added to use unprotected versions of car and cdr for speed in debugged programs.

For no good reason calling a function with the wrong number of arguments is not flagged in a lucid way. This, too, is a common occurrence and should be checked explicitly by the functions that actually do the calling, or by define.

2. Backtracing

In LISP 1.5 the occurrence of an error causes a "backtrace" or listings of all the functions that have blocks saved on the push down list. In compiled program this is quite informative, but the interpreter, because of some clever coding, leaves a track of "CONS COND MAPLIST CONS COND MAPLIST....." which is not very helpful.

A major improvement in backtracing would be to print the arguments to each function in the back trace. This will pinpoint the immediate cause of the error, and also detail the cause of the cause, etc., saving much effort and frequently a rerun with tracing.

The arguments of compiled functions are readily available if stored on the push down list and can be found if stored elsewhere, but a small agreement between the interpreter and the backtracer is necessary to make the interpreter's tracks meaningful. It is also necessary to put every function call on the push down list, rather than just the truly recursive ones.

3. Tracing

The current TRACE facility is effective, but tends to wallpaper. A worthwhile addition is Minskey's "alarm clock". A count of function calls or some such thing is kept and printed when an error occurs at the end of a function.

If tracing is desired at some point well along in a function the value of this counter when tracing should start is given, and no tracing is done until that point. This will cause considerable reduction in paper generated.

Dean Wooldridge has patched this into LISP 1.5 to help with obscure errors in his simplify program, and it has been quite useful.

4. Diagnostic Define

At least for checking purposes, there should be a version of DEFINE or an M-expression to S-expression translator that checks function calls for proper number of arguments, undefined functions makes lists of bound, unbound, and program variables, checks for proper list format, and generally checks for as many of the common errors as is possible. Useful byproducts of such checking are lists of functions and variables used.

At the last LISP 2 conference I was rather loudly opposed to the idea of a large define program. My mind was changed by reading the ALGOL translater for the B5000. It is well commented and neatly paragrapned,f and I found I could read it almost like a novel. Students using this translator seemed to benefit from the relatively good error checking it provides. I now feel that the gain from such an approach is worth considerable effort.

5. Notes on debugging a version of LISP

The above detailed debugging aids especially atom-proof car and cdr will of course be of service to the person writing a new version of LISP, but there are a few more that would save untold hours of decoding dumps.

A printout of the list structure storage area, that puts out actual locations of list structure. It must mark as it prints to avoid printing the same thing more than once. I have no definite ideas as to what the appearance of this should be, but it seems that its output should be no more voluminous than an ordinary dump.

A printout of storage assignments whenever they are set would be most helpful. I am unduly biased by the last few runs I had debugging LISP but it seems that I have spent half my time decoding octal dumps to get this information.