ソフトウエア 工学 9 - 3
(1979. 3. 29)

Lisp Debugging Tools

Toshiaki Kurokawa

Information Systems Lab., TOSHIBA R & D Center

ABSTRACT

In this paper discussed are debugging tools in the programming language Lisp. First, debugging process is reviewed and concluded that it is indispensable for the process of software production and that sharp debugging tools are very valuable. Next, Lisp is considered to be a programming system where those tools are incorporated. Lisp1.5, Interlisp, and Lisp1.9 are reviewed to see what kinds of tools are utilized today and what will be developed in the future. These Lisp experiences are useful to design other programming systems and to develop more useful software production systems.

§1 Introduction

Debugging is a popular activity in programming, but Lisp is a less popular language. It is natural that one suspects if the theme 'Lisp debugging tool' is of any use at all. Eventually, the theme is useful for everyone involved in programming because here Lisp should be a programming system where a programmer can expect some useful tools for writing, debugging, and editing his programs.

The concept of the programming system has gained its position in the software engineering world (see Wada /25/). Programming tools and systems are important subjects for today's software engineers (see Brooks /2/).

Lisp debugging tools are good examples to review the (debugging) tools in the programming system. It tells you how to recognize, develop, and utilize the programming tools and systems.

In the followings, §2 reviews debugging process. §3 explains the programming system aspects of Lisp, especially why Lisp has become such a system. A relevant-talk is found in Sandewall /20/. §4 presents the actual debugging tools taken from Lisp1.5/13/, Interlisp /24/ and Lisp1.9 /9/. §5 gives the account of the future directions of the debugging tools.

§6 concludes that we have climbed to the stage where a useful highly productive software developping system can be constructed. Lisp programming system including its debugging tools is a prototype for such an advanced system.

§2 Debugging Reconsidered

Debugging is a well known process for programmers, but it is so strange to non-programmers that they ask "Why programmers are making so many bugs? They must be too careless in their job!"

Well, from the psychological viewpoint (see page 161 in /17/), the number of bugs is

too large to be considered to be caused by human carelessness.

The principal reason is that the tools and procedures with which we construct programs are incomplete. Although Dijkstra needed not test his program made by his new method /4/, it is not yet available for the practical use.

Another reason is that a bug is defined as an inconsistency between the actual behavior of the program and the expected one. It is usual that the programmer's expectation is not complete so that he found the true expectation only after the program is made.

Well then, is it possible to forget debugging process when we overcome the above difficulties? The answer is regretfully 'No'. The first reason is that we cannot expect a programmer to be perfectly careful. Careless bugs cannot be exterminated. The second reason is that the program is a product which a variety of persons will use. So a user's expectation may be different from the originator's. Strictly speaking, this process is a modification, but it can be included in the debugging process, for the essential property is the same.

The last reason is that when we consider a software production process such as in Fig. 1, the checking process assuring the bug-free status is a component of the debugging process.

So it is concluded that the debugging process will last forever and that the debugging tools are indispensable. It is important to create usful debugging tools, to use them in a systematic way, and to make them to be our common property.

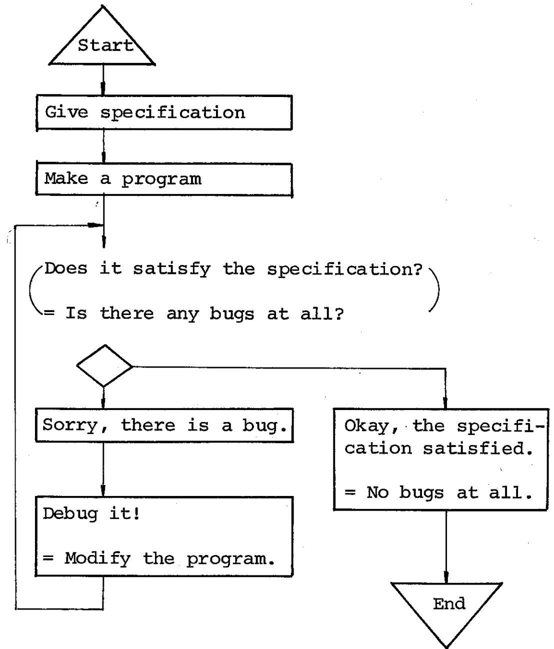

Fig. 1 A Process of Software Production

§3 Lisp—An Attempt to a Programming System

As mentioned previously, Lisp is regarded as a programming system so that it can tell what kinds of debugging tools are desirable in the general programming system.

[1] What is a programming system?

It would be better to give the definition: a computer supported environment where a program is made. The components are editor, interpreter, compiler, debugger and so on, and they are systematically integrated.

One may ask "What is the difference from the time sharing system and the operating system?" Well, it lies in the principal aim. Operating systems help the effective use of the computing resources. Time sharing systems provide the on-line, cooperating use of the

computer. Both do not primarily pursue the program development stage itself.

However, current time sharing systems such as Multics /3/, TENEX /1/, and UNIX /19/, provide efficient tools for program construction. And that Lisp has been developed and used in this kind of time sharing, on-line environments. Actually it is very important for a Lisp system to be able to invoke such system facilities at any time.

[2] Why Lisp has become a programming system?

1) Lisp is an interpreter language — Interactiveness.

Lisp program is executed by the interpreter.(*) When an error occurs, the interpreter halts and does not abandone all the execution. The user is permitted to investigate the situation and resume the execution. Here the debugging tools are very useful. Thus on-line debug is said to be 3 time efficient than off-line debug /2/.

2) Lisp has a program-data uniformity — Modifiability.

Because the Lisp program is expressed and stored in the same structure as data, it is very easy to write a program to modify the program. For example, editors, compilers, pretty-printers, optimizers, translators, and even interpreter itself can be written in Lisp.

3) Lisp is a functional language — Modurality.

(*) McCartly says that it was an accident to have the interpreter /14/.

A Lisp program is a collection of functions. It is natural to incorporate software tools in the system. Tools are gathered gradually through years and that a user can use them, refine them, and add new tools by himself.

4) Lisp is a common language — Portability.

Although there are various Lisp systems and there exist many differences, it is easy to transfer a Lisp program from one to the other. Tools written in Lisp are also easy to transfer. Lisp users can appreciate this benefit.

[3] What are told about debugging from Lisp?

1) Interactive debugging is very easy.

In a batch environment, it is difficult to detect and fix the bug because it is impossible to halt the execution, investigate the situation, fix the bugs and resume the execution.

2) Incorporated tools are very useful.

If the debugging tools can be invoked only after the program is discarded, the user will not use them however efficient they may be. Tools in the necessary situation are very valuable.

By the manual modification, another bug may be planted. The tools and systems to modify the program are indispensable for the productivity progress.

[4] What kinds of influence exist to other languages and systems?

1) Several new programming languages have been developed.

Excluding Lisp-based languages such as Micro-planner /21/ or Conniver /15/, there are new languages following the Lisp directions towards a programming system. Smalltalk /5/ and Logo /18/ are good examples. They are CAI languages which put stress on modification and debugging of the program.

2) Several tools are available for other languages.

Pretty-printer is the good example which is available in Pascal /25/, Fortran /8/, and PL/I /27/.

3) Micro-computer has introduced a cheap interactive environment.

Basic is one of the most popular language in micro-computers, because it has an editor incorporated. A programming system is most suitable for this interactive environment. So-called Lisp machines /12/ can be appreciated in this viewpoint.

§4. Lisp Debugging Tools — Experience

In this section reviewed are Lisp1.5/13/, Interlisp /24/, and Lisp1.9/9/.

[1] Lisp1.5

Tracer is the only debugging tool. Eventually, tracer is a useful tool even today. It is also useful to analyze the program. See the discussion in the next section.

[2] Interlisp

The main features of Interlisp debugging tools came from Teitelman's PILOT system /23/. They are the followings:

1) DWIM (Do-What-I-Mean) — automatic spell correction

Misspelled terms (variables or functions) are automatically corrected. It is done according to the following procedure: The system maintains lists of existing terms. When an unrecognized term (unbound variable or undefined function) appears, DWIM searches the list for the nearest. If the appropriate term is found within a limited time and distance, it is returned as the user's intention. It should be noted that if the search fails or plural candidates are found, then the error occurs. And that the user can interrupt the program any time when he thinks the system has picked up a wrong one.

This feature is said to be the most impressing /20/.

2) break package

When an error occurs, the system calls this package. The user, then, can investigate and change the status and resume the program. This package can be explicitly called by the function BREAK. It is useful to observe the program behavior.

3) program advising

To modify the existing routines including system functions, advising facilities are provided. Without altering the original codes, some procedures are attached before and after the function evaluation. Let f(x) be the object, M_{before} and M_{after} are such attachments, then the advised (i.e. modified) function f' is defined as $f' = M_{after}(f(M_{before}(x)))$. Interlisp contains other tools too.

4) Lisp editor

This editor edits Lisp data (S-expressions), i.e, it directly alters the present codes of the function. Alternative to this

editor is the conjuncted process of source text editing and function redefinition.

5) Pretty-printer

It clarifies the structure of the program by indentations.

6) Back-tracer

It edits the stack informations to present the evaluation steps so far.

7) Program analyzer

A function PRETTYSTRUCTURE tells the relations between functions in the program. It tells that a function calls so-and-so functions. Maclisp /16/ has a powerful INDEX package for this purpose.

8) UNDO capability

At the top level, interlisp has a facility to undo (i.e. cancel) the effect of preceding statements.

[3] Lispl.9 /9/

Although Interlisp is the most powerful system in the world, it cannot be said to be the ultimate. Some attempts on Lispl.9 are valuable.

1) channel system /10/

Input/output facilities in Lispl.9 are based on the conceptual channel which enables the following facilities:

(i) Monitoring file I/O at the terminal.

It is easy to detect the bug in file I/O.

(ii) Saving terminal I/O to the file.

Both the key-inputs and system-outputs can be gathered in the file for the later analysis.

(iii) Simulating terminal I/O by the file.

Repeating the same test is especially easy. It is also useful to demonstrate key-input program when combined the monitoring facility.

(iv) Tracer output can be saved on the separate file.

Voluminous output of the tracer can be saved without affecting terminal output.

(v) Null output facility can suppress unnecessary output.

Without changing the program, time consuming output can be suppressed.

2) connecting 'system debug and tracer'

Lispl.9 is an evolving system as most Lisp systems are. It means that a new bug is always detected, so the fast debugging is necessary. To this end, Lispl.9 incorporates a function PTRCE which invokes a system debug and tracer in TOSBAC-5600 /26/.

The package is a machine word level (not the Lisp level) debugger. Snapshot, patch, breakpoint setting, and tracing the machine word are possible. The user can, at any time, exit the package and resume the Lisp execution.

This package includes a statistics program for the execution counter of the location. It is useful to analyze the load of Lisp 1.9 programs.

§5 Lisp Debugging Tools — in the Future

Before predicting the future, we had better classify the debugging tools. There are three classes.

[1] Tools for bug detection

This is the first stage of debugging. Tracer, break-package, and back-tracer belong to this. Snapshot and tracing part of the system debug and tracer as well as channel system can also be included. Pretty-printer and program analyzer which are usually used to see that there is no bug are members because they can tell the existence of a bug.

[2] Tools for bug extermination

DWIM, program advisor, editor, and UNDO are this kind of tools. Patch facility is the typical one.

[3] Tools for constructing bug-free program

Up to the present this kind of tools have not yet incorporated. However, it is one of the main projects on software engineering and the many experiences, methodologies and theories have already accumulated.

Among the above three, the first kind of tools are popular in many Lisp systems. Interlisp is famous for its second class of tools. Then the advanced future system should incorporate the third kind. Of course, the first and second tools can be (and should be) more advanced and refined.

The following themes are occured to the author:

1) It is necessary to develop tools to construct bug-free programs.

Perhaps the most urgent is the specification describing and debugging system. Some specification systems are already presented (eg. /7, 11/), but the experience is not enough to produce an effective tools.

Automatic programming project is more ambitious and complete. And much more research is necessary both on theory and other aspects (esp. on human interface and its productivity).

2) It is promising to develop an automated debugging aid.

Even a limited automatic debugger such as DWIM is known to be very helpful. So if a little more advanced tools are available, they must be of great use. The debugging process itself has been researched in CAI/6/ and AI/22/. The program verification system which tells the debugger where to stop has been vigorously researched. The combination of the two will provide an advanced automatic debugging system.

3) More useful editor is wanted.

As Sandewall stated /20/, the Interlisp editor is not satisfactory. Perhaps the future computing environments and advanced terminal systems will offer various approaches to invent an ideal editor.

4) Powerful tracer is useful.

Tracer is a classical and even now useful tool. In the near future, a sophisticated tracer will be incorporated where conditional tracing, conditional breaking, conditional snapshot, and useful statistics will be available. And the user can edit the trace outputs so that unnecessary informations can be eliminated. It can be regarded as a sophisticated program monitor.

5) Intelligent program analyzer is useful.

Program analysis is necessary not only for the debugging but also for the compiler to

produce an efficient codes. Every experience should be integrated to make such an analyzer.

§6 Conclusion

Debugging process and programming system are at first discussed in this paper. Debugging process is indispensable and debugging tools should be incorporated into the programming system.

Lisp has been developed as a programming system, and its debugging tools suggest their status in the future programming system. Concerning actual examples, debugging tools of Lisp1.5, Interlisp, and Lisp1.9 are surveyed.

It is predicted that the future efforts will be paid in the construction of the bug-free system and the powerful tools such as editor and tracer which enable fast debugging.

These experiences and experiments in Lisp will lead to the more ambitious research problems, i.e. the automatic programming and the automatic debugging.

It is not long since the programming system approach was popularly recognized. Lisp has, however, experiences over ten years, and it is very helpful for everyone who tries to develop a programming system with high productivity.

References

/1/ D. G. Bobrow et al. "TENEX, a Paged Time Sharing System for the PDP-10" CACM, 16, 3 (Mar. 1972).

/2/ F.P. Brooks Jr. "The Mythical Man-Month" Addison-Wesley (1975).

/3/ F. J. Corbato and V. A. Vyssotsky "Introduction and overview of the Multics system" Proc. FJCC (1965).

/4/ E. W. Dijkstra "A Discipline of programming" Prentice-Hall (1976).

/5/ A. Goldberg & A. Kay (eds) "Smalltalk-72 Introduction Manual" Xerox PARC, SSL 76-6 (1976).

/6/ I. P. Goldstein "Understanding Simple Picture Programs" MIT AI-TR 294 (Sep. 1974).

/7/ J. V. Guttarg "The specification and application to programming abstract data types" Univ. of Toronto, CSRG-59 (1975).

/8/ B. W. Kernighan and P. J. Plauger "Software Tools" Addison-Wesley, (1975).

/9/ T. Kurokawa "LISP1.9 Programming System" J. Inf. Proc. Soc. of Japan, 17, 11 (Nov. 1976). (in Japanese)

/10/ T. Kurokawa "Input/Output Facilities in LISP1.9" SOFTWARE — Prac. & Ekp., 8 (1978).

/11/ T. Kurokawa "An Informal Introduction to Function-class: A programmable Specification Technique" unpublished memo (Jan. 1979).

/12/ T. Kurokawa "Lisp Activities in Japan" unpublished memo (Jan. 1979).

/13/ J. McCarthy et al. "LISP 1.5 Programmer's Manual" MIT Press (1966)

/14/ J. McCarthy "History of Lisp" ACM SIGPLAN Notices, 13, 8 (Aug. 1978).

/15/ D. V. McDermott and G. J. Sussman "The Conniver Reference Manual" MIT AI Memo 259a (1974).

/16/ D. A. Moon "MACLISP Reference Manual" MIT Project MAC (Aug. 1974).

/17/ P. Naur et al. "Software Engineering" Mason/Charter Pub. Inc. (1976).

/18/ S. A. Papert "Teaching Children Thinking" Programmed Learning and Educational Technology, 9, 5 (1972).

/19/ D. M. Ritchie and K. Thompson "The UNIX Time Sharing System" CACM, 17, 7 (1973).

/20/ E. Sandewall "Programming in an Interactive Environments: the "LISP" Experiences" ACM Comp. Surveys, 10, 1 (1978).

/21/ G. J. Sussman et al. "Micro-planner reference manual" MIT AI Memo 203A (1973).

/22/ G. J. Sussman "A Computational Model of Skill Acquisition" MIT AI-TR 297 (Aug. 1973).

/23/ W. Teitelman "Toward a programming laboratory" Proc. 1st IJCAI (1969) also in /17/.

/24/ W. Teitelman "Interlisp Reference Manual" Xerox PARC (1974).

/25/ E. Wada "Current tendencies on software development tools" Proc. Software Engineering Symposium (Jan. 1979). (in Japanese).

/26/ TOSBAC-5600 TSS System Debug and Tracer, TOSHIBA.

/27/ indent command in "Multics Programmers Manual, commands and active functions" HIS AG92 (Jan. 1975).