Invited Paper

HiLISP—a Common Lisp with an Optimizing Compiler on a Mainframe

MICHIAKI YASUMURA*††, KATSUHIKO YUURA*, MASAAKI KUROSU**,
TOSHIHISA AOSHIMA*, NOBUYUKI TAKEICHI*† and YOSHIMITSU OSHIMA*

We have developed a Lisp system based on Common Lisp on a main-frame computer. It is called HiLISP, which is the acronym of Hitachi's interactive Lisp Processor or High speed Lisp. We have designed L-code as an abstract Lisp machine code that is used both for the interpreter and the compiler. The HiLISP interpreter executes heavy Common Lisp features, such as lexical scope and multiple values efficiently. The HiLISP compiler optimizes compiled code by employing several optimization techniques such as program transformations, type inferences, and local optimizations. HiLISP system also provides a full screen editor and a full-fledged debugger. The most notable extension to Common Lisp is the Japanese character set handling feature. Based on our prototype implementation, a software product called VOS3 Lisp was developed and delivered as the first commercial Common Lisp system in Japan. HiLISP has been used for many applications, such as diagnostic expert system for LSI process, logic design system, layout CAD, natural language processing system, and so on. Based on the HiLISP system, an object oriented system and an interactive program transformation system have been also developed.

1. Introduction

Lisp is the second oldest programming language and has been widely used among AI researchers in academic community. Lisp is suitable for prototyping and interactive programming. It is also useful for symbolic computations because of the powerful memory management and metamorphism of data and functions. Nevertheless, Lisp had many dialects which are not compatible with each other.

We have started designing our Lisp system for an AI research tool in our laboratory. At the beginning, the language specification was a kind of MacLisp family. Just after our project started, we learned the Common Lisp specification [21].

Common Lisp is different from the older Lisp's in several respects. In older Lisp's, some problems exist:

(1) Semantics of most interpreters are usually different from that of the compilers, since variables are bounded dynamically in interpreters while in compilers they are bounded statically, which is called static scope rule.

(2) When a function is passed as an argument or returned as a result, free variables of the function are not treated correctly in some Lisp implementations. This problem is called function argument problem or fun-arg problem, in short.

(3) Lisp is a functional language; It normally returns single result and arguments cannot be altered (call by value parameters). When a user wants to return multiple results, he/she must make a list to combine these results.

(4) In Lisp programs, variable names and/or function names may often conflict with each other due to the single global name space for large applications.

(5) Recently many types are gradually introduced to Lisp's and the number of specific function names are increasing.

Common Lisp has solved these problems as follows:

(1) Adopts static scope rule both for compiler and for interpreter.

(2) Adopts static scope closure, which handles free variables in a function argument correctly.

(3) Extends to allow multiple results, which are called multiple values.

(4) Introduces packages, which are used to divide the single global name space into many sub spaces.

(5) Introduces generic functions, which allows polymorphism of types.

We decided to change our language specification to Common Lisp. Though Common Lisp has several desirable features described above, it is big and heavy by nature. And it is a challenging subject for implementors to implement it efficiently and compactly.

2. Design Philosophy and Development Methods

We made our design goals for our Lisp-HiLISP

*Central Research Lab., Hitachi Ltd.
**Design Center, Hitachi Ltd.
†Currently, Hitachi Information Systems, Ltd.
††Currently, Keio University at Shonan Fujisawa.

which is the acronym on Hitachi's interactive List Processor or High speed Lisp. The latter one symbolizes our ultimate goal.

Our design goals are as follows:

(1) To run efficiently—We would have liked to make our Lisp processor as fast as other language processors, such as C or Pascal on stock hardware. Also we wanted it to run faster than or at least not slower than non-Common Lisp's.

(2) To comply with Common Lisp—The language specification must be conformed to that of Common Lisp with slight extensions.

(3) To support Japanese character set—Japanese characters and strings should be handled in the same way as English alphabet.

(4) To provide user friendly programming environments—It should provide a full screen structure editor and a full-fledged debugger which must provide user friendly interfaces.

(5) To provide larger memory space—Older Lisp processors on mainframes had memory of not more than 16MB memory, but it is necessary to allow at least 2GB memory for newer Lisp processor.

(6) Also—interfaces to other languages and to Operating System are required.

The goals are listed in order of importance. These goals may conflict with each other. For example, to comply with Common Lisp, we have to support some heavy features of Common Lisp, for example, key word parameter, multiple values, generic functions, full closures, and so on. In such cases, higher priority precedes lower priority. And we implemented normal case as efficient as possible while special case may run in less efficient manner.

Before designing our HiLISP system, we defined our design goals which have been described above. Among them, runtime efficiency and Japanese character set handling are notable, because we thought most Lisp processors were very slow and few Lisp systems supported Japanese characters at that time. A few years later when our first prototype was running, we learned most Lisp users felt almost the same way as we had thought. According to the JEIDA report [16], when classified by the response of Lisp users to their Lisp processors at that time, 15% of the responders were not satisfied with runtime speed, 13% complained of incompatibilities and lack of capabilities 12% wanted Japanese character supports, 8% wanted better programming environments, and 8% wanted more memory spaces. Also 13% of the responders felt graphic system supports were poor and 9% wanted object oriented capability. But 9% of the responders were satisfied with their processors which were believed to be built by themselves.

To develop HiLISP, we first designed an abstract Lisp machine, the code of which would be used both for the interpreter and the intermediate code of the compiler. We call it L-code. L-code represents pseudo Lisp machine code and was implemented in macro assembler. L-code is designed for three purposes:

(1) To minimize machine dependent features, as few as possible.

(2) To obtain high productivity during the development of our Lisp system.

(3) To describe efficient code which must be comparable to assembly code.

The first purpose is not only for portability but also for integrity and security of HiLISP system. The development of HiLISP system was done by several people and it is very important to have clear interface to every module. We think the detail of implementation, such as data structures, tag information, calling sequence and so on should be encapsulated from each implementor as much as possible. Thus this first purpose can go together with the second purpose. Because usually he or she can concentrate on what has to be implemented. Generally speaking, an average programmer can use L-codes which can be expanded into best code sequence that has been carefully implemented. But this is not always the case. Therefore we made a peephole optimizer for the intermediate code of the

Table 1 Summary of typical Lcode.

#	Classification	Lcode	Remarks
1	Function interfaces	¥ENTRY	function entry
		¥PREPARE	frame creation
		¥CALL	function call
		¥RETURN	function call
		¥EVAL	eval call
2	variable accesses	¥VALUE	value reference
		¥SETVALUE	value setting
		¥INTERN	interning
		¥BIND	binding
		¥UNBIND	unbinding
3	stack operations	¥PUSH	push
		¥POP	pop
		¥ARGREF	argument reference
		¥ARGSET	argument setting
4	list operations	¥CAR	car reference
		¥CDR	cdr reference
		¥CARSET	car setting
		¥CDRSET	cdr setting
5	memory allocations	¥CONS	cons
		¥LIST	list
6	data accesses	¥GETSYM	set symbol in register
		¥GETSTR	set string in register
		¥GETFIX	set fixnum in register
7	branches	¥IFxxx	xxx is a data type, t, nil
		¥GOTO	unconditional branch
8	data definitions	¥END	end of function
		¥CONST	constant definition
		¥SYMBOL	symbol definition
9	utilities	¥KEY	keyword param match
		¥GET	property list retrieval
		¥TYPERROR	type error process

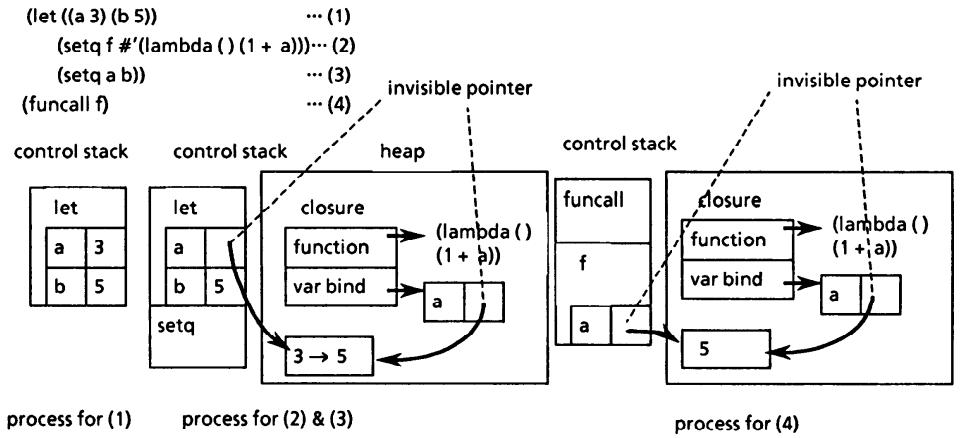

Fig. 1 An implementation of variable access in stack and closure by invisible pointer.

compiler, which will be described later, and also added some variations of L-code for implementing critical part of interpreter and built-in functions. The summary of L-code is shown in Table 1.

At the beginning we first designed L-code and wrote the kernel of the interpreter in L-code. Then we developed the compiler and built-in functions. By using L-code, program steps of interpreter and built-in functions became much smaller than that of only assembly coded ones, and we could rewrite the whole system with little interface trouble.

3. Interpreter

Common Lisp is different from older Lisp's since it is a compiler oriented Lisp. For example, Common Lisp adopts static scope rule, and it contains generic functions. The consequence is that a Common Lisp interpreter would be slower than those of older Lisp's in general.

We tried to make our HiLISP interpreter not much slower than the older Lisp interpreters. Here we describe how we have implemented some heavy Common Lisp features.

(1) Stack frame

We have decided that HiLISP should have two stacks, one for control and parameter passing and the other for binding stack and that it should pass parameters on the stack and return the result on the registers.

(2) Variable access

For variable access, a shallow binding technique is usually used for older Lisp interpreters, since it is faster than deep binding technique using a-list. But with a shallow binding technique it is difficult to handle static scope rule and fun-args properly.

In HiLISP interpreter [32], we put deep binding structure in array form on a stack which is faster than that in list form in heap. When it encounters a function closure, it moves the binding structure into heap and changes access pointers into invisible pointers (Fig. 1).

(3) Multiple values

A naive implementation of multiple value is to put a result or results always in a stack and to keep the number of result(s) in a register.

In HiLISP we prefer to put single result in a register, and put only multiple results in a stack. Therefore we develop multi-path method for this problem [32]. This method is based on the idea of changing return address for different numbers of result(s). One might be worried about the increase of combination of paths, but this is not the case. Because there are only two states, single value state and multiple value state, and the combination of multiple paths can be merged into confluent of paths.

In a single result case there is no time loss compared to the older Lisp's implementation and in multiple result case it has a little time loss. Since the latter case is rare, the overall increase in overhead is small.

(4) Built-in functions

Overall interpreter speed depends on not only the kernel part, but also built-in functions. Because in Lisp, a large part of the execution time is usally spent in built-in functions.

We designed and coded most built-in functions to be fast and some important built-in functions to be as fast as possible. For this purpose, the L-code approach was useful.

(5) Memory management

In HiLISP each cell consists of two words, one for tag and the other for pointer to allow full address capability.

HiLISP was implemented on a mainframe with vir-

tual memory and we adopted copying garbage collection method to keep locality. Later on when the VOS3 Lisp was built based on HiLISP, this garbage collection method was changed into a mixture of mark and sweep method for *cons* and copying method for other data types on behalf of the memory efficiency.

4. Compiler

4.1 Structure of the Compiler

The HiLISP compiler compiles Lisp programs in there phases and generates two intermediate codes, L-code and LAP. L-code is an abstract machine code and LAP is a Lisp Assembly Program which is a machine dependent assembly code.

The first stage of the HiLISP compiler analyzes source code and generates L-code. At this stage, closure is analyzed and some program transformations are done (this will be explained in the next sub section). Then the next stage of the compiler generates LAP from L-code and final stage generates machine code.

4.2 Optimizations

Before starting the design of HiLISP compiler, we analyzed several Lisp programs. Firstly, we learned that function calls in Lisp are much more frequent than those in other languages, say, C, Pascal, or Fortran and that it is not unusual for only function calling to spend more than half of the overall runtime. Secondly, Common Lisp language adopts generic functions which yield lots of runtime type checks. Finally, since we implement our HiLISP compiler on a mainframe, which is a highly pipelined machine, the HiLISP compiler should generates codes which utilizes the pipeline mechanism.

Therefore in order to design optimizations of the HiLISP compiler, we concentrated the following three points:

(1) Function call optimization
(2) Type check optimization
(3) Local optimization to boost pipeline.

(1) Function call optimizations

Two approaches are possible to attain function call optimizations; one is (a) reduction of calling time, the other is (b) reduction of number of calls.

In order to reduce function call time, the preparation time of the stack frame for function call and unframe process time for function return must be shortened as much as possible. We designed the function call instruction sequence and its machine cycle to be as short as possible under the assumption that the interface with interpreter must be preserved. We also partitioned single type of calling sequence into several types of short patterns depending on the callee's conditions such as the self call, machine code call, and/or fixed number parameters. Therefore ¥call instruction of L-code has several variations.

For reducing the number of calls, we applied the tail recursion removal method, which is common in many Lisp compilers, enforced the inline expansion of built-in functions, and developed an automatic expansion method for self recursive functions and user functions calls (Fig. 2). Automatic expansion is carefully done by checking side effects and by preparing temporaries (if necessary) to prevent increase of function calls. The detail of the method was explained in [27].

(2) Type check optimization

In order to optimize runtime type check in Lisp, there are two methods; one is (a) the reduction of type check time at runtime, and the other is (b) the compile time type check.

Type check time at runtime is sped up by performing more frequent tag check earlier and by reordering instruction sequence to minimize the branch time after checking. The latter was implemented as a part of local optimizations.

The compile-time type check reduces runtime type check by identifying the types of data or operations

Table 2 Performance comparison of HiLisp with Pascal.

#	Benchmark	HiLISP	Pascal 8000	ratio (P/L)
1	Tak	41. ms	69. ms	1.68
2	Flo-tarai-4	22.	20.	0.91
3	Qsort-100 (list version)	37.	66.	1.8
4	Bubble-50 (array version)	5.3	2.6	0.49
5	Nqueen-8 (list version)	95.	304.	3.2
6	Nqueen-8 (array version)	53.	25.	0.47
	Geometric Mean	—	—	1.15

M680H

```
(defun f(x)(if (px)(g(fx1)(fx2). . .)(qx))
    ↓
(defun f(x)
  (if (px)(g (if (px1)(g(fx1|x←x1)(fx2|x←x1). . .)(qx1))
             (if (px2)(g(fx1|x←x2)(fx2|x←x2). . .)(qx2))
             . . .)
      (qx)))
```

Fig. 2 Scheme of Self-inline expansion ($x_1|x\leftarrow x_2$ means the substitution of x in x_1 by x_2.).

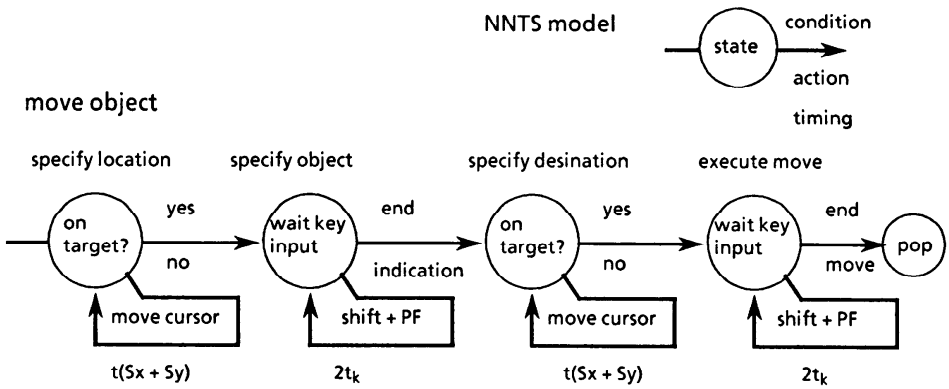

Fig. 3 NNTS model and a simple example.

based on the users' declaration and other available information. The detail of the method was explained in [27].

(3) Local optimization

The local optimization of HiLISP is classified into the optimization in L-code level (Local-opt1) and the optimization in LAP level (Local-opt2).

Local-opt1 optimizes L-code locally in machine independent way. The main optimizations are as follows:

- Removal of unnecessary codes.
- Optimization of branch codes.
- Removal of stack limit chekcs.

Local-opt2 optimizes LAP code locally in machine dependent way. The main optimizations are as follows:

- Removal of unnecessary instructions.
- Reordering of instruction sequences.

Mainframe computers have powerful pipeline control mechanism which executes most operations in one cycle effectively. But when the pipeline is disturbed, the execution time becomes slower than one cycle. The following cases are the sources which disturb pipeline:

- Condition code set and its use (conditional branch).
- Index register/base register set and their uses.
- Value set and use in one location in memory.

Local-opt2 checks above cases and reorders instruction sequence so that conflicting instructions are laid apart. Notice that branch optimization done in Local-opt1 is also useful to enhance pipeline capability, because it reduces the number of branches.

4.3 Performance Evaluation

It is widely believed that Lisp programs are slower than Pascal or C programs. After evaluating several benchmark programs, we found this is not true. If the Lisp compiler and its built-in functions are carefully designed and implemented to generate efficient codes. Lisp programs can run no slower than programs written in C or Pascal. Table 2 shows some results of the comparison between the performance of Lisp programs and Pascal programs. For each program, each algorithm for Lisp and corresponding algorithm for Pascal are the same. Ratio is the execution time in Pascal divided by that in Lisp. In geometric mean, the ratio of performance is almost the same. Lisp is faster in *tak* which has many self recursion and in which tail recursion removal and self recursion expansion are effective. As the data for *Qsort* and *Nqueen* indicate, Lisp is faster in list version and Pascal is faster in array version in general. Usually users tend to write prototype programs in Lisp with list structure then rewrite them in Pascal or C with array structure, and they misunderstand Lisp is slower than Pascal or C. But the truth is that programs written with lists are slower than those with arraies.

5. Programming Environments

HiLISP was built on a mainframe computer which has half-duplex terminals with local screen editing capabilities. We tried to utilize this local screen editing capabilities to enhance usability of Lisp programming environments. Under this consideration, we developed a structural editor, which we internally called HiEditor and a full debugger called HiDebugger.

(1) HiEditor

HiEditor is an amalgamation of screen editor and syntax (or structure) editor [12]. To be edited by HiEditor is not only programs/data in files but also programs/data in heap. The specification of the screen editing capabilities is compatible with the ASPEN which is a general purpose editor for Hitachi's mainframes [38].

For syntax editing, we have introduced the concept of structural cursor which represents any level of S-expression visually. The S-expression specified by the structural cursor can be deleted, moved, and copied. In addition to the structural cursor, correspondence of

parentheses can be checked simply by pushing a function key.

In order to evaluate editing operations, we have developed the NNTS (Node Network with Time Specification) model (Fig. 3). This NNTS model is an augmented extension to GTN (Generalized Transition Network) model proposed by Kieras [11]. Compared to GTN, NNTS model has time description capability and mental process description capability. The time description capability of NNTS model is influenced by KLM model by Card [4]. By using NNTS model, we found the speed of HiEditor is 2.0 times faster than that of simple screen editor [17].

We have developed HiEditor all written in HiLISP except screen control part.

(2) HiDebugger

HiDebugger contains backtracer, stepper, break kernel, tracer, inspector, and so on. One unique feature of HiDebugger is the "where" command which shows erroneous function and location in source to the user.

At the beginning of our development of the HiLISP system, we made a very small debugger, which included only backtracer and inspector for stack frame. But it was very helpful to have a debugger at the very beginning for our development. Gradually our debugger became larger with the growth of the HiLISP system.

6. Some Extensions

At early stage, HiLISP was a subset of Common Lisp, though we extended its specification. The most important extension is Japanese character set handling. The Japanese character set handling was an important extension to our purpose, because one of our presumable applications was natural language processing. In Lisp, if we could extend character set in character type and in string type, it would be used not only string and character data but also function and variable names.

To support Japanese character set in Lisp, both character type and string type should be extended to allow Japanese characters which have double byte length. There are three ways to implement Japanese character set:

(1) Mix string approach (three string approach)—two byte character set coexists with one byte character set.

(2) Fat string approach (one string approach)—every one byte character is converted internally into two byte character.

(3) Independent string approach (two string approach)—one byte character strings and two byte character strings do not interfere with each other.

The second and third approaches are easier to implement than first one, but second one needs more memory space and third one lacks the capability of handling Japanese-English mixed texts. Therefore we first adopted mix string approach. Originally we made three string types, one for English only strings, another for Japanese only strings, and the other for mixed type in order to use memory space more efficiently. For implementing mixed string, we actually mixed one byte characters with two byte characters and case shift characters were used to separate them. But this implementation was hard to complete, because the string functions for mixed string needed very complicated coding. And we switched to fourth approach, which is the modified approach of third one. Form the implementor's view point, this new approach is an extension of two string approach; one for thin string, the other for fat string which may contain both English characters and Japanese characters in double byte boundary. From the user's point view, fat string is the only normal string and thin string can only be used by a special declaration.

Other extensions of HiLISP to Common Lisp are display manipulating functions, interfaces to TSS, interfaces to other languages, and so on.

7. Applications and Evolutions

Based on our HiLISP prototype built in our laboratory, engineers in Software Works built a software product called VOS3 Lisp, which was shipped on April 1987 as the first commercial Common Lisp made in Japan.

HiLISP has been used for many AI applications in our laboratory; The examples are layout CAD system, Processor Architecture Simulator called PASIM [13], LSI process diagnostic system [7], natural language interface routine, Knowledge based tutor system, and so on.

VOS3 Lisp has been used in many universities and companies. The Symbolic Algebra system, Reduce, is running on VOS3 Lisp at the Computer Centre of University of Tokyo. The Syusin, which is a Japanese text manipulation language designed by S. Mizutani [15], runs on VOS3 Lisp at Tokyo Woman's Christian University. Another example is a design verification system for closed sequential control circuits of power supplies [35] developed by the collaboration of TEPCO (Tokyo Electric Power Company) and Hitachi Ltd.

One year after the shipment of VOS3 Lisp, a Common Lisp system for Work Station was shipped from the Software Works. The Lisp is called HI-UX Lisp, since it runs on HI-UX system of the 2050 Work Station. The HI-UX Lisp is fully compatible with VOS3 Lisp.

On our HiLISP system, we have developed two systems for extending usability of HiLISP, one is the two HiOBJ systems which are object oriented systems on HiLISP and the other is IPTS-an Interactive Program Transformation System.

Originally HiOBJ-1 was designed on the different concept with that of CLOS. HiOBJ-1 has unified class

definition capability for encapsulation, simple slot access capability, slot daemon, and so on [28]. A simple graphic manager was built on the HiOBJ-1. See Appendix for the syntax of HiOBJ-1. Later on HiOBJ-2 [35] was developed. The specification of HiOBJ-2 complys with CLOS [37], but the implementation takes after HiOBJ-1.

From the experiences of the optimizations of the HiLISP compiler, we would liked to build an interactive program transformer, where the system can get enough information from the user to transform programs into more efficient ones. The system we built was called IPTS [33]. This system is a rule based pattern driven program transformation system. By using this system, a layout CAD program was transformed into a 15% faster program [34].

8. Conclusion

The HiLISP was designed and implemented on a mainframe computer as a tool for AI researches in our laboratory. The language specification is based on Common Lisp, but we extended its specification to allow Japanese character set handling. The HiLISP system was carefully implemented to give enough performance and usability on a mainframe computer.

Based on the HiLISP prototype, VOS3 Lisp was developed and shipped as the first commercial Common Lisp made in Japan. Both HiLISP and VOS3 Lisp have been used for many AI applications. Based on HiLISP further researches, such as object oriented system, window system, program transformation system, and tutor system are still going on.

Acknowledgments

The authors are grateful to Ayako Takada, Kunio Nomoto, and Hisashi Takahashi who participated in development of HiLISP. The authors would like to acknowledge Dr. Takeshi Nakayama for his contribituion to NNTS model. The authors also thank Kazuo Morita, Ken Sakaibara, Mitsuhiro Nagata, and Hiroshi Isobe of Software Works for their efforts to make HiLISP into a software product.

References

1. ALLEN, J. R. Anotomy of Lisp, *McGraw-Hill* (1978).
2. BROOKS, R. A., GABRIEL, R. P. and STEELE, G. L. Jr. An Optimizing Compiler for Lexically Scoped LISP, *Proc. of the 1982 ACM Compiler Construction Conf.* (1982), 261-274.
3. BURSTALL, R. M. DARLINGTON, J. A Transformation System for Developing Recursive Programs, *J. ACM*, **24**, 1 (January 1977), 44-67.
4. CARD, S. K., MORAN, T. P. and NEWELL, A. The Keystroke-Level Model for User Performance Time with Interactive System, *Comm. ACM*, **23**, 7 (1980), 396-410.
5. CHIKAYAMA, T. Implementation of the UtiLisp System, *Trans. of IPSJ*, **24**, 3 (in Japanese) (1983), 599-604.
6. DARLINGTON, J. Program Transformation, *Functional Programming and its Applications*, Ed. by J. Darlington, *et al.*, Cambridge Univ. Press (1982).
7. FUNAKOSHI, K. and MIZUNO, K. Rule-Based Diagnostic Expert System for LSI Manufacturing Process Flow, *The Hitachi Hyoron*, **70**, 11 (in Japanese) (November 1988), 131-135.
8. HAGIYA, M. YUASA, T. Implementation of Kyoto Common Lisp, *First Conf. Proc. of Japan Society for Software Science and Technology* (In Japanese) (1984), 65-68.
9. IDA, M., TAKEUCHI, I., HAGIYA, M., YASUMURA, M., YUASA, T. and TERASHIMA, M. Panel Discussion: Common Lisp, *Trans. IPS Japan*, **27**, 1 (in Japanese) (January 1986), 67-77.
10. ISHIHARA, K., YASUMURA, M. and TAKAHASHI, S. Overview of Hitachi's Knowledge Information Processing System, *Hitachi Review*, **37**, 5 (October 1988), 303-308.
11. KIERAS, D. and POLFON, P. G. An Approach to the Formal Analysis of User Capability, *Int's Journal of Man-Machine Studies*, **22** (1985), 365-394.
12. KUROSU, M., NAKAYAMA, T. and AOSHIMA, T. HiLISP Programming Environment (1): Software-oriented Display Editor and its Implementation, *34th National Conf. of IPSJ* (in Japanese) (Mar. 1987), 935-936.
13. KOIKE, M., NAGASAKA, M., KURIYAMA, K. and WADA, K. Processor Architecture Simulator—PASIM: Overview, *39th National Conf. of IPSJ* (in Japanese) (October 1989).
14. MANNA, Z. Mathematical Theory of Computation, *McGraw-Hill* (1974), 448.
15. MIZUTANI, S. Guide to Syusin (Red Lisp)—A Japanese Fasioned Programming Language for String Manipulation, *Tokyo Woman's Christian University* (in Japanese) (1989).
16. JEIDA (Japan Electronic Industry Development Association), Ed., *The Report of the Surveys on Micro Computers* [II]—Common Lisp—, 61-A-235 [II] (in Japanese) (1986), 155.
17. NAKAYAMA, T., KUROSU, M., AOSHIMA, T. and OSHIMA, Y. HiLISP Programming Environment (2): A Model for Evaluation of Editing Operation, *34th National Conf. of IPSJ* (in Japanese) (March 1987), 937-938.
18. OKUNO, H. The Proposal of the Benchmarks for the third Lisp Contest and the first Prolog Contest, *Report of WGSYM*, IPSJ, **28-4** (in Japanese) (1984).
19. OKUNO, H. The Report of the Third Lisp Contest and the First Prolog Contest, *Report of WGSYM*, IPSJ, **85**, 30 (1985).
20. PARTISCH, H. and STEINBUGER, R. Program Transformation Systems, *ACM Computing Surveys*, **15**, 3 (1983).
21. STEELE, G. L. Jr. Common Lisp: the Language, *Digital Press* (1984), 645.
22. TAKADA, A., YASUMURA, M. and AOSHIMA, T. New Scheme of High Speed Code Generation for HiLISP compiler, *3rd Conf. Proc. of Japan Society for Software Science and Technology* (in Japanese) (1986), 97-100.
23. TAKEICHI, N., YASUMURA, M., YUURA, K. and MORITA, K. High Performance List Processor System, *The Hitachi Hyoron*, **69**, 3 (in Japanese) (March 1987), 13-16.
24. WHOLEY, S., FAHLMAN, S. E. The Design of an Instruction Set for Common Lisp, *Conf. Record of the 1984 ACM Symp. on Lisp and Functional Programming* (1984), 150-158.
25. YASUMURA, M., TAKADA, A. and YUURA, K. Program Transformation Techniques in Recursive Programs, *Riken Symp. on Functional Programming*, FP-87-03 (in Japanese) (Feburary 1987), 20-27.
26. YASUMURA, M., TAKADA, A., YUURA, K. Some Program Transformations of Recursive Programs in Lisp, *34th National Conf. of IPSJ* (in Japanese) (March 1987), 733-734.
27. YASUMURA, M., TAKADA, A. and AOSHIMA, T. Design and Implementation of an Optimizing Compiler for Common Lisp, *Trans. IPS Japan*, **28**, 11 (in Japanese) (November 1987).
28. YASUMURA, M. and YUURA, K. HiOBJ—A Proposal of Object Oriented Language on Common Lisp, *5th Conf. Proc. of Japan Society for Software Science and Technology* (in Japanese) (September 1988).
29. YUURA, K., TAKADA, A., AOSHIMA, T., YASUMURA, M., KUROSU, M. and TAKEICHI, N. Implementation of HiLISP—High Performance Common Lisp, *33rd National Conf. of IPSJ* (in Japanese) (October 1986), 475-476.
30. YUASA, T. and HAGIYA, M. Implementation of Kyoto Common Lisp, *Report of WGSYM, IPSJ*, **34-1** (in Japanese) (1985).
31. YUURA, K. and YASUMURA, M. High Speed Methods for the HiLISP Interpreter, *Report of WGSYM, IPSJ*, **40-5** (in Japanese) (1987).
32. YUURA, K. and YASUMURA, M. Some High Speed Methods and their Experimental Evaluations for Common Lisp Interpreters, *Trans. IPS Japan*, **30**, 6 (in Japanese) (June 1989), 719-724.

33. YUURA, K. and YASUMURA, M. Improvement of Efficiency of Lisp Programs by Interactive Program Transformations, *5th Conf. Proc. of Japan Society for Software Science and Technology* (in Japanese) (September 1988).
34. YUURA, K., YASUMURA, M. and NAGATA, M. HiLISP: High-Performance Common Lisp and its Optimizing Tool, *Proc. of the first European Conf. on the Practical Application of Lisp* (March 1990).
35. YUURA, K., SAKAIBARA, K., TAKAHASHI, H. and FUNATSU, T. Object-oriented Programming with HiOBJ-2 on CLOS, *39th National Conf. of IPSJ*, (in Japanese) (October 1989), 1348–1349.
36. YAMADA, N., KOBAYASHI, Y., FUJII, D., UEDA, Y., ITO, J., MATSUDA, S. and YOSHIZAWA, J. A Design Verification for Sequential Control Circuits No. 2, *39th National Conf. of IPSJ* (in Japanese) (October 1989), 1689.
37. BOBROW, D. G. *et al.* Common Lisp Object System Specification, *Draft X3J13 Document* 88-002R (1988).
38. VOS3 Programming Supporting Editor ASPEN Manual, 8090-3-330, *Hitachi, Ltd.* (in Japanese) (1985).

(Received November 8, 1989)

Appendix: Syntax Summary of HiOBJ-1 [28]

defclass *class-name* (*class-options*)({*slot-description*}*)
  {*doc-string*}*{*method-description*}*[Macro]
class-options::={[{*class-parameter*}*]
  :super({*class-name*|[*class-arguments*]})*)|
  :instance|{:constructor *init-name lambda-list*}*}*
class-parameter::=*var*
class-arguments::={:*class-parameter value*}*
slot-description::=(*slot-name*[*slot-options*])
slot-options::={:init *value*|:type{*type-name*|*class-name*}|
  :alloc{instance|class|family|none}|
  :access{private|family|public}|
  :read-only|:if-gets *method-name*|:if-sets *method-name*}*
method-description::=
  (method *method-name method-lambda-list*{*declaration*|*doc-string*}*{*form*}*)
method-lambda-list::=({*var*|*parameter-specifier*}*
  [&optional{*var*|(*var*[*initform*[*svar*]])}*] [&rest *var*]
  [&key{*var*|({*var*|(*keyword var*)}[*initform*[*svar*]])}*
[&allow-other-keys]]
  [&aux{*var*[*initform*]})*])
parameter-specifier::=(*var class-name*)
declaration::=(declare{*declaration-specifier*}*)
declaration-specifier::=*method-options*|
  other-declaration-specifiers-as-CommonLisp
method-options::={(access{(access{private|family|public})
message-passing::=(*selector-name instance*{*arg*}*)|
  (send *instance selector-name arg-list*)
selector-name::=*method-name*
slot-access::=(*slot-name instance*[:class-name])|.*slot-name*
slot-update::=(setf(*slot-name instance*[:class-name]) *value*)|
  (setf.*slot-name value*)
with-form::=(with *instance*{*form*}*)
psuedo-variable::=self
instance-creation::=(new *class-name*[*class-arguments*][*initial-values*])
  (*init-name*{*value*}*)
class-arguments::={:*class-parameter value*}*
initial-values::={:*slot-name value*}*