Local Optimization in a Compiler for Stack-based Lisp Machines

Larry M. Masinter and L. Peter Deutsch

Xerox Palo Alto Research Center

Abstract: We describe the local optimization phase of a compiler for translating the INTERLISP dialect of LISP into stack-architecture (0-address) instruction sets. We discuss the general organization of the compiler, and then describe the set of optimization techniques found most useful, based on empirical results gathered by compiling a large set of programs. The compiler and optimization phase are machine independent, in that they generate a stream of instructions for an abstract stack machine, which an assembler subsequently turns into the actual machine instructions. The compiler has been in successful use for several years, producing code for two different instruction sets.

I. INTRODUCTION

This paper describes the local optimization phase of a compiler for translating the INTERLISP [Teitelman et al., 1978] dialect of LISP into stack-architecture (0-address) instruction sets [Deutsch, 1973]. We discuss the general organization of the compiler, and then the set of optimization techniques we have found most useful. The compiler and optimization phase are machine independent, in that they generate a stream of instructions for an abstract stack machine, which an assembler subsequently turns into the actual machine instructions. The compiler has been in successful use for several years, producing code both for an 8-bit LISP instruction set for the Alto and Dorado, [Deutsch, 1978, 1980, Burton *et al.*, 1980], and a 9-bit instruction set for Maxc, a microprogrammed machine running the TENEX operating sytem [Fiala, 1978].

There are always tradeoffs in designing a compiler. Each additional optimization usually increases the running time of the compiler as well as its complexity. The improvement in the code generated must be weighed against the benefit gained, measured by the amount of code improvement weighted by the frequency with which the optimization is applicable. Rather than provide a multiplicity of compiler controls, which most users would not want to know about, the compiler designer should use empirical knowledge of "average" user programs and make appropriate design choices. One of the major purposes of this paper is to publish some empirical results on the relative utility of different code transformations, which can aid designers in making such choices.

Why this compiler is different

Compiling LISP for a 0-address architecture differs from compiling other languages such as PASCAL or ALGOL for several reasons. Procedures are independently compiled, so that global optimization techniques are not relevant. Compiling for a stack-based instruction set is different from compiling for more conventional machine architectures, in that register allocation is not relevant, and randomly addressable compiler-generated temporary variables other than top-of-stack are difficult to access.

In systems which provide interactive, symbolic debugging of compiled code, a compiler must not manipulate source programs too freely, since even common optimizations like tail recursion removal make it difficult or impossible to explain the dynamic state of the program in terms of the original source. However, LISP also provides an interpreter which can be used for debugging purposes when strict faithfulness is needed; interpreted and compiled code can be mixed freely. Thus, we take the view that the compiler can rearrange the implementation of an individual function in any manner consistent with the semantics of the original program, even if fine-grained debugging information may be lost or altered (e.g., if variables that appeared in the source get eliminated).

What we did not handle

The compiler concentrates on local optimizations. More global transformations such as pulling invariants out of loops or duplicate expression elimination would probably pay off often enough to be worth the additional complication in an environment where speed

was of great concern and the individual functions were large.

Related work

A few of our compiler's transformations, such as cross jumping and tail recursion removal, have been part of the literature for some time. We know of three other LISP compilers that compile into a machine-independent intermediate language and do substantial optimization.

The Standard LISP project at the University of Utah has produced a transportable compiler similar to ours [Griss & Hearn, 1979]. Their intermediate language is register- rather than stack-oriented. Their report mentions a number of the optimizations in our list, plus others only applicable to register machines, but their list is shorter and not accompanied by empirical data.

Another similar compiler was the subject of a Ph.D. dissertation [Urmi, 1978]. The author in this case was more concerned with the design of instruction sets than with optimizing the use of a given architecture. His report contains extensive statistics on the opcode frequencies, and interesting suggestions for instruction set design, including a consideration of both stack- and direct-address architectures; however, his optimizations are all in the "peephole" category, being limited to a few adjacent instructions, except for the usual optimization of ANDs and ORs.

The RABBIT compiler [Steele, 1978] translates an unusual lexically scoped LISP dialect into code for a register machine. Its optimization techniques are extremely sophisticated with regard to removal of recursions and variable bindings. However, the differences in coding style resulting from lexical scoping are so large that a comparison between RABBIT's goals and those of our compiler would not be meaningful.

Results

Optimization in the byte compiler provides an average 5-10% speed improvement and a 10-15% space improvement over completely unoptimized code. While significant, this does not make it one of the more significant factors affecting the performance of our LISP systems [Burton et al., 1980]. The most significant effect that a reasonable optimizing compiler has for its users is a certain amount of unconcern for vagaries of syntax. Programmers can write their routines for clarity, without concern for purely syntactic devices which might otherwise affect performance. For example, while inserting assignments inside expressions is allowed and occasionally perspicuous, it generally is more readable to perform variable assignments in separate statements, and to subsequently use the variables in an unnested manner. Knowing that the compiler will do an adequate job of optimization means that a program author can make choices based on legibility, even in the most time-critical routines.

II. ABOUT THE COMPILER AND THE OBJECT LANGUAGE

The compiler operates in several passes. The first pass takes the S-expression definition of the function being compiled, and walks down it recursively, generating a simple intermediate code, called ByteLap, analogous to assembly code. During this first pass, the compiler expands all macros, CLISP, record accesses and iterative statements. A few optimizations are performed during this pass, but most of the optimization work is saved for later. The next pass of the compiler is a "post-optimization" phase, which performs transformations on the ByteLap to improve it. Transformations are tried repeatedly, until no further improvement is possible.

After the post-optimization phase is done, the results are passed to an assembler, which transforms the ByteLap into the actual machine instructions. We currently have two different assemblers in use, which generate code for two different instruction sets: one for the Maxc 9-bit instruction set and one for the Alto/Dorado 8-bit instruction set. The Maxc and Alto/Dorado implementations of INTERLISP differ considerably; for example, the Maxc system employs shallow variable binding, while the Alto and Dorado systems employ deep binding. The translation from ByteLap to machine code is straightforward.

The structure of ByteLap

The ByteLap intermediate code generated by the compiler can be viewed as the instruction set for an abstract stack machine. The format of ByteLap is described here to simplify subsequent discussion of optimizations. There are 15 opcodes, each of which has some effect on the state of the linear temporary value stack. The instruction set is:

(VAR **var**)	Push the value of the variable **var** on the stack.
(SETQ **var**)	Store the top of the stack into the variable **var**.
(POP)	Pop the stack (i.e., throw away the top value and decrement stack depth by one).
(COPY)	Duplicate (push again) the top of the stack.

(CONST val)	Push the constant **val** on the stack (**val** may be of any LISP data type, e.g., an atom or a number.)
(JUMP tag)	Jump to the location **tag**.
(FJUMP tag)	Jump to the indicated location if top-of-stack is NIL, otherwise continue. In either case, pop the stack.
(TJUMP tag)	Similar to FJUMP, but jump if top-of-stack is non-NIL.
(NTJUMP tag)	Similar to TJUMP, but do not pop if it jumps. This is useful when a value is tested and then subsequently used.
(NFJUMP tag)	Analogous to NTJUMP.
(FN n fn)	Call the function **fn** with **n** arguments.
(BIND (v_1...v_n)(n_1...n_k))	Bind the variables v_1,...,v_n to the **n** values on the top of the stack. Also bind the variables n_1,...,n_k to NIL. All bindings are done in parallel. Remember the current stack location.
(UNBIND)	Save the current top of stack. Throw away any other values on the stack since the last (stacked) BIND, and undo the bindings of that BIND. Re-push on the stack the saved value. This is used at the end of PROG or LAMBDA expressions whose value is used.
(DUNBIND)	Similar to UNBIND, but do not restore the value.
(RETURN)	Return top-of-stack as the value of the current function, throwing away any other values on the stack.

Note that a given ByteLap opcode could have one of several different translations in the actual code executed. For example, both the Dorado and Maxc implementation have a separate opcode for pushing NIL, in addition to a more general constant opcode. The final code generation phase transforms the (CONST NIL) ByteLap instruction into the appropriate opcode. Operations such as arithmetic or CAR are encoded as FN calls, even though the instruction sets have specialized instructions to perform those operations. The assemblers distinguish between the built-in operations and those that must actually perform external calls; the compiler and the optimization phase do not care. Furthermore, a sequence of ByteLap instructions can assemble into a single machine instruction; for example, both instruction sets have instructions which can do a SETQ and a POP in the same instruction. These are easily detected with a short look-ahead during code generation.

III. Compiler Optimizations

One of the most important ground rules for the optimization phase has been that all optimizations are conservative: they must not increase either code size or running time. Only optimizations which experience has shown to be useful are described here.

The statistics given in the text below were obtained as a result of compiling a total of about 2200 functions, producing 65000 bytes of object code. Numbers in <angle brackets> in the text indicate the number of times that a given optimizing transformation or technique was applicable.

Optimizations during code generation

A few optimizations are performed during the initial code generation phase. In particular, the compiler keeps track of the execution context of any given expression (similar to many other LISP compilers we know of). Thus, in the recursive descent of the S-expression definition, the flag **effect** is set if the current expression is being compiled for effect only, and the flag **return** if the value is being returned as the value of the entire function.

Remove no-effect constructs when compiling for effect <162>

Compiling a variable or constant for effect results in no code generated. A call to a function with no side effects merely causes its arguments to be evaluated for effect: for example, a macro might expand into (CAR (RPLACA X Y)), which if executed for effect only performs the RPLACA, but if the value is used will return the value stored.

Remove extraneous POP <2035>

Knowledge of **return** context is used to omit extraneous POP instructions, since unused values can be left on the stack to be swept away when the frame is released by a (RETURN). For example, in the function

```
(LAMBDA (X) (PRINT X) (TERPRI))
```
the first pass emits
```
(VAR X) (FN 1 PRINT) (FN 0 TERPRI) (RETURN)
```
rather than
```
(VAR X) (FN 1 PRINT) (POP) (FN 0 TERPRI)
(RETURN).
```
The compiler also uses **return** context to eliminate extraneous JUMPs after arms of a conditional to the end of the conditional code (each arm of the conditional is compiled in **return** context, which will cause it to be terminated by a (RETURN) opcode).

The compiler also removes tail recursion in **return** context <36>. In addition, constant folding is done in the first pass for functions which are constant on constant arguments (e.g. EQ and arithmetic opcodes) <34>. Constant folding is done after the code for each argument is generated, so that constant detection can be achieved by looking for CONST opcodes, rather than pre-expansion of macros.

Post-optimizations

The second pass of the compiler consists of several local transformations on the generated ByteLap code which are tried repeatedly in turn until no further improvement can be made <6461 passes total, including the final unsuccessful pass on each function>. While the compiler contains many transformations, empirical results of compiling a large number of files show that the following transformations are the most useful—we have excluded transformations which were rarely effective. For each transformation we give its name, a symbolic version of it, a brief discussion, and an example in which the optimization would be effective.

COPY introduction <1018>

val val —> **val** (COPY)

This transformation reduces neither code size nor execution time; however, it often enables other optimizations. The **val** opcodes can be two identical CONST or VAR opcodes, or a SETQ followed by a VAR with the same variable. For example, the expression
```
(FOO (SETQ X (FUM)) X)
```
compiles to
```
(FN 0 FUM) (SETQ X) (VAR X) (FN 2 FOO)
```
which gets transformed to
```
(FN 0 FUM) (SETQ X) (COPY) (FN 2 FOO).
```

Variable duplication <1137>

(SETQ **var**) (POP) (VAR **var**) —> (SETQ **var**)

This transformation occurs frequently after assignments. For example, the expressions
```
(SETQ X Y) (COND (X (FN)))
```
compiles to
```
(VAR Y) (SETQ X) (POP) (VAR X)
(TJUMP L1) (FN 0 FN) L1:
```
which transforms into
```
(VAR Y) (SETQ X)
(TJUMP L1) (FN 0 FN) L1:
```

Dead assignment <661>

(SETQ **var**) *{no subsequent use of* **var***}* —> ()

The compiler scans ahead a short distance for either a (RETURN) or subsequent (SETQ **var**) with no intervening instruction which either uses (VAR **var**) or else calls a function which might see the binding of **var**. For example, after the examples in both *COPY introduction* and *Variable duplication*, the assignment to X might well be "dead", and the (SETQ X) removed.

Unused push <734>

val (POP) —> ()

Although the first pass avoids generating values followed by POP by the **effect** mechanism, enough instances arise where subsequent optimizations uncover unused values to make this transformation worthwhile during the post-optimization phase. **val** can be a CONST, VAR, or COPY. In addition, if **val** is a (FN **n fn**), where **fn** is a side-effect free function, it is replaced by **n** (POP)s.

Merge POP with DUNBIND <105>

(POP) (DUNBIND) —> (DUNBIND)

This simple transformation takes advantage of the fact that the DUNBIND opcode implicitly pops any values left on the stack since the last BIND.

JUMP OPTIMIZATIONS

Vacuous jump <1033>

(JUMP **tag**) **tag:** —>
(cJUMP **tag**) **tag:** —> (POP)

While the first pass ByteLap generation explicitly deletes these <265 occurrences>, this transformation is useful to clean up after others. In the pattern, cJUMP is either TJUMP or FJUMP.

Invert sense of jump <488>

(FJUMP **tag1**) (JUMP **tag2**) **tag1:**
—> (TJUMP **tag2**)

This transformation can occur, for example, when there are explicit GO's in the source. For example, the expression

```
(COND (X (GO LABEL1)))
```
compiles to
```
(VAR X) (FJUMP L1) (JUMP LABEL1) L1:
```
which transforms into
```
(VAR X) (TJUMP LABEL1) L1:
```

COPY introduction for TJUMP <241>

val (NTJUMP **tag**) **val**
—> **val** (COPY) (TJUMP **tag**)

This transformation notes that, whether or not the JUMP is taken, the value **val** will remain on the stack. The transformation is effective for both NTJUMP and NFJUMP. Note that **val** will be NIL in one of the cases.

JUMP code in-line <457>

(JUMP **tag**) ... **tag:** *{code}* —> *{code}* ...

This transformation moves the entire segment *{code}* in line only in the situation where the JUMP is the only way of reaching **tag**.

Jump-through <2259>

(jump tag) ... **tag:** (JUMP **tag2**)
—> **(jump tag2)** ...

One of the most common transformations in the compiler occurs when the target of a jump is itself a jump instruction. For example, the code generated for
```
(COND (A B) (T C))
```
is:
```
(VAR A) (FJUMP L1) (VAR B) (JUMP L2)
L1: (VAR C) L2:
```
If the variable B is replaced by a COND clause, the target of the jump at the end of that CONDs second clause would itself be a jump instruction. The **jump** in the pattern above can be any of the four jump opcodes. For example,
```
(COND (A B) (T (GO TAG))
```
would result in the fragment:
```
(VAR A) (FJUMP L2) ... L2: (JUMP TAG)
```
which can be transformed into
```
(VAR A) (FJUMP TAG) ...
```

Unreachable code <1670 occurrences, 1784 instructions>

(JUMP **tag**) *{code}* —> (JUMP **tag**)

The code after a JUMP or RETURN which is not itself jumped to can be deleted. The first pass avoids generating any constructs of this form, but such situations can be generated by other transformations. For example, in both preceding examples, the code at L2 might well be unreachable and deleted.

NTJUMP introduction <610>

val (TJUMP **tag**) ... **tag: val**
—> **val** (NTJUMP **tag+1**) ...

This optimization is essentially *COPY introduction* across jumps. For example,
```
(PROG NIL
 LP (FOO X)
    (COND ((SETQ X (CDR X)) (GO LP)))
    ...)
```
results in
```
LP: (VAR X) (FN 1 FOO) (POP) (VAR X)
(FN 1 CDR) (SETQ X) (TJUMP LP)...
```
which is then transformed to
```
LP: (VAR X) LP1: (FN 1 FOO) (POP)
(VAR X) (FN 1 CDR) (SETQ X)
(NTJUMP LP1) ...
```

NTJUMP introduction with code movement <506>

val (FJUMP **tag**) **val** *{code1}* ... **tag:** *{code2}*
—> **val** (NTJUMP **tag2**) *{code2}* **tag2:** *{code1}*

This transformation is a variation of *NTJUMP introduction* where it is necessary to move code around. The two code sequences *{code1}* and *{code2}* must end with a JUMP or a RETURN. Note that this transformation moves the entire segment of code *{code2}* inline. For example, the expressions
```
(COND (X (FN1 X)) (T (FN2) (GO LAB))
```
compile to
```
(VAR X) (FJUMP L1) (VAR X) (FN 1 FN1)
(JUMP L2) L1: (FN 0 FN2) (JUMP LAB) L2:
```
which gets transformed to
```
(VAR X) (NTJUMP L3) (FN 0 FN2)
(JUMP LAB) L3: (FN 1 FN1) (JUMP L2) L2:
```

Jump to NIL/POP <834>

```
(FJUMP tag) ... tag: (CONST NIL)
—> (NFJUMP tag+1)
(NcJUMP tag) ... tag: (POP)
—> (cJUMP tag+1)
```

The pattern NcJUMP stand for either flavor of N-conditional jump. In the first situation, the NIL which is being found by the FJUMP may be logically distinct from the NIL after tag. For example, the expression
```
(COND (A ...) (T (MYFN NIL))
```
compiles as
```
(VAR A) (FJUMP L1) ...
L1: (CONST NIL) (FN 1 MYFN)
```
which is transformed into
```
(VAR A) (NFJUMP L2) ...
L2: (FN 1 MYFN).
```

The second form of the transformation normally occurs only after other transformations, where a conditional, originally thought to be executed for value, does not need the value being preserved by the NcJUMP.

Removal of loop variables <679>

```
(SETQ var) (POP) (JUMP tag)
... tag: (VAR var)
—> (SETQ var) (JUMP tag+1)
```

This transformation is common in loops. For example,

```
(PROG NIL LP (PROCESS X)
(SETQ X (NEXT X)) (GO LP))
```

compiles as

```
LP: (VAR X) (FN 1 PROCESS) (POP)
(VAR X) (FN 1 NEXT) (SETQ X) (POP)
(JUMP LP)
```

This transforms to:

```
LP: (VAR X) LP1: (FN 1 PROCESS) (POP)
(VAR X) (FN 1 NEXT) (SETQ X) (JUMP LP1)
```

Cross jumping <1721 occurences>

{code} (JUMP **tag**) ... *{code}* **tag:**
—> (JUMP **tag2**) ... **tag2:** *{code}*

This frequent transformation improves code space with no effect on running time. For example, the expression

```
(COND (A (FOO X)) (T (FOO Y))
```

compiles as

```
(VAR A) (FJUMP L1) (VAR X) (FN 1 FOO)
(JUMP L2) L1: (VAR Y) (FN 1 FOO) L2:
```

The instruction before (JUMP L2) is identical to the instruction before the label L2, and so this can be transformed into

```
(VAR A) (FJUMP L1) (VAR X) (JUMP L3)
L1: (VAR Y) L3: (FN 1 FOO)
```

Jump copy test <733>

val fn1 (jump tag) val ... tag: val
—> **val** (COPY) **fn1 (jump tag+1)**

In this transformation, **fn1** is a "clean" function of one argument, e.g., (FN 1 LISTP) or (FN 1 CDR), or even (CONST **val**) (FN 2 EQ). In this case, "clean" means that the function cannot change the value of **val**. For example, the expression:

```
(COND ((LISTP X) (CAR X))
      ((NUMBERP X) (ADD1 X)))
```

results in the fragments

```
(VAR X) (FN 1 LISTP) (FJUMP L1) (VAR X)
... L1: (VAR X) (FN 1 NUMBERP) ...
```

which transforms into

```
(VAR X) (COPY) (FN 1 LISTP) (FJUMP L2)
... L2: (FN 1 NUMBERP)...
```

Return optimizations

Return merge

```
(TJUMP tag) {code} (RETURN)
  ...tag2: {code} (RETURN)
—> (FJUMP tag2) ... tag2: {code} (RETURN)
```

This is an effective code transformation which can merge completely unrelated (with regard to flow-of-control) return sequences. It does not affect speed, only space. *Return merge* is unique in not preserving the normal invarient that stack-depth is constant at any location in the code. Normal code generation only creates sequences of instructions where the stack-depth at any location is static; all other transformations preserve that property. However, the two occurrences of *{code}* in the pattern need not be at the same stack-depth, and thus, stack-depth would be ambiguous after **tag2**. This is important if the target machine language is dependent upon stack depth in the translation from ByteLap, as is the case with the Maxc instruction set. *Return merging* must be disabled if the two *{code}* sequences occur at different stack depths, and if *{code}* contains any stack-level-sensitive operations.

Needless POP before RETURN <590>

(POP) **val** (RETURN) —> **val** (RETURN)

This transformation is attempted only after it is known that there is no opportunity for *Unused push.* In addition to removing POP opcodes, this transformation also removes DUNBIND and UNBIND opcodes in the same position (except when **val** is a variable which was bound in the frame corresponding to the UNBIND or DUNBIND).

Unused variable in BIND <580>

```
(BIND ... (.. var ..)) {var not used}
—> (BIND ... (.. ..))
(BIND (.. var) ...) {var not used}
—> (POP) (BIND (..) ...)
```

This transformation eliminates binds of local variables which are not used. Only the last variable bound to a value can be so removed, because of the difficulty of inserting a POP at the appropriate place back in the instruction stream. (This is an example where source level transformation might be better way of doing optimization. Unfortunately, the last use of a variable is often removed by *COPY introduction*, which has no analogue in source code transformations.) To detect unused variables, the compiler scans the code linearly for uses of each variable in every BIND. For example, the expression

```
(PROG (X) (SETQ X (FUM)) (FOO X X))
```

compiles into

```
(BIND () (X)) (FN 0 FUM) (SETQ X) (POP)
(VAR X) (VAR X) (FN 2 FOO)
```
which, after several transformations, turns into
```
(BIND () (X)) (FN 0 FUM) (COPY)
(FN 2 FOO).
```
Since X is no longer used, it can be eliminated. Note that this transformation is not applicable to special variables (variables which can be referenced freely by functions called from this one, e.g., FUM and FOO).

Unused BIND <2035>

```
(BIND (v1 ... vm) (vm+1 ... vn)) (VAR v1) ...
(VAR vm) {no other mention of v1...vm}
—> (CONST NIL) {n-m times}
```

<Of the 2035 occurrences, 440 eliminated BINDs which were generated in the compilation of mapping functions.> This transformation eliminates BINDs when the variable list is empty or when the variables bound are only mentioned, in order, immediately following the BIND. When this transformation is made, the compiler must also find all corresponding DUNBIND's for this frame and turn them into the appropriate number of POP's. In addition, for every UNBIND the stack level must be exactly one greater than it was at the BIND. If so, the UNBIND can simply be deleted; if not, this transformation cannot be made. Note, however, that where a PROG or LAMBDA expression is the value returned by a function, no UNBIND or DUNBIND opcodes are generated. For example, the expression
```
((LAMBDA (X) (FOO X X)) (FUM))
```
compiles into
```
(FN 0 FUM) (BIND (X) ()) (VAR X)
(VAR X) (FN 2 FOO)
```
which, after *COPY introduction* and *Unused BIND* can be transformed into
```
(FN 0 FUM) (COPY) (FN 2 FOO).
```

CONCLUSIONS

Because our instruction sets are so well suited to the LISP language, it is possible to write quite simple non-optimizing compilers for our LISP machines. In fact, we have written a simple but usable compiler in less than three pages of LISP code. However, local transformations can have an important impact on code space and running time.

As in production systems, the choice of order of application of transformations can affect the results. Without effectively trying all possible orderings, one transformation can prevent a better one from being used. In successive transformations made on a sample of user LISP programs, however, we have not observed this to be a major problem.

The programs our compiler generates are still not optimized, in the strict sense of that term. A sample of user LISP programs which were "hand optimized" show that code size could be compressed by as much as an additional 15% in some cases, with no speed penalty. However, the transformations involved seem to require either much special-case pattern matching or else transformations which temporarily reduce either space or speed. As usual when employing "hill-climbing" algorithms, by requiring that all transformations we employ are strict improvements, we occasionally find local optima which prevent better solutions from being found.

Optimizing on a simple intermediate language is quite effective. Many of the transformations made are not expressible as source language transformations (e.g., the COPY operator has no direct counterpart in the LISP language). Those that would be easier to express as source transformations are often enabled by transformations which have no direct analogue. Peephole optimizers working on more complex assembly languages must be aware of more special cases, because there are many more kinds of operations.

BIBLIOGRAPHY

[Burton et al, 1980]
Richard R. Burton, Larry M. Masinter, Daniel G. Bobrow, Willie Sue Haugeland, Ronald M. Kaplan and B. A. Sheil, "Overview and implementation status of DoradoLisp", to appear, 1980.

[Deutsch, 1973]
L. Peter Deutsch. "A Lisp machine with very compact programs". *Proceedings of the Third International Joint Conference on Artificial Intelligence*, Stanford, 1973.

[Deutsch, 1978]
L. Peter Deutsch. "Experience with a Microprogrammed Interlisp System". *IEEE Micro-11 Conference,* Asilomar, 1978.

[Deutsch, 1980]
L. Peter Deutsch. "ByteLisp and its Alto Implementation". *1980 Lisp Conference,* Stanford, August 1980.

[Fiala, 1978]
Edward R. Fiala. "The Maxc Systems". *IEEE Computer*, **11:5**, May 1978.

[Griss & Hearn, 1979]
Martin L. Griss and Anthony C. Hearn. "A Portable LISP Compiler". *Report UCP-76, Department of Computer Science, University of Utah*, June 1979.

[Steele, 1978]
Guy L. Steele. "RABBIT: A Compiler for SCHEME (A Study in Compiler Optimization)". *Report AI-TR-474, MIT Artificial Intelligence Laboratory*, May 1978.

[Teitelman et al, 1978]
Warren Teitelman *et al. INTERLISP Reference Manual.* Xerox Palo Alto Research Center, October 1978.

[Urmi, 1978]
Jaak Urmi. "A machine independent LISP compiler and its implications for ideal hardware". *Linkoping Studies in Science and Technology Dissertations No. 22*, Linkoping, Sweden, 1978.