The Implementation of *PC Scheme*

David H. Bartley
John C. Jensen

Computer Science Center
Texas Instruments Incorporated
P.O. Box 226015, M/S 238
Dallas, Texas 75266

Abstract

PC Scheme is a compiler-based implementation of Scheme for PC-class machines. The compiler generates code for an idealized virtual machine which is emulated with threaded code techniques. The design has traded off the requirements of space and speed effectively, resulting in one of the fastest PC-class LISP systems known to the authors.

1 Introduction

Scheme [RRRS 85] is a lexically scoped dialect of LISP which was originally developed at MIT [Steele 78a] and has subsequently been used extensively for research, education, and application programming. Like COMMON LISP, Scheme features lexical scoping and first-class functions. Unlike COMMON LISP, Scheme supports continuations as first-class objects and is properly tail-recursive. Many Scheme dialects, including ours, have also incorporated environments and engines as first-class objects.

Texas Instruments has been interested in Scheme for some time as a vehicle for experiments in compilation techniques and the design of architectures for symbolic computing. *PC Scheme*, a commercial by-product of this research, was created to support the development of such AI packages and products as TI's Personal Consultant™ and Arborist.™ Since TI and IBM® PCs with Intel® 8088 and 80286 processors and typical memory capacities of 512K bytes were to be used as both the development and target machines for these efforts, *PC Scheme* was required to be both fast and small.

Following Powell [Powell 84], the design philosophy of our system is "best simple." Whenever possible, our design decisions favored the simplest and most cost-effective alternatives available. Uncommon features were made to "pay their own way," while common ones were streamlined as much as possible. The result is a compilation strategy and virtual machine architecture that are highly tuned to our purposes.

2 The Virtual Machine (VM)

The key to a successful LISP implementation on a small machine is careful design of the environment in which programs execute. The selection of data representations is particularly critical because of the pervasive use of type dispatching in program code and the garbage collector. Although our requirement for speed dictated a compiler-based approach, the expected bulkiness of true native code on 8-bit processors was excessive for the small memories available. We chose instead to emulate a virtual machine representing an idealized Scheme architecture. Our VM simplifies the compiler considerably yet is surprisingly fast.

2.1 VM Implementation

Threaded code [Bell 73] is a well-known implementation technique for the emulation of language-specific architectures. It combines compact encoding of programs with rapid execution. Our byte-level encoding is similar to most Pascal P-code implementations: the byte pointed to by the emulated "program counter" is used as an index into a table of machine language subroutine addresses. Each of these machine language routines performs an operation, optionally conditioned by the contents of subsequent "operand" bytes in the emulated VM instruction stream, and then executes the NEXT sequence shown below to fetch and execute the next VM instruction.

```
xor   AX,AX               ; clear AX
lods  byte ptr ES:[SI]    ; get next
mov   BX,AX               ;  opcode
shl   BX,1                ; make word index
jmp   op_table+[BX]       ;  and jump
```

The emulation overhead for this NEXT sequence represents about 16% of the total execution time for an average mix of VM instructions.

As shown in Figure 1, the VM emulator and that part of the runtime support which is written in assembly language and C consume 137,500 bytes of memory. The rest of the system is written in Scheme. The first 76,700 bytes of Scheme code are loaded into the heap during system initialization. The rest of the system code is autoloaded as needed.

2.2 VM Architecture

Unlike P-code, which is a stack-based architecture oriented towards strongly typed languages, our virtual machine architecture is register-based and emphasizes the requirements of LISP-like languages. The choice of a register-based model was motivated by the desire to investigate compilation techniques for such register-based machines as the Motorola 68000, the National/TI 32000, and Berkeley's SOAR [Ungar 84], a tagged Reduced Instruction Set Computer (RISC). The VM architecture largely follows the tenets of the RISC philosophy but integrates most low-level run time features and many commonly used primitive Scheme functions as VM instructions. This is a valuable aid to increased performance for an emulated VM, though less appropriate for an actual implementation in hardware.

The heart of the *PC Scheme* VM is a set of 64 general purpose registers, R0-R63. Arguments to functions are passed in registers where they remain available for use until no longer needed or until another function call occurs. Except as described in section 2.5, all user variables and temporaries reside in the registers so they may be conveniently accessed by VM instructions. There are also special purpose registers for the global and fluid environment pointers, the current code block pointer, and the current program counter.

The VM is a conventional load/store architecture with most instructions operating directly on the registers. For fast decoding, each VM instruction has its own specific operand format and its handler is responsible for fetching and operating upon the operand bytes. Register operands are encoded as the register number scaled by 4, the width in bytes of an entry in the underlying table of VM "registers." Other operand types include immediate values, constant table indices, stack frame offsets, and branch displacements. A typical VM instruction is illustrated below.

```
Instruction:   ADD R5,R7        ; R5 := R5 + R7

Byte 1:   Opcode = ADD,     encoded as 80
Byte 2:   SD operand = R5,  encoded as 5*4 = 20
Byte 3:   S operand = R7,   encoded as 7*4 = 28
```

Other virtual machine architectures for LISP include variations on the MIT LISP Machine [Greenblatt 84] and SPICE LISP [Wholey 84]. RABBIT [Steele 78b], MACSCHEME, and an unnamed "small Scheme implementation" at MIT [Schooler 84] propose VMs specific to the needs of Scheme. Of these, only RABBIT is register-based.

Description		*Size (bytes)*
native code:	program	72,000
	static data	23,700
	runtime stack	12,300
	C's heap	29,500
	(subtotal)	137,500
Scheme:	standard system	76,700
Autoloaded Scheme:	Edwin editor	109,000
	SCOOPS	18,700
	structure editor	8,200
	miscellaneous	30,900
	total	381,000

Figure 1: Memory space allocation

The others employ simple stack models, with instructions typically operating on implicitly specified operands at the top of the stack. Like *PC Scheme*, MACSCHEME and the MIT "small Scheme" are bytecoded implementations (although the MIT design was never completed). The RABBIT VM is actually a subset of LISP with global variables serving as VM registers. SPICE and the LISP Machines have microcoded implementations.

These LISP-oriented architectures have several features in common. Generally, instructions operate only on tagged data and validate the types of their operands. Frequently used operations like `MEMBER`, `SYMBOLP`, and `CONS` are implemented as instructions. Except for RABBIT, they have been designed with reduced compiler complexity as a specific goal and are oriented primarily to supplying the needs of software developers rather than to achieving the highest possible performance.

Conversely, *PC Scheme* is an experiment in the design of fast delivery vehicles for AI programs. This has lead us to consider optimizing compilers and RISC-like, register-based VMs for the same reasons that they are of interest to designers of conventional processors. Explicit operand addressing significantly reduces the number of pushes, pops, and other data movement instructions generated, resulting in shorter object programs compared to traditional stack VMs. Performance is further enhanced by the high average level of work performed by each VM instruction compared to the overhead of the NEXT operation. Much of this work is dedicated to software tag checking, however—an operation that can be performed much more effectively with suitable hardware. We believe that our VM is a suitable architecture for a RISC implementation.

2.3 Memory Model and Data Representations

PC Scheme memory consists of a linear address space subdivided into logical pages to support a BIBOP (big bag of pages) tagging mechanism. An object is represented by a three byte pointer. The page number and page displacement are stored in the top byte and bottom two bytes, respectively.

The default page size is computed automatically as a function of available physical memory each time *PC Scheme* is loaded and varies between 3072 and 5136 bytes. A pointer is mapped into its associated physical memory address by looking up the base (segment) address for the page in a table and adding its displacement component. This mapping permits pages to be placed arbitrarily in physical memory. A given page may be made larger than the default value as needed to accommodate extremely large objects such as code blocks and arrays.

(Although the released product is restricted to 640KB physical memory (768KB for TI PCs), experimental versions exploiting extended and expanded memories execute somewhat more slowly due to address mapping complications.)

The type of a referenced object is determined by inspecting a table of type attributes indexed by the number of the page in which it resides. A simple bit test or byte comparison suffices for all of the data type predicates and to distinguish the various representation formats. An index byte is also associated with each page entry to allow fast dispatching on type. Although each page holds objects of only one type, all objects except list cells, fixnums, and characters have redundant type headers to aid the sweep phase of garbage collection.

The *PC Scheme* data types implemented as heap-allocated objects are list cells, arbitrary-precision integers, 64 bit IEEE floating point numbers, symbols, strings, vectors, continuations, functions, code blocks, I/O ports, and environments.

Integers in the range $-16,384$ to $+16,383$ are represented as immediate values; a page number is reserved as a 'fixnum' tag and the numeric value is stored in the displacement field of the reference. Character objects are represented similarly.

2.4 Garbage Collector

Long pauses for garbage collection are confusing and often distressing to end-users of applications written in many PC LISPs. Although "mark and sweep" is acceptably fast on a nominal 512KB memory, much slower compaction algorithms are often employed to avoid the effective loss of space to fragmentation. *PC Scheme* uses a mixed strategy. When a memory allocation request cannot be satisfied from a free memory pool, a mark and sweep garbage collection is performed. If this doesn't recover enough memory, a memory compaction phase is run to coalesce small fragments of free memory into larger blocks.

The memory compaction operation copies referenced objects from pages which are the least full to those which are the most full. Thus, it moves the fewest objects possible to accomplish dense packing of data. Fewer than half of the objects in the heap are moved, even in the worst case (typically much fewer since system code is packed densely by the initial load and is never moved). After the objects have been moved, a sequential pass is made through memory to relocate all pointers to moved objects.

The mark and sweep phase causes a pause in execution of 0.5 to 2.5 seconds for a 512KB 80286 system. The longest pause when compaction is needed is about six seconds. We consider this delay more acceptable than the VM overhead that a real-time collector would incur.

2.5 Control Model

Unlike most LISPs, Scheme provides "escape functions" called *continuations* which may be called any number of times; the invocation need not be within the dynamic scope of the binding of the continuation. This behavior rules out the use of a simple contiguous control stack for the general case, so a judicious use of heap-allocated structures is needed. Likewise, the existence of first-class function objects, called *closures*, dictates heap allocation of some lexical environment bindings.

The control stack holds the state needed to execute a continuation. Since the compiler is generally unable to determine which control frames will be retained indefinitely, it optimistically allocates them in a contiguous stack area. When `CALL-WITH-CURRENT-CONTINUATION` is invoked, the stack is copied into the heap to make a continuation object. To reduce the worst-case cost of this copying and to permit arbitrary growth of the stack, a small buffer is used to hold the topmost stack frames. When this buffer overflows, all frames but the currently active one are copied into a continuation object in the heap. Thus, a continuation is represented by a chain of such control stack segments.

The compiler identifies those variables which have indefinite extent because they are freely referenced from closures and arranges for them to be heap-allocated at run time. Thus, the data structure stored in a closure object does not reference the stack. Temporaries and variables which are not accessed freely from any closure are allocated to registers or to locations in the stack.

Smalltalk-80™ procedure activation environments (contexts) correspond closely to Scheme closures and continuations. [Deutsch 84] and [Suzuki 84] report that 85% or more of Smalltalk-80 contexts may be allocated in a stack-like manner since operations that require indefinite retention of contexts occur infrequently. Deutsch allocates contexts created by normal sends on the stack. When a reference to a context is created (making it an object), space is allocated in the heap but the context remains cached in the stack as long as possible. Heap-allocated contexts must be returned to the stack before they may be executed.

Suzuki also has multiple representations for Smalltalk-80 contexts but permits a context to be executed in the heap. As with *PC Scheme*, a small stack buffer behaves like a cache for heap-allocated environments and limits the time taken to save it to the heap.

Both Smalltalk-80 implementations rely on "lazy" allocation methods at run time rather than static analysis at compile time. We adopt the same approach for continuations because `CALL-WITH-CURRENT-CONTINUATION` has dynamic effects on the retention of environments. We prefer static analysis of free variable references from closures, however, since it allows individual variables to be selectively heap-allocated. Although we make no attempt to delay transferring such variables from the stack (or registers) to the heap, it would be possible to do so.

3 The Compiler

PC Scheme relies solely on its compiler for the evaluation of Scheme programs. An interpreter was considered but was not implemented for the following reasons:

- Providing both a compiler and an interpreter takes more space than having either alone, especially when integrated with a comprehensive system maintenance facility.
- It can be difficult to guarantee absolute consistency between two different evaluators. Many users prefer to avoid certain subtle errors by debugging their programs in their final compiled form rather than in interpreted form, despite the richer development environment that interpreters make possible.
- The debugging aids provided with our compiler suffice for most programmers' needs.
- Our compiler's speed is adequate for interactive development, particularly when program modules are separately compiled and placed in "fast-load" format after they are tested.

The phases of the compiler are: macro expansion and alpha conversion, local optimization, closure and environment analysis, code generation, peephole optimization, and assembly. The relative amount of time spent in each phase is shown in Figure 2. A separate utility is available for converting files in compiled form into "fast-load" form.

3.1 Local Optimization

As with most sytems with architectures tuned towards the needs of LISP, the performance of *PC Scheme* depends more on the "fit" of the VM to the needs of the language than it does on extensive compiler optimization. The local optimization phase, called the *simplifier*, is designed primarily to cope with anticipated inefficiencies in macro-expanded expressions and to shape the code to fit the preferences of the code generator. The peephole optimizer is quite effective in overcoming the lack of more intelligent optimizations elsewhere; it typically reduces the program size by a quarter.

A Beta-conversion optimizer similar to the one in RABBIT [Steele 78b] was tried and found to be successful but proved too slow for our purposes. However, the compiler produces surprisingly good code in most instances due to careful tuning of macro definitions, the simplifier, the code generator, and the peephole optimizer. This can be seen in the sample code for the Fibonacci function in Figure 3.

Compiler Phase	*Time*
macro expansion	18 %
local optimization	5 %
closure and env. analysis	6 %
code generation	23 %
"peephole" optimization	29 %
assembly	19 %

Figure 2: Relative time spent in each compiler phase

3.2 Closure and Environment Analysis

We have found that two related optimizations, avoiding function closures and allocating variables to VM registers instead of the heap, have considerable impact on the size and speed of compiled Scheme programs. Following [Steele 78b], we analyze each program to determine which `LAMBDA` expressions require that closure objects be created at run time because they are treated as data ("funargs"). All lexical variables which must be retained because they are accessible from such funargs are then marked for heap allocation in environment objects. The heuristic algorithms used to identify these funargs and variables are fast but somewhat pessimistic.

Conventional programming languages seldom create a closure for a procedure at run time because all calls to the procedure can generally be found through static analysis by the compiler. This is often the case in Scheme programs as well and such procedures are well worth identifying. A related optimization is to determine when a given call to a closed function can jump directly to the appropriate code location rather than indirectly through the closure object. Thus, we distinguish whether a `LAMBDA` expression is closed from whether a given call to it uses the closure object.

Closure analysis is performed as a simple walk through the program tree, marking `LAMBDA` expressions to be closed whenever any of the following occur:

- The `LAMBDA` expression appears as a funarg; that is, other than in the function position of an application or as the bound value in a `LETREC` pair.
- The value of the `LAMBDA` expression is bound by a `LETREC` to some variable and that variable appears as a funarg or its value is reassigned by `SET!`.

In addition, functions taking an arbitrary number of arguments are always closed. Such functions have argument lists of the form `(A B . C)`, where `C` is a so-called *rest* argument. This feature was added late in the development of *PC Scheme* and is not supported optimally.

Having determined which functions must be closed, the compiler next identifies those lexical variables which have indefinite extent in order to allocate space for them in the heap. All other lexical variables may be allocated to registers or locations in the stack.

```
(define (fib n)
  (if (<? n 2)
      n
      (+ (fib (- n 1)) (fib (- n 2)))))

FIB
          LOAD       R3, (QUOTE 2)      ; r3 := 2
          LOAD       R2, R1             ; r2 := n
          <?         R2, R3             ; r2 := (<? n 2)
          JUMP       L5, NULL?, R2      ; (go L5) if (null? r2)
          EXIT                          ; return value in r1
L5
          PUSH       R1                 ; save n
          %+IMM      R1, (QUOTE -1)     ; r1 := (- n 1)
          CALL-OPEN  FIB, 0, 0, (1)     ; jsr to FIB
          PUSH       R1                 ; save (fib (- n 1))
          LOAD       R3, (STACK 0 0)    ; r3 := n
          LOAD       R1, R3             ; r1 := n
          %+IMM      R1, (QUOTE -2)     ; r1 := (- n 2)
          CALL-OPEN  FIB, 0, 0, (1)     ; jsr to FIB
          POP        R2                 ; restore (fib (- n 1))
          +          R1, R2             ; (+ (fib ...)(fib ...))
          EXIT                          ; return value in r1
```

Figure 3: Sample *PC Scheme* code for the Fibonacci function

A variable has indefinite extent if it is freely referenced from a funarg or from a function which is called directly or indirectly from a funarg. Rather than compute the transitive closure of function calls, we consider a lexical variable to have indefinite extent if *any* funarg is defined within its scope and it is freely referenced by *any* function. This assumes that any function may be called by any other.

(In retrospect, the decision not to compute the transitive closure of calls from funargs appears misguided. As shown in Figure 2, additional time spent in this phase would still be overshadowed by the costs of code generation, peephole optimization, and assembly.)

Variables are heap-allocated only if they must exist at run time—identifiers which are bound to open functions are never accessed as variables and so are not allocated any storage at run time. More precisely, a lexical variable is heap-allocated if it has indefinite extent and must exist at run time; it must exist at run time if it is modified by SET! or is initialized to some value other than an open function.

This analysis is complicated slightly by the existence in *PC Scheme* of the special form THE-ENVIRONMENT, which reifies the lexical environment at a point as a first-class data object. The appearance of THE-ENVIRONMENT in a variable's scope causes the variable to exist at run time with indefinite extent.

When the compiler is operating in debug mode (specified by the value of a flag variable), it skips most optimizations and marks all lambda expressions for closure and all variables for heap allocation. This preserves a closer correspondence between the object code and its source, allows every lambda expression to be traced, and permits debugger access to all variable bindings. In this mode, programs typically execute at about one fourth the normal speed.

3.3 Register Allocation

The code generator uses a stack-like approach to allocate variables and temps to VM registers. Registers R1 through Rn hold function arguments at function entry. As LET and LETREC expressions are entered, the next available registers in ascending numerical order are assigned to hold the new bindings. (Registers are not allocated to heap-allocated variables or to identifiers bound to open functions.) The temporary results from in-line expression evaluation normally occur at this simulated "top of stack," although most operations frequently access their operands directly without stack-like pushes and pops. The "peephole" optimizer performs copy propagation and targeting across entire blocks at a time, so redundant register moves are infrequent.

This straight-forward approach is an example of the "best simple" design philosophy. Its success is due to several factors:

- The availability of 64 general VM registers alleviates the need to minimize the total number of registers in use at any one time.
- The peephole optimizer does well at rectifying locally inept instruction selection and register allocation.
- Several common functions (e.g., MEMQ, +, NOT) are implemented as VM instructions, avoiding register saves and restores around out-of-line calls

- Tail-recursive calls do not require register saving.
- Many functions have high-probability execution paths that do not involve further function calling (e.g., `(lambda (n)(if (<? n 2) n ...))`).

Improved code quality at the cost of longer compilation times could be obtained with a global register allocation strategy coupled with more inter-procedural variable analysis. RABBIT [Steele 78b] employs a stack-like allocator similar to ours that works across procedure boundaries and would work well with our VM. [Murtagh 84] describes an algorithm for allocating environment frames and display entries so they may be shared among several mutually nonrecursive procedures. We are investigating the application of these ideas to a register allocator for Scheme.

3.4 Function Calls

The VM registers are a globally allocated resource and must be preserved across function calls. *PC Scheme* uses the "caller saves" protocol in order to reuse the registers for argument passing and so that only active registers are saved. The compiler minimizes the total number of pushes and pops by delaying them until the last possible moment. Thus, between out-of-line calls, saved variables are accessed directly from the stack instead of popped into registers and then accessed. Indeed, pops are delayed until the two branches of an IF expression are rejoined, since the stack height must be made consistent at such points. Tail recursive calls use the same argument passing mechanism but do not require register saving.

A key benefit to passing arguments in global registers is that subsequent calls with the same arguments in the same positions in an argument list may not require register shuffling. As a trivial example, the code generated for the function `(LAMBDA (A B) (FOO (+ A 1) B))` is simply the following.

```
%+IMM    R1,'1              ; Arg1 := (1+ A)
LOAD     R3, (GLOBAL FOO)   ; fetch closure
CALL-TR  R3                 ; Arg2 is unchanged
```

4 Performance

Table I compares *PC Scheme*'s execution speed against other PC-class LISP systems—Golden Common Lisp,® Large Memory (GCLISP LM™) version 2.1; IQLISP™ version 1.7; TLC™-LISP version 1.51; and MacScheme.™

Most of these benchmarks are taken from the Gabriel benchmark set [Gabriel 85] and were run without modification, except to account for differences in the LISP dialects. The 80286 runs were made on an IBM PC/AT™ with 640K bytes of system memory, except for GCLISP LM, which requires additional protected memory to run the interpreter (1.5MB) and compiler (3MB). MacScheme was tested on a 512K byte Apple® Macintosh.™

The numbers show the execution time in seconds averaged over three test runs in a clean, garbage collected, system. The first run of a series was retested whenever it appeared to be distorted by autoloading or other initialization effects.

The timings show that the compiled systems consistently outperform the interpreted ones and that *PC Scheme* is intermediate in performance between TLC-LISP, another "pseudo-coded" VM, and the native code version of GCLISP LM. These results confirm our expectations about our register-based virtual machine and threaded code emulator. We have also observed that our generated code is considerably more compact than both interpreted S-expressions and native object code. Our approach appears to be a good compromise between space and speed for small PCs, as intended.

Each system has its own unique strengths and weaknesses. We note that *PC Scheme*'s performance with integers larger than $\pm 2^{14}$ suffers on benchmarks with larger values; a benchmark that counts a million iterations of an expression is dominated by bignum arithmetic.

5 The Development Environment

The *PC Scheme* system includes an editor, debugger, and object-oriented programming facility called SCOOPS.

Lisp	*mode of*	*host*	BROWSE	DERIV	DIV2	DIV2	FACT	FIB	TAK
system	*execution*	*processor*			*iter*	*recur*	1000	20	18 12 6
PC Scheme	bytecode	80286	581.51	101.21	25.03	58.64	6.76	6.47	19.17
MacScheme	bytecode	68000	*(1)*	243.36	60.17	143.69	*(1)*	22.50	72.24
TLC-LISP	"P-code"	80286	743.45	146.34	88.00	110.47	*(4)*	18.46	55.11
GCLISP LM	native code	80286	248.33	39.46	16.50	29.85	*(3,4)*	2.04	6.65
TLC-LISP	native code	80286	*(2)*	*(2)*	40.85	77.81	*(4)*	12.56	34.00
GCLISP LM	interpreter	80286	2008.37	219.30	253.68	221.09	*(3,4)*	35.30	144.36
IQLISP	interpreter	80286	*(1)*	293.62	348.60	273.73	*(3)*	37.08	154.10
TLC-LISP	interpreter	80286	3107.76	369.74	325.62	*(3)*	*(4)*	49.38	162.67
Notes:	*(1)* This test was not run.								
	(2) Stack overflow during compilation.								
	(3) Stack overflow during execution.								
	(4) GCLISP LM and TLC-LISP restrict integer values to 32 bits.								

Table I. Comparative execution times for several PC LISPs

PC Scheme's editor is a substantially modified version of Edwin, an EMACS-style editor which was originally developed by the Scheme project at MIT. As a Scheme program itself, Edwin was easily integrated into our development environment, although it was necessary to cut its size considerably. Edwin's Scheme mode provides parenthesis matching, indenting commands, and the ability to evaluate an expression, region, or the entire buffer. Moving between the Scheme listener and the editor is facilitated by the ability to split the screen into separate windows for each.

Both the size and speed of the editor were considerable concerns at the onset of the port. The majority of the code size reduction came through careful selection of features to omit. Achieving acceptable speed hinged on identifying a few key functions which could be made VM "instructions."

Although *PC Scheme* does not provide extensive source-level debugging support, it has extensive trace and breakpoint facilities and an interactive Inspector with commands to display and manipulate call stack frames and lexical environments, edit variable bindings, trace back through a chain of procedure calls, and evaluate expressions in the environment of a breakpoint. All user-correctable errors trap to the Inspector.

SCOOPS is an experimental object-oriented programming system with multiple and dynamic inheritance based on first-class environments. Although it is similar in concept and syntax to the LOOPS [Bobrow 83] and Flavors [Weinreb 83] systems, the implementation of SCOOPS relies heavily on the features of the Scheme language.

Other development features include a pretty-printer, window system, color graphics, autoloading, structure editor, and a utility that converts compiled object files into fast-load format.

6 Concluding Remarks

PC Scheme has met our goals for high performance and compact code for AI applications on PC-class machines. The byte-encoded VM and relatively simple compiler appear to offer a better compromise in these respects than either traditional interpreters or native code compilers. Although we have emphasized performance over richness of the development environment, hundreds of users have found *PC Scheme* to be a productive and friendly system.

Our experience with *PC Scheme* has vindicated our selection of Scheme over COMMON LISP as our application language for small PCs. Scheme's radically simpler structure and absence of expensive features make it much easier to compile and execute efficiently. Unfortunately, it is harder to make many of COMMON LISP's expensive features "pay their own way." Moreover, COMMON LISP lacks first-class continuations, engines, and environments, which are important to many users.

The focus of our work is now shifting towards supporting both Scheme and COMMON LISP with a portable byte-code emulator. The principal difficulty for COMMON LISP, the immense size of a complete implementation, is ameliorated by our compact encoding. We see no fundamental problems in extending our VM to encompass multiple values, complex procedure calls, and other aspects of the language. Indeed, our results so far corroborate Steele's conjecture [Steele 78b] that Scheme is an excellent basis for compiling other languages—particularly other LISPs.

Acknowledgements

We would like to thank the following members of the CSC Computer Architecture Branch who assisted in the creation of PC Scheme: Rusty Haddock, Paul Kristoff, Don Oxley, Amitabh Srivastava, John Gateley, Mark Meyer, and Glen Slick. Keith Carlson, Terry Caudill, and Jennifer Hao were responsible for its release by our product group. Jan Stevens guided the development of the manuals. Bill Cohagan prototyped an experimental translator from Pascal to Scheme and experimented with COMMON LISP with the help of Ken Lewis.

References

[Bell 73] Bell, J. R., "Threaded Code." *Communications of the ACM*, XVI, (1973) pp. 370-372.

[Bobrow 83] Bobrow, D.G.; and Stefik, M.J.. *The LOOPS Manual*. Palo Alto, CA: Xerox Corporation, 1983.

[Deutsch 84] Deutsch, L. Peter, and Schiffman, Allan M. "Efficient Implementation of the Smalltalk-80 System." In *Conference Record of the 11th Annual ACM Symposium on Principles of Programming Languages*. January 1984, pp. 297-302.

[Gabriel 85] Gabriel, Richard P., *Performance and Evaluation of Lisp Systems*. The MIT Press, 1985.

[Greenblatt 84] Greenblatt, R. D., et al. "The LISP Machine," *Interactive Programming Environments*, D. R. Barstow, H. E. Shrobe, E. Sandewall, eds. McGraw-Hill, 1984.

[Murtagh 84] Murtagh, Thomas P. "A Less Dynamic Memory Allocation Scheme for Algol-like Languages." In *Conference Record of the 11th Annual ACM Symposium on Principles of Programming Languages*. January 1984, pp. 283-289.

[Powell 84] Powell, Michael L. "A Portable Optimizing Compiler for Modula-2." In *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction*. June 1984, pp. 310-318.

[RRRS 85] Clinger, W., ed. "The Revised Revised Report on Scheme." Massachusetts Institute of Technology AI Memo No. 848 (August 1985).

[Schooler 84] Schooler, Richard, and Stamos, James W. "Proposal for a Small Scheme Implementation."

Massachusetts Institute of Technology Laboratory for Computer Science Memo No. TM-267 (October 1984).

[Steele 78a] Steele, Guy Lewis, Jr., and Sussman, Gerald J. "The Revised Report on Scheme, a Dialect of Lisp." Massachusetts Institute of Technology AI Memo No. 452 (January 1978).

[Steele 78b] Steele, Guy L. "RABBIT: a Compiler for Scheme." Massachusetts Institute of Technology AI Technical Report No. 474 (May 1978).

[Suzuki 84] Suzuki, Norihisa and Terada, Minoru. "Creating Efficient Systems for Object-Oriented Languages." In *Conference Record of the 11th Annual ACM Symposium on Principles of Programming Languages*. January 1984, pp. 290-296.

[Ungar 84] Ungar, David, et al. "Architecture of SOAR: Smalltalk on a RISC." In *Proc. 11th Annual International Symposium on Computer Architecture*. 1984, pp. 188-197.

[Weinreb 83] Weinreb, D.; Moon, D.; and Stallman, R. *Lisp Machine Manual*. Cambridge, MA: Massachusetts Institute of Technology, 1983.

[Wholey 84] Wholey, Skef. "The Design of an Instruction Set for Common Lisp." In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*. August 1984, pp. 150-158.

Trademark Information