Experiences Creating a Portable Cedar

Russ Atkinson, Alan Demers, Carl Hauser, Christian Jacobi, Peter Kessler, and Mark Weiser
Xerox PARC
3333 Coyote Hill Road
Palo Alto, CA 94304

Abstract: Cedar is the name for both a language and an environment in use in the Computer Science Laboratory at Xerox PARC since 1980. The Cedar language is a superset of Mesa, the major additions being garbage collection and runtime types. Neither the language nor the environment was originally intended to be portable, and for many years ran only on D-machines at PARC and a few other locations in Xerox. We recently re-implemented the language to make it portable across many different architectures. Our strategy was, first, to use machine-dependent C code as an intermediate language, second, to create a language-independent layer known as the Portable Common Runtime, and third, to write a relatively large amount of Cedar-specific runtime code in a subset of Cedar itself. By treating C as an intermediate code we are able to achieve reasonably fast compilation, very good eventual machine code, and all with relatively small programmer effort. Because Cedar is a much richer language than C, there were numerous issues to resolve in performing an efficient translation and in providing reasonable debugging. These strategies will be of use to many other porters of high-level languages who may wish to use C as an assembler language without giving up either ease of debugging or high performance. We present a brief description of the Cedar language, our portability strategy for the compiler and runtime, our manner of making connections to other languages and the Unix* operating system, and some measures of the performance of our ‘‘Portable Cedar’’.

Introduction

Cedar is a large complicated language with many machine dependent constructs. Its original compiler was targeted for a single proprietary architecture, the D-machine [Lampson and Pier]. A large amount of Cedar code is in use (over 2 million lines). All of these constraints seemed to make a portable Cedar language, and a portable Cedar environment, almost impossible. We have a success story to report: it was not that bad. Furthermore, our success story is one with many lessons, both detailed and general, for others attempting to make portable versions of modern languages and environments.

One key lesson is that C [Kernighan and Ritchie] is a feasible portable intermediate language if you treat it as pure intermediate code. People complain that the original C++ implementation [Stroustrup] generates intermediate C which must be actually worked with by people (*e.g.* for

* Unix is a trademark of AT&T Bell Laboratories.

debugging); people worry that C intermediate code means inefficient final code (although there have been few measurements to support this, it remains folklore). Our generated C is machine dependent (along a few efficiency-driven dimensions like word length and byte order), very efficient, almost completely unreadable, and almost never seen by humans. We use unmodified Unix source debugging tools on Cedar-language source. We present measurements that our code is as efficient as directly compiled hand-written C code for both simple (*e.g.* dhrystone) and complicated (*e.g.* page rendering) programs.

A second key lesson is our technique of implementing modern language features in a portable language-independent and operating system-independent layer. Our experience is that a featureful language like Cedar (or, we conjecture, Smalltalk-80 [Goldberg and Robson] or Common LISP [Steele] or Modula-3 [Cardelli, *et al.*], *etc.*) need neither force its language requirements onto the depths of the operating system (as Cedar formerly did and Lisp-machine-style Lisps [Weinreb and Moon] still do), nor develop a thick insulating layer from the operating system (as most Unix LISPs do). Our approach to integrating Cedar into its base system is lightweight, and consists of several layers which together provide the language-independent features like garbage collection, exception handling, runtime types, and threads. We have run identical large Cedar applications on D-machines running the Cedar environment, on Sun SPARC and Motorola 68020 processors running SunOS, on 68020's running Mach [Accetta, *et al.*], and on homebrew imbedded SPARC-based controller boards with no operating system at all.

Related Work

We think of the C language as our target machine language, and can then use any competent C compiler as a platform. Several other translators have used C at various levels to take advantage of its wide availability. Where the source language is reasonably close to C, a preprocessor is sufficient, as the first C++ implementation shows. However, C has also been used for languages whose features differ substantially from C [Yuasa and Hagiya] [Bartlett] [Weiner and Ramakrishnan]. Rather than generate C source, some translators use just the code generation phase of the C compiler to achieve a measure of target machine independence [Feldman] [Kessler]. Languages other than C have been sometimes been chosen as targets in an attempt to be portable [Albrecht, *et al.*].

Mimosa: A Compiler from Cedar to C

The Mimosa compiler translates the Cedar language into machine dependent C. There is a front end, compiling into a simple intermediate form, and a back end, translating from the intermediate form to C code. Another back end generates machine code for the Xerox Dragon processor [McCreight]. In the future, back ends may generate other machine codes, other assembler languages, or other versions of C.

Although treating C as merely an intermediate language has significant advantages, the decision to keep program maintenance in Cedar was made independently on the merits of the Cedar language. We were not willing to do a one-time translation from Cedar to C (even readable C). For example, such things as compiler-generated runtime checks should not be exposed to the programmer for maintenance.

Smalltalk-80 is a trademark of ParcPlace Systems, Inc.

SPARC is a trademark of Sun Microsystems, Inc.

Front end

The front end is a descendant of the Mesa [Xerox] compiler used to generate code for the D-machine. It has been substantially modified over the past three years to be retargetable. We parameterized the front end to cover a large variety of architectures. Word size, addressing granularity, and byte order are the most important of these parameters. Other parameters include floating point format and restrictions on contiguous addresses (if any). While this parameter space does not cover the entire range of commercial machines, we intend it to cover most widely used processors. Currently we support the Motorola 68020 and Sun SPARC processors.

The Cedar language itself exposes details of the target machine, so the parameterization of the front end affects the semantics of the Cedar language. We have adopted a set of guidelines so programmers can keep code portable. As an example, Cedar has a type, INTEGER, that is meant to be used for fast signed arithmetic operations; an INTEGER value occupies one machine word (normally determined by register width). In cases where the machine also supports a wider arithmetic operation, LONG INTEGER is provided, although for most machines the two types are the same. In cases where specific widths are required, for example, to describe externally constrained data, the types INT16 and INT32 can be used to specify machine integers with specific numbers of representation bits.

The front end has 2M bytes of source in 120 source files, or about 50K lines of code. This does not include a package which manipulates the tree-structured intermediate form shared between front end and back end. The source for that package has 5K lines of code in 12 files. Taking both of these sections as the front end gives about 55K lines of source in 132 files.

Everything in the front end was affected by the retargeting, and some files were completely rewritten. The use of an intermediate code is completely new. The old code generator was changed to produce the intermediate code, in 16 files and over 8K lines of code. A few other files are completely new, for about 30% of the source lines. Since substantial changes have been made to other parts of the compiler, it would be reasonable to estimate that over 50% of the source lines have received major change (not counting the lines that were completely discarded). We estimate that about 2 person-years overall were spent in redoing the front-end.

Back end

The back-end generates machine dependent C code from intermediate form. It was written completely from scratch for the port, consists of 10K lines of Cedar in 24 modules, and took about 6 person-months to write.

We originally chose C so we could start compiler work before we had selected a target architecture. The platforms we want to run on all come with C compilers, many of them with target-specific optimizers. By choosing C as the machine code to be generated, a high degree of portability is achieved. However, we chose to generate efficient C for a specific target machine, rather than generating portable C. Being one step removed from the actual machine makes it harder to control certain details such as layout of local frames, allocation of registers, *etc.*. Parameterization of the back end provides enough knowledge of the target architecture to generate production-quality C. However, the parameterization of the back end for the Motorola and SPARC architectures is identical, since the architectures do not differ in ways that affect the generated C code.

We also chose C because it was lenient enough for our needs. A language with stronger type rules would have hindered us more than it would have helped. There is hardly any type information in our C source, in part because C types and Cedar types are only partly compatible, and in part because we wanted to avoid many of the type coercions that are part of the semantics of C. All of the variables and parameters in our generated C are unsigned words or bytes, and are cast whenever it is necessary to generate typed operations. Since the front end lays out records, arrays and non-local references to frames, the back end generates more addressing arithmetic than is usual in hand-coded C. This leads to C code that is barely readable, but has good performance. By not exposing the intermediate C source to human readers we can transform a program to enhance its performance without caring about loss of readability. For example, we can access a single bit of a packed structure more efficiently if it happens to be a sign bit. Often the code includes constructs that even a C programmer would balk at, though fortunately our C compilers are able to process it.

Only some of the primitive operations of the Cedar language have corresponding operations in C. More complex features, such as nested procedures, are implemented in C with standard compiler techniques. There is no direct way to translate Cedar's signals or lightweight processes into the C language; this discrepancy is resolved by introducing a runtime system. The implementation of such features in C is by procedures and data structures implemented in the runtime system, just as it would be if we were directly compiling Cedar into machine code.

PCR: A Runtime System

The Cedar runtime environment, to which the generated C code is targeted, is written mostly in Cedar, the rest is written in C, except for a small amount of assembler code.

The environment is built in layers. The lowest layer is akin to an operating system, and provides dynamic loading, threads support, and storage management (including garbage collection). This is about 20K lines of C code and less than a 100 lines of assembler (either SPARC and Motorola 68020 at the moment). This layer is not specific to Cedar, and is in fact intended to provide a language-independent base for high level languages. Called the Portable Common Runtime (PCR), it is described elsewhere [Weiser, *et al.*]. The PCR is described below only where its functionality is particularly important to implementing Cedar features.

The next layer provides the lowest level of Cedar-specific support: imports and exports of Cedar interfaces; and a small library of basic utilities like bit moves and typed storage allocation. It is about 5K lines of C. Called CedarPreBasics, it is the last layer of non-Cedar code, and supports the complete Cedar language except for the ATOM, LIST and ROPE data types, and exception handling.

Support for the remaining features of Cedar is provided by the penultimate layer, called CedarCore, which contains 400 lines of C code and 10K lines of Cedar. From CedarCore on, the full Cedar language is supported. The final support layer, BasicCedar, while not necessary for the language itself, contains services that are considered essential for most Cedar applications. For example it includes several kinds of hash table mechanisms and a general-purpose stream I/O package.

Cedar lightweight processes, interface binding, and exception handling are handled by the runtime system. Portable implementations of these features are discussed below. They are representative of the functionality found in each layer of the runtime system.

Threads

The Cedar language includes a primitive operation FORK, which creates a new ''lightweight'' process (or *thread*) running in the same address space as its parent. The language also has a MONITOR mechanism based on [Hoare], including variables of type CONDITION with WAIT, NOTIFY and BROADCAST primitives to provide synchronization between processes. An ABORT operation can be used to wake up a process that may be waiting on an unspecified condition.

The D-machine Cedar implementation had microcode support for processes, but such support is not essential. Efficient threads implementations on conventional hardware already exist as part of Mach [Accetta et.al.]. In addition, Unix-based threads packages of varying degrees of sophistication are becoming widely available [Cooper] [Kepecs] [Doeppner]. It is important to note, however, that Cedar is quite demanding in this area if the excellent ‘‘feel’’ of the D-machine implementation is to be retained. It would not be acceptable if a compute-bound thread could seize the processor, or if execution of an I/O operation by a single thread caused all other threads to block. Thus, a simple coroutine threads package, which might be adequate in a simulation environment, would not meet our needs. The PCR threads package provides the features we require in a reasonably portable way by using the signal handling, shared memory and asynchronous I/O features available in advanced Unix systems. Some advanced debugging facilities, such as the ability to freeze, examine and thaw individual threads under program control, will be needed for eventual implementation of a full debugger.

Interface binding

The CedarPreBasics layer implements the loadstate: the dynamically constructed mapping from interface items to their implementations. The loadstate implementation is responsible for the final steps in static type checking which insures that dynamically loaded modules mesh correctly with the already-existing types in the system.

An interesting aspect of the loadstate implementation was the method we chose to convey type, import, and export information from the Mimosa compiler to the loadstate. The D-machine Cedar compiler incorporates parts of its symbol tables in the files containing executable code. The D-machine loadstate implementation shares knowledge of these data structures with the compiler, and can interpret them to build the loadstate. In making Cedar portable we realized that issues of byte order, cross compilation, and performance strongly argue against sharing symbol table structures. Instead, the necessary information is conveyed in executable code, encapsulated in an installation procedure for each module. When a module is dynamically loaded into a running Cedar world its installation procedure is called, calling in turn upon the loadstate to type check and bind imported and exported interfaces.

Exception handling

Cedar’s signal mechanism lets programmers write uncluttered code for normal cases, isolating the code for exceptional cases in catch phrases. Since signals propagate over procedure call boundaries, the exceptional cases can be handled where it makes most sense to do so. Enable scopes are either blocks or single procedure applications. Every catch phrase ends by disposing of the signal in one of three ways: it is rejected, forcing catch phrases in ancestral procedure frames to examine and dispose of the signal; it is resumed, supplying a value for the original signal application; or it is terminated. Termination forces the procedure call stack to be unwound to the procedure invocation where the catch phrase was established, which then continues execution with variable values as of the time that the locus of control last left that frame.

Our implementation of Cedar’s signal semantics uses a per-thread catch stack to record active catch phrases. Entering and exiting enable scopes must be cheap because they are part of the execution path in frequent cases. Raising signals and processing them according to the catch stack can be more expensive because it happens relatively infrequently. Entering an enable scope requires pushing several words of data onto the catch stack, *i.e.* about the cost of a procedure call. Raising a signal requires traversing the catch stack, invoking each catch phrase in turn and processing its result: reject, resume or terminate. The catch stack interpretation is itself Cedar code and uses Cedar signals.

The hardest problem in this area was the correct implementation of the termination case of a catch phrase. We use essentially the C library setjmp and longjmp procedures, but for Cedar we must ensure that local variables are restored to their state at the time of the last call out of the frame. In the case of a register machine, this means restoring the registers to the values they had when the stack frame was last left (not necessarily their values at setjmp time). On the SPARC processor or the VAX these semantics are provided by the setjmp and longjmp in the standard library. For the 68020, a considerably more complicated mechanism is required. Our implementation of longjmp for the 68020 walks up the frame stack, restoring registers by interpreting the procedure entry code for each active procedure to determine the registers saved there. For some language implementations on some processors this becomes much more difficult; the C implementation for the DEC WRL Titan processor, for instance, saves and restores registers at arbitrary points in the program text, making a principled register-restoring stack walk difficult [Powell].

One of the noticeable features is the system level approach. The compiler front end notes the scope of each catch phrase and compiles each catch phrase into a separate procedure. The compiler back end generates code to enter and exit the enable scopes at the appropriate points. Exiting enable scopes is complicated by certain styles of exits from blocks. The runtime system then implements the signal handling mechanism. Three pieces of the system share responsibility for this language feature.

Building on Unix and C

One of our major goals in making Cedar run on commercial hardware was to take advantage of software developed in the larger computing community. Further, we wished to begin making Cedar's superior facilities for building large systems available in the Unix environment. To this end several tools have been developed and the Cedar language has been extended. Intercalling between Cedar and C programs is provided. First, Cedar programs can call arbitrary C entry points using a new variant of the MACHINE CODE construct in the Cedar language. Of course, such uses are inherently unsafe, so their use is restricted. Second, tools are available to generate the necessary calls on the loadstate to import Cedar procedures and variables into C programs. Finally, tools to describe Cedar data structures in C and vice versa have been written.

Using these facilities, we have described a substantial portion of the Unix system call interface in Cedar interfaces, including file and socket I/O. Cedar programs use Cedar interfaces for type-checked access to C procedures. Applications of this technique include an X window client [Scheifler and Gettys] and a SunRPC-based Mimosa compilation server.

Performance and Practice

Porting Effort

One of the advantages of retargeting an existing translator is that you have on hand a large body of code to translate. We quickly gained considerable experience with porting code from the D-machine version of Cedar. We have also gained experience with writing code such that it runs unmodified on several machine architectures.

In excess of 365K lines of Cedar code have been ported in the year that the compiler has been available. These packages range in size from small (1K lines for an arbitrary-precision number package) to huge (the compiler itself is 50K lines). A measure of our success in using the same source for both architectures is that only 12K lines of architecture-specific code has been created for the port so far. The modular structure of the Cedar language has allowed us to hide those architecture-specific implementations behind interfaces that exist in both worlds. The layered structure of Cedar encourages us to believe that most of the architecture-specific implementations that are needed for applications have already been written.

The first reasonably large program to be ported was a Scheme [Rees and Clinger] interpreter: 14 files with 9K lines of Cedar. This occurred in January, 1988, and took one person approximately a month (including discovering many compiler and runtime bugs). The next large program was the Cedar Imager [Swinehart, *et al.*], soon followed by an interpreter for the Interpress page description language, together consisting of 60 modules and 40K lines of code. It took one person about three elapsed months. This code was older and more tuned for efficiency, and so uncovered more compiler and runtime bugs. The compiler itself was ported some months after that and took about 6 elapsed weeks.

Some programs are more easily ported than others. In general, Cedar programs that do not exploit knowledge of the D-machine (e.g. word size, addressing granularity, or other hardware features) can just be compiled with the new compiler and run on C platforms. In some cases, data structures can be preserved across architectures by the use of types with specific representations, though possibly at some cost in execution speed.

Performance of generated code.

In practice, the performance of code generated by Mimosa for a given task is comparable to the performance of hand-written C code for the same task. In some cases the hand-written C code is slightly better, although this gap continues to narrow as the compiler gains maturity.

In one case, Dhrystone II [Weicker], the Mimosa-generated C code for the Cedar version actually runs faster than the hand-written C code. The reason is that Cedar strings are word-aligned and length-containing, while C strings are zero terminated. Faster routines for string comparison and string copying, gave the Cedar code a significant advantage for this program. This example illustrates one of the dangers of performing naive comparisons between compilers and between languages.

For good display and printing performance using bitmap graphics it is important to have fast bit block moving operations. One particular routine, written in Cedar, performs a bitwise OR of a block into unaligned memory. This routine, running on a Sun-4/260, transfers blocks that are 20 bits wide by 20 bits high at over 34,000 blocks per second. A careful examination of the C code produced by the Mimosa compiler revealed no significant room for improvement.

We do not pay an execution penalty for writing in Cedar and then translating to C code. This result is in agreement with similar results for other languages [Bartlett]. Since we can exploit optimizing C compilers, we can get good performance without our having to invest in the expertise needed to generate production quality machine code for our current and future platforms.

Debugging

We think of the C language as the assembler language of our target architecture, but we are not willing to debug using only assembler level debugging tools. We use the dbx debugger provided with most of our platforms to access the implementation details of the generated C code [Linton]. Dbx supports several languages, but in all those translators debugging information is provided solely by the front end, and is passed unchanged through the assembler. Since we are using the C compiler as our assembler, providing debugging information is a little more complex. Some of the information needed for debugging is known only to our front-end, *e.g.* decisions about parameter passing, record layout, *etc.*, and all type information. Some of the information needed for debugging is known only to the C compiler, *e.g.*, frame layouts, register allocations, addresses of procedures, *etc.* These two sources of information are merged during a post-processing step by replacing the name, type, and location information from the C compiler with the corresponding information from the Cedar front-end. Dbx then sets and reports breakpoints by reference to Cedar source and uses the Cedar names for the corresponding C variables and procedures. Since we were not willing to modify dbx, we make do with the type system permitted by dbx's interpreter and dbx's pretty-printing of data structures.

Since Cedar, and in fact the PCR in general, supports dynamic loading, we have worked out a scheme for using dbx on dynamically constructed load images. As each module is dynamically loaded the runtime system dribbles symbol table and relocation information to a log file. When the debugger is invoked, the log file is used to construct a synthetic a.out file, with only a symbol table, representing the state of the currently loaded modules. The synthetic a.out can be used to debug the dynamically constructed process.

A better job of supporting Cedar (and inter-language) debugging is awaiting the construction of something like the Cedar Abstract Machine [Swinehart, *et al.*]. We think we can impose such a system on a C platform in the same way we have handled dynamic loading in the PCR: any data structures we need can be constructed at load time by executing code in the module being loaded. This technique frees us from all but the very lowest levels of interactions with the target platform and allows us to be independent of the debuggers provided on those platforms.

Conclusions and Recommendations

We have taken a featureful language designed to be executed on a proprietary architecture and made it portable by having its compiler generate C. We have taken a large body of code written in that language and ported it to industry-standard platforms using the new compiler. We have achieved excellent efficiency as good as hand-coded C and we have gained leverage from work already done for other languages and other systems. We have learned several important lessons along the way, including how to use C as an assembler language, how to use C debuggers for debugging Cedar source, and how language-independent layers can support what are ordinarily language-dependent runtime features like threads and garbage collection. The techniques we have developed will be of growing importance as the computing world is based increasingly on interoperability and the use of existing tools.

Acknowledgments

We thank the following people who helped us build and use the Mimosa compiler and the language tools that support it: Jean-Christophe Cuenod, Jim Foote, Linda Howe, Bill Jackson, Christian LeCocq, Andy Litman, Eduardo Pelegr -Llopart, Bryan Lyles, Michael Plass, and all our other colleagues who have been willing to help us find the bugs in the system. We also thank the program committee members who gave us feedback on the extended abstract.

References

Accetta, J.J., Baron, R.V., Bolosky, W., Golub, D.B., Rashid, R.F. Tevanian, A., and Young, M.W., ‘‘Mach: A New Kernel Foundation for UNIX Development’’, *Proceedings of Summer Usenix*, July, 1986.

Albrecht, P., Garrison, P., Graham, S., Hyerle, R., Ip, P., Krieg-Br ckner, B., ‘‘Source-To-Source Translation: Ada to Pascal and Pascal to Ada’’, *Proceedings of the ACM-SIGPLAN Symposium on the Ada Programming Language*, *SIGPLAN Notices*, **15**, *11*, November, 1980.

Bartlett, J., ‘‘SCHEME->C a Portable Scheme-to-C Compiler’’, Research Report 89/1, DEC Western Research Laboratory, January, 1989.

Cardelli, L., Donahue, J., Glassman, L., Jordan, M., Kalsow, B., and Nelson, G., ‘‘Modula-3 Report’’, DEC Systems Research Center, August, 1988.

Cooper, E. and Draves, R., ‘‘C Threads’’, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, March, 1987.

Doeppner, T.W. Jr., ‘‘Thread Calls’’, unpublished manuscript, Brown University, 1987.

Feldman, S.I., ‘‘Implementation of a Portable Fortran 77 Compiler Using Modern Tools’’, *Proceedings of the ACM SIGPLAN 1979 Symposium on Compiler Construction, SIGPLAN Notices*, **14**, *8*, August, 1979, pp. 98-106.

Goldberg, A. and Robson, D., ‘‘Smalltalk-80: the language and its implementation’’, Addison-Wesley, 1983.

Hoare, C.A.R., ‘‘Monitors: An Operating System Structuring Concept’’, *CACM*, **17**, *10*, October, 1974.

Kepecs, J.H., ‘‘Lightweight Processes for UNIX Implementation and Applications’’, *Summer Conference Proceedings, Portland 1985*, USENIX Association, 1985.

Kernighan, B.W. and Ritchie, D.M., *The C Programming Language*, Prentice-Hall, 1978.

Kessler, P.B., ‘‘The Intermediate Representation of the Portable C Compiler, as Used by the Berkeley Pascal Compiler’’, Unpublished manuscript, Computer Science Division, EECS, University of California, Berkeley, CA, April, 1983.

Lampson, B. and Pier, K., ‘‘A Processor for High-Performance Personal Computer’’, *SIGARCH/IEEE Proceedings of the 7th Symposium on Computer Architecture*, La Baule, May, 1980, pp. 146-160.

Linton, M.A., ‘‘dbx’’, *Berkeley UNIX User’s Manual*, Computer Science Division, EECS, U. C. Berkeley, California, April, 1986.

McCreight, E., ‘‘The Dragon Processor’’, *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS II)*, Palo Alto, October, 1987, pp. 65-69.

Powell, M. private communication.

Rees, J., and Clinger, W. (Eds.), ‘‘Revised^3 Report on the Algorithmic Language Scheme’’, *SIGPLAN Notices*, **21**, *12*, December, 1986.

Scheifler, R. and Gettys, J., ‘‘The X Window System’’, *ACM Transactions on Graphics*, **5**, *2*, April 1986, pp. 79-109.

Steele, G., *Common LISP: The Language*, Digital Press, 1984.

Stroustrup, B., *The C++ Programming Language*, Addison-Wesley, 1986.

Swinehart, D., Zellweger, P., Beach, R., Hagmann, R., ‘‘A Structural View of the Cedar Programming Environment’’, *TOPLAS*, **8**, *4*, October, 1986.

Weicker, R., ‘‘Dhrystone Benchmark: Rationale for Version 2 and Measurement Rules’’, *SIGPLAN Notices*, **23**, *8*, August 1988.

Weiner, J.L. and Ramakrishnan, S., ‘‘A Piggy-Back Compiler for Prolog’’, *Proceedings of the SIGPLAN ’88 Conference on Programming Design and Implementation, SIGPLAN Notices*, **23**, *7*, June, 1988.

Weinreb, D. and Moon, D., *Lisp Machine Manual*, MIT AI Lab, 1981.

Weiser, M., Demers, A., and Hauser, C., ‘‘The Portable Common Runtime Approach to Interoperability’’, submitted to *Twelfth ACM Symposium on Operating Systems Principles*.

Yuasa, T. and Hagiya, M, *Kyoto Common Lisp Report*, Research Institute for Mathematical Sciences, Kyoto University, no date.

Xerox Corporation, Mesa Language Manual, XDE3.0-3001, Version 3.0, November, 1984

Appendix Code Examples

Here is a small example Cedar program with one global variable, a global procedure, some local variables, and a nested procedure that makes several non-local variable accesses. This program is not a typical Cedar program.

```
Test: CEDAR PROGRAM = {
    global: INT;
    Outer: PROC [] = {
        local: INT;
        a: PACKED ARRAY [0..31] OF [0..15];
        Nested: PROC [index: [0..31]] = {
            local <- global+a[index];
            };
        Nested[index: 5];
        };
    }.
```

Below is the C code which is generated for the example program. It begins with type declarations which declare everything to be unsigned integers or characters or types constructed from those basic types. Symbols from the Cedar code have had unique suffixes added to to them, so that similar names don’t interfere in the C code (which has much more restrictive name scopes).

The translation of the outer procedure is straightforward, since all it does is set up a procedure descriptor for the nested procedure and call it with the appropriate actual parameters. Procedure descriptors are used for nested procedures, as well as for procedure variables and procedure parameters.

The first two lines of the nested procedure set up its addressing environment: a pointer to the global frame and a static link derived from the procedure descriptor passed as its last argument. The remaining line is the translation of the body of the nested procedure. The non-local references are fairly self-explanatory. The first use of the index variable selects a byte offset within the array, and the second use of the index variable selects either a 0 bit or a 4 bit shift of the selected byte.

```
typedef unsigned word, *ptr;
typedef unsigned char byte, *bPt;
typedef struct {word f0, f1, f2, f3, f4, f5, f6, f7} W8;
typedef word (*fPt)();
typedef struct {word f0, f1, f2} W3;
typedef struct {W8 f; W3 r} W11;
static void Nested_60();

static void Outer_30() {
	W11 var_1406;
	(* (( (ptr) &var_1406)+4) ) = ( ((word)  (fPt) Nested_60) );
	(* (( (ptr) &var_1406)+5) ) = 1;
	(void) Nested_60(5, (word) (( (bPt) &var_1406)+16)/* var_1390 */ );
};
static void Nested_60(index_1330, formal_1438)
	word index_1330;
	word formal_1438;
{
	register ptr gf_1422 =  (ptr) &globalframe;
	formal_1438 = (formal_1438   16);
	(* (( (ptr) formal_1438)+6) ) = ((* (( (ptr) gf_1422)+4)/* global_1190 */  ) +
	  (((*  (bPt) ((( (bPt) formal_1438)+28) + (index_1330 >> 1)) ) >> (4
	  (((index_1330 & 1) << 2)))) & 017));
};
```