Elk: The Extension Language Kit

Oliver Laumann, Carsten Bormann
Technische Universität Berlin, Germany

ABSTRACT

In the past, users generally were at the mercy of the authors of an application when it came to adapting it to their individual needs and tastes. Fitting an application with an *extension language* (or *embedded language*) enables users to customize and enhance it without having to modify its source code. Recently, variants of the programming language Lisp have become increasingly popular for this purpose, to the point where the abundance of different dialects has become a problem. Of the two standardized dialects of Lisp, only *Scheme* is suitably modest, yet sufficiently general, to serve as an extension language.

Elk, the *Extension Language Kit*, is a Scheme implementation that is intended to be used as a general, reusable extension language subsystem for integration into existing and future applications. Applications can define their own Scheme data types and primitives, which provide for a tightly-knit integration of the C/C++ parts of the application with Scheme code. Library interfaces such as those to various X Window System libraries show the effectiveness of this approach. Various features of Elk such as dynamic loading of object files and freezing of fully customized applications into executables (implemented in those UNIX environments where it was feasible) increase its usability as the backbone of a complex application. Elk has been used as such for nearly five years within a locally-developed ODA-based multimedia document editor; it has been used in numerous other projects after it could be made freely available three years ago.

1. Introduction

The designers and implementors of a large or complex application can rarely anticipate all requirements future users will have on the application. Typically, users wish to be able to customize the user-interfaces of applications according to their personal tastes or requirements, or they have the desire to extend the functionality of an application (either by combining existing functions into new ones or by adding entirely new capabilities). This is especially true for daily-used applications such as text editors and for applications with a high degree of user-interaction or with complex graphical user-interfaces.

Any application can certainly be customized by modifying its source code and recompiling it. But this approach is often not feasible, as the source code of the application or the tools needed to recompile it may not be available. Even if it were feasible, it would be a time-consuming process; it would be hard to keep up with new releases of the application; and the coexistence of multiple, similar versions of the same application would be a general maintenance headache.

The alternative to this approach is to not “hard-wire” the entire functionality and all external aspects of an application in the source code at all, but to provide means to customize the application’s behavior later by it’s users.

Early Customization and Extension Languages

Many applications support at least simple methods for customization, such as command line options or configuration files. More powerful tools for customization are *macro languages*, *command languages*, or *scripting languages* that are typically found in text editors and word processors. Prominent examples of such customization and extension languages are the macro language of the now legendary TECO editor and, in UNIX, the macro language of the *troff* text formatter [Ossanna 1979] and the configuration language of the *sendmail* program.

Although many of these classic extension languages are quite powerful (some of them are full-fledged programming languages), they have a reputation of being “cryptic” and hard to understand and use by untrained users. The prevailing opinion seems to be that only experts can actually benefit from these types of extension languages (for example, people who have mastered the *sendmail* configuration language in all details are commonly appointed the status of a “guru”). In fact, it can be observed that only very few users of the *troff* text formatter (whose macro language is reputed to be particularly cryptic) are using macro packages written by themselves; many users give up after some time and fall back on vendor-supplied macro packages or packages written by a “troff-guru”.

Experience also indicates that simplified or specialized extension languages often grow and have more features added until they resemble a full programming language. Such “organically grown” extension languages are likely to be contorted designs as they will consist of several levels of extensions glued on to their initial, more limited design.

High-Level Extension Languages

Recently application designers have begun to abandon specialized and cryptic macro-style extension languages in favor of extension languages that resemble usual high-level programming languages, mainly languages with Algol/Pascal-style or Lisp-style syntax and semantics. Prominent examples of such high-level extension languages are TPU developed by DEC, the *Ness* language of the Andrew Toolkit [Hansen 1990], AutoDesk’s CAD extension language (a dialect of Lisp), and *Emacs-Lisp*, the extension language of Richard Stallman’s popular GNU Emacs editor [Stallman 1981, Lewis et al. 1990].

Emacs was the first wide-spread application to employ an already existing and widely used high-level programming language as its extension and customization language. Emacs-Lisp is a dynamically-scoped dialect of Lisp with additional functionality for text-editing oriented operations. The approach taken by Emacs has been tremendously successful; during the last years users of Emacs have contributed a wealth of extensions written in Emacs-Lisp.

Elk as a General, Reusable Extension Language

Using Lisp or Lisp-style languages as extension languages seems to enjoy growing popularity; several applications besides Emacs now use dialects of Lisp as their extension language. This development has one disadvantage: the number of coexisting incompatible (but similar) extension languages is continually growing. Users have to learn a new language for each new application, and application writers keep implementing new extension language interpreters instead of reusing existing ones.

These problems can be solved by making available a general, reusable extension language implementation that application writers can include into their applications. The main objective of the *Elk* project was to develop such a generic extension language and make it freely available to encourage application writers to use it in their applications.

2. Overview of the Extension Language Kit

The Evolution of Elk

The development of Elk began when we were searching for a suitable extension language implementation for ISOTEXT [Bormann et. al. 1988, Bormann 1991], an ODA-based document processing system with a graphical user-interface. ISOTEXT is almost entirely written in C++; its user-interface is based on the X window system [Scheifler et al. 1986] and the OSF/Motif widget set. Customizability and extensibility by means of a full extension language was a basic requirement on the design of ISOTEXT.

As we considered language design to be the domain of a “selected few”, we decided to use an existing programming language as the basis for the extension language of ISOTEXT. This decision was also influenced by our desire to develop a general, reusable extension language implementation that is not hard-wired into one specific application. For a number of reasons an interpreted language seemed preferable: it should be possible to add extensions to or modify extensions in a running application without re-linking it; bugs in extensions should not crash the application; interpreted languages usually offer better debugging facilities; and implementing an interpreter generally requires less efforts than implementing a compiler.

From the beginning we favored Lisp or a dialect of Lisp as the basis for a general extension language. Most dialects of the Lisp family are “small”, easy to implement, general-purpose languages with simple syntax and powerful semantics, and the suitability of Lisp as an extension language had already been demonstrated by several applications, among them GNU Emacs. Early in the project we considered to use Emacs-Lisp, but it appeared infeasible to isolate the Lisp-interpreter from the rest of Emacs. In addition, at the time we investigated Emacs-Lisp it was lacking several desirable language features, such as support for floating point and arbitrary precision numbers (*bignums*). We also considered to use MIT Scheme [MIT 1984], but due to the enormous size of its implementation it would have dominated the size of the application.

Scheme as an Extension Language

As other implementations of Lisp or Lisp-like languages available at the time of our investigations did not meet our requirements, we finally decided to write an interpreter for the Lisp dialect Scheme [Clinger et al. 1991, Dybvig 1987, Springer et al. 1989, Abelson et al. 1985].

This Scheme interpreter is the main component of the Elk package. Scheme is a simplified, "cleaned-up" dialect of Lisp with first-class procedures and static scoping rules. The Scheme language is based on only a few language features and semantic concepts; it consists of a small core if syntactic forms, a set of extended forms derived from them, and a number of standard procedures (*primitive* procedures) that operate on a rich set of types of objects (among them numbers, lists, vectors, symbols, characters, and strings). In 1991 Scheme became an IEEE standard [IEEE Std 1178-1990] (the standard document, although only 50 pages long, includes the complete formal semantics of the language).

The standardization effort has increased the acceptance of Scheme; for instance, the Extension Language Working Group of the CAD Framework Initiative has recently selected Scheme as the extension language for future CAD applications [CFI 1991a, CFI 1991b]. Among the established programming languages we consider Scheme the ideal candidate for a general extension language – it is standardized; its semantics are well-defined; it has a simple syntax and is easy to implement; and it is sufficiently small to not dwarf the application to be extended.

Extending the Extension Language

The implementation of an extension language must itself be extensible. To enable code written in such a language to manipulate objects or state of the application to be extended, the language's base set of primitive procedures and data types must be augmented by application-specific primitives and types. In fact, easy extensibility of the language has been the primary design consideration in the development of Elk (as opposed to performance or number of language features). To allow Elk programs to be expressive in the context of a given application, application writers are encouraged (and expected) to extend the language base of Elk by a rich set of application-specific data types and Scheme primitives to operate on objects of these types. Adding new types and primitives to Elk is an inexpensive operation; it is not uncommon for an application to define hundreds of application-specific Scheme primitives.

All primitive procedures of Elk are C or C++ functions. This is true for both built-in primitives (such as *car* and *cdr*) and primitives defined by extensions. From the Scheme programmers' point of view, primitives and types from the base set of the language are indistinguishable from application-specific primitives and types. Extensions "register" new primitives with the interpreter by supplying the name of the primitive along with a pointer to the function implementing the primitive and other information. New types are defined in a similar way. Registration of new primitives and types typically takes place at the time the interpreter is invoked or when a compiled extension is loaded into the running interpreter.

Another way to use the extension mechanisms of Elk is to provide interfaces to libraries, such as the C library or the libraries of the X window system (e. g. *Xlib*). Elk has no facility to directly import "foreign" functions (although such a facility could be written as an extension; in fact, this has been done at a company where Elk is used). Therefore, a small amount of code acting as "glue" between Elk and the library has to be written to make the contents of a library available to Scheme programmers. The main purpose of this interface code is to check the arguments supplied to the library functions, to convert Scheme objects into C types, and to convert the results of library functions back into Scheme objects. Library extensions often act as glue between the application to be extended and the libraries used by the application; they allow the application writers to abstract from the details of the libraries. Although it is useful to distinguish between *library* extensions and extensions interfacing to *applications*, there is no technical difference – in both cases a collection of types and functions is made available to the Scheme world.

The Contents of the Extension Language Kit

The Elk distribution consists of the Scheme interpreter (the *interpreter kernel*) and a number of library extensions. These provide interfaces to the X11 “Xlib” (similar to “CLX” [CLX 1991] in its functionality, but implemented on top of Xlib), to the X11 toolkit intrinsics (“Xt”) and the Athena and OSF/Motif widget sets, and to a small number of functions of the C library. The X extensions are especially useful for application writers whose applications have graphical user-interfaces based on X; Elk enables them to write their user-interfaces or parts thereof in Scheme to achieve maximum customizability.

Elk can also be used as a free-standing Scheme implementation. In combination with the X extensions it is well-suited as a tool for interactively exploring X, for teaching X to beginners, and as a platform for rapid prototyping of X-based applications.

3. Using Elk in Applications

Bringing Everything Together

In contrast to other extension language implementations (e. g. TCL [Ousterhout 1990]), Elk does not provide its functionality in the form of a library that is statically linked into an application to be extended. Instead, the object modules comprising the application and all required library extensions are dynamically linked with and loaded into the running Scheme interpreter. To accomplish this, the *load* primitive of Elk has been extended to load object files – compiled extensions written in C or C++ – besides files containing Scheme code. Dynamic loading enables applications to load less frequently used modules into the running program on demand; such an application is initially smaller than the equivalent statically linked application (where all modules must be combined into one large executable file).

Dynamic loading of object files is often used together with the *dump* primitive that creates an executable file from the running interpreter, similar to *unexec* of GNU Emacs or *dumplisp* in some Lisp systems. The *dump* primitive of Elk differs from existing, similar mechanisms in that the newly created executable, when called, starts at the point where *dump* was called in the original invocation (as opposed to the program’s *main* entry point). Here the return value of *dump* is “true”, while in the original invocation it returns “false” (not unlike the UNIX *fork* system call).

Dynamic Loading and Dump in Cooperation

To generate a new instance of an application one would typically invoke the Scheme interpreter, load all object modules and all Scheme code required initially, perform all initializations that can survive a “dump”, and finally dump an image of the running interpreter containing all the loaded code into a new executable on disk. The use of *dump* avoids time-consuming activities like loading of object files and other initializations on each startup. The dumped executable, when started, resumes after the call to *dump*; at this point one would perform the remaining, environment-dependent initializations and finally invokes the application’s “main program” (e. g. enter the X toolkit’s event processing main loop). Listing 1 shows a (slightly simplified) Scheme program that generates and starts a new instance of an application.

```
;;; Load initially required object files and Scheme files of
;;; application and dump image into executable file.
;;; Dumped file enters application's main loop on startup.

(load 'main.o)      ; initial object modules
(load 'edit.o)
(load 'x11.o)       ; (a library extension)
...
(load 'ui.scm)      ; initial Scheme files
(load 'custom.scm)
(load 'x11.scm)
...
(initialize-application)

(if (dump 'a.out)
    (begin                  ; dumped a.out starts execution here
      (initialize-some-more)
      (main-loop-of-application)
      (exit)))

;; Original invocation gets here when dump is finished.  We're done.
```

Listing 1: Scheme code to generate and start an application

Note: Filenames can be given as symbols (besides the usual string literals). A more meaningful name than a.out would probably be chosen in practice.

On systems that do not support dynamic linking and loading of object files (such as older versions of UNIX System V) or where *dump* cannot be implemented, the interpreter kernel and the application and library extensions are linked statically and combined into one executable.

In any case, in an application using Elk, the control initially rests in the Scheme interpreter. The interpreter acts as "main program" of the application; it is the interpreter's *main()* function which is invoked on startup of the program. Therefore the first code to execute in an application is Scheme code; this Scheme code provides the shell functionality of the application (it is hence called *shell code*). The shell code may perform a few simple tasks, for instance, load a user-provided initialization file containing customization code for the application and then enter the application's main loop, or it may be as complex as in ISOTEXT, where the entire X-based user-interface is written in Scheme.

Making Oneself Known to the Extension Language

The application, as it is linked with the extension language interpreter, has full access to all external functions and variables of the interpreter kernel. The interpreter, on the other hand, does not have any knowledge of the contents of dynamically linked and loaded object modules; all it sees of an object file being loaded is the file's symbol table. To obtain "hooks" into a newly loaded extension, the interpreter searches the symbol table of each object file being loaded for functions whose names start with the prefix “init_” (*extension initialization functions*) and invokes these functions as they are encountered. Likewise, to support extensions written in C++, any C++ static constructors found in the symbol table are called. When linked statically, the interpreter must scan its own symbol table on startup to find and invoke the initializations functions.

Besides initializing private data of the modules being loaded, these initialization functions register with the interpreter the Scheme primitives and Scheme data types implemented by the extensions. To enable extensions to register new primitive procedures and types, the interpreter kernel exports two functions: *Define_Primitive()* to register a new Scheme primitive and *Define_Type()* to register a new type.

The arguments supplied to *Define_Primitive()* are a pointer to the function implementing the primitive procedure, the Scheme name of the primitive, the minimum and maximum number of arguments, and a symbol indicating the *calling discipline* of the primitive. Calling disciplines are: normal procedure with fixed number of arguments, such as *car*; procedure with variable argument list, such as *append*; and *special form* (variable number of unevaluated arguments). *Define_Type* is invoked with the Scheme name of the type, the size of the type’s representation in C or C++, two functions implementing the *eqv?* and *equal?* predicates for objects of this type, a function that is called by the interpreter to print an object of the new type (the type’s *print function*), and a function providing information about the type to the garbage collector. The return value of *Define_Type()* is a “handle” to the newly defined type (basically a small, unique integer); its main uses are to check the type of arguments supplied to primitive procedures and to instantiate objects of this type.

4. Notes on the Implementation

Implementing Continuations

Finding a way to efficiently implement Scheme’s *continuations* called for considerable efforts during the design phase of Elk. Continuations are a powerful language feature; they support the definition of arbitrary control structures such as non-local loop and procedure exits, *break* and *return* as in C, exception handling facilities, explicit backtracking, co-routines, or multitasking based on *engines*.

The primitive procedure

```
(call-with-current-continuation receiver)
```

packages up the current execution state of the program into an object (the *continuation* or *escape procedure*) and passes this object as an argument to *receiver* (which is a procedure of one argument). Continuations are first-class objects in Scheme; they are represented as procedures of one argument (not to be confused with the *receiver* procedure). Each time a continuation procedure is called with a value, it causes this value to be returned as the result of the *call-with-current-continuation* expression which created this continuation. If the procedure *receiver* terminates normally (i. e. does not invoke the continuation given to it), the value returned by *call-with-current-continuation* is the return value of *receiver*.

As long as the use of a continuation is confined to the runtime of the *receiver* procedure, *call-with-current-continuation* is similar in its functionality to *catch/throw* in most Lisp dialects or *setjmp/longjmp* in C. However, continuations, like all procedures in Scheme, have indefinite extent; they can be stored in variables and called an arbitrary number of times, even after the *receiver* and the enclosing *call-with-current-continuation* have already terminated. Listing 2 shows a program fragment where continuations are used to get back an arbitrary number of times into the middle of an expression whose computation has already been completed. While not particularly useful, this example demonstrates that continuations can be used to build control structures that cannot be implemented by means of less general language features like catch/throw or setjmp/longjmp in C.

```
(define function
  (lambda (n m)
    (+ n (mark m)))                    ; return n+m

(define get-back "uninitialized")

(define mark                           ; identity function, but also
  (lambda (value)                      ; assign current continuation
    (call-with-current-continuation    ; to a global variable
      (lambda (continuation)
        (set! get-back continuation)
        value))))

(function 10 20)  →30                  ; invoke function
(get-back 5)  →15                      ; resume with new value
(get-back 0)  →10                      ; ...once more
```

Listing 2: Using continuations with unlimited extent

The different approaches that can be used in implementing continuations are intimately tied to the strategies used for interpreting the language itself. Scheme interpreters generally employ a lexical analyzer and parser – the *reader* – to read and parse the Scheme source code and produce an intermediate representation of the program. During this phase, symbols are collected in a global hash table (in Lisp jargon, the symbols are *interned*), and a tree structure representing the program's S-expressions is built up on the heap of the interpreter. The majority of interpreters compile this intermediate representation into an abstract machine language (such as *byte code*) in a second pass (in practice, these passes may be combined in one pass). The evaluator is then implemented as an abstract machine which interprets the low-level language; this machine – usually a simple stack machine – may even be implemented in hardware.

In an abstract machine implementation, the obvious approach to implement *call-with-current-continuation* is to package up the contents of the registers (program counter, stack pointer, etc.) and the current runtime stack of the abstract machine. As continuations have indefinite extent, it would not suffice to just capture the abstract machine's registers (as the C library function *setjmp* does for the real machine). To be able to continue the evaluation of procedures that have already returned and whose frames are therefore no longer on the stack, a continuation must also embody the contents of the abstract machine's stack at the time it is created. When a continuation is applied, the machine resumes the "frozen" computation by restoring the saved registers and stack contents of the abstract machine.

This technique would not work in Elk, because at the time a continuation is created, arbitrary library functions may be active in addition to Scheme primitives. For instance, consider the extension interfacing Elk to the "Xt" toolkit intrinsics of the X window system. When using this extension, a typical scenario is that some Scheme procedure invokes the primitive that enters the toolkit's event dispatching main loop (*XtAppMainLoop()*). When an event arrives (for example, a mouse button press event), the toolkit's main loop invokes a callback function, which in turn calls a user-supplied Scheme procedure to be executed when a mouse button is pressed. This Scheme procedure might in turn invoke yet another function from the "Xt" library, and so on. A similar example would be a *qsort* or *ftw* extension to Elk, where the user-supplied function called by the *qsort()* or *ftw()* C library function would invoke a procedure written in Scheme.

The interpreter's thread of execution at any time apparently involves both Scheme primitives and library functions (such as *XtAppMainLoop()* and *qsort()* in the examples above) in an arbitrary combination. Therefore, a continuation must not only embody the execution state of the active Scheme procedures, but also that of the currently active library functions (such as local variables used by the library functions). In the approach followed by Elk, a continuation is created by capturing the machine's registers – like *setjmp* in C does – and the C runtime stack. When a continuation is applied later, the registers and the saved stack contents are copied back. Actually, we did not follow the usual "abstract machine" technique in Elk at all; instead, the Scheme evaluator directly interprets the intermediate representation produced by the reader. In a sense, it is the "real" machine (the hardware on which Elk is executed) that plays the role of the abstract machine in implementations with byte-code compilation.

Although the abstract machine technique usually yields faster execution of Scheme code, the performance of Elk is comparable to that of existing interpreters employing the abstract machine approach, and the implementation of Elk is less complex than that of comparable interpreters using byte-code compilation. While the technique to implement continuations in Elk is not portable – it is based on certain assumptions on the machine's stack layout and the C compiler and runtime environment – implementations of the small machine-dependent part now exist for most major machine architectures.

The Implementation of "dump"

Continuations provide a natural basis for implementing the execution-state preserving semantics of the *dump* primitive. When called, *dump* invokes *call-with-current-continuation* (actually, since it is written in C, the interpreter's internal version of *call-with-current-continuation*). The real work is done in the *receiver* procedure; it stores the newly created continuation into a global variable, creates an executable file from the image of the running process, sets a global *was-dumped* flag to indicate that a dump has taken place, and finally returns "false". The return value of the *dump* primitive is the return value of this call to *call-with-current-continuation*, i. e. "false" if a dump has just been performed.

When the interpreter − either the original program or a dumped executable − is started, it examines the *was-dumped* flag as its very first action. If the flag is set, the running interpreter was started from a dumped executable. In this case the interpreter immediately invokes with an argument of "true" the continuation that was saved away by a call to *dump*; this causes that call to *dump* to finish and return "true" to its caller. If, on the other hand, the *was-dumped* flag is not set (i. e. the running process was not started from a dumped image), the interpreter initializes and starts up as usual.

Before writing an image of the running process to disk, *dump* has to close all open Scheme file ports, as open file descriptors would not survive a *dump* − they would no longer be valid in the dumped executable. Generally, this is true for all objects pointing to information maintained by the UNIX kernel, such as the current directory, the current signal dispositions, resource limits, or interval timers. Users and implementors of Elk extensions must be aware of this particular restriction. For instance, users of the X11 extensions have to make sure that, if *dump* is to be used, connections to X-displays are only established in the dumped invocation.

To be able create an executable from the running process, *dump* has to open and read the a.out file from which the running process was started (actually, if the system linker has been called to dynamically load object files, the output of the most recent invocation of the linker is used instead of the original a.out). The symbol table of the new executable is copied from the a.out file of the running program; in addition, the a.out header has to be read to obtain the length of the text segment and the start of the data segment of the running process. To do so, *dump* has to determine the filename of a.out file from which the process was started based on the information in *argv[0]* and in the PATH environment variable. This approach is obviously based on several prerequisites: *dump* must be able to access its a.out file (*argv[0]* must carry meaningful information; the file must be readable) and the running program's a.out file must not have been stripped. It would have been advantageous for the implementation of *dump* if the entire a.out file had been automatically mapped into memory on startup, like it is done, for instance, in NEXT-OS/Mach.

dump combines the data segment and the "bss" segment of the running process into the data segment of the new executable. If Elk had a separate heap for storing constant objects (future versions may have one), *dump* would place this read-only part of the memory into the new executable's text segment to make it sharable. When the interpreter's heap is written to disk, *dump* seeks over the unused portions of the heap, so that fake blocks can be used for these parts of the file. This results in a considerable conservation of disk space in the final executable, as at least half of the interpreter's heap is unused at any time due to the garbage collection algorithm of Elk.

Since the a.out formats used in the numerous versions of UNIX differ vastly, Elk has to include separate implementations of *dump* for the currently supported a.out formats. Version 1.5 of Elk handles the BSD-style a.out format used in BSD and "derived" UNIX versions (such as SunOS 4.1), the COFF a.out format (used in older releases of UNIX System V and in A/UX), Extended COFF of MIPS-based computers running Ultrix, and the ELF a.out format of System V Release 4 and related UNIX versions (Solaris 2.0).

Dynamic Loading of Object Files

When loading an object file during runtime, addresses within this object file must be relocated to its new location in the program's address space. To allow extensions to directly reference objects of the interpreter kernel, such as the heap and the built-in primitives, unresolved references into the *base program* must be resolved during dynamic loading. Finally, the object file needs to be able to export its entry points (such as Elk's extension initialization functions) to the base program.

More than one object file may have to be loaded into one invocation of Elk. To manage non-trivial, hierarchically structured sets of extensions, where a number of high-level extensions require one or more lower-level extensions to be loaded, it is essential that object files loaded later can make use of the symbols defined by previously loaded object files. As this style of dynamic loading allows building complex systems from small components incrementally, we will use the term *incremental loading*.

With the advent of 4.0 BSD in 1980 [Joy 1980], support for incremental loading was added to the system linker and has since been supported by most major UNIX variants: when the -A option and the name of the base executable are supplied to the linker, linking is performed in a way that the object file produced by the linker can be read into the already running executable. The symbol table of the resulting object file is a combination of the symbols defined by the base program and the newly defined symbols added by the linking process, from the object file or from libraries used in linking. Only this newly linked code and data is entered into the resulting object file. The incremental style of dynamic loading is achieved by saving the resulting output file each time the linker is invoked and using this file as the base program for the next incremental loading step, such that both old and new symbols can be referenced.

Incremental loading is generally supported by the linkers of UNIX versions that use the BSD-style a.out format and by those of several UNIX systems based on more modern a.out formats (e. g. Ultrix). It is not supported by any existing release of UNIX System V. Some newer UNIX versions that have shared libraries and dynamic linking (such as System V Release 4 or SunOS) offer a library interface to the dynamic linker. In some systems this kind of interface is intended to replace the incremental loading functionality of the system linker. These dynamic linker interfaces usually come in the form of a library that exports functions such as *dlopen()* to map a shared object module or shared library into the address space of the caller (the base program) and *dlsym()* to obtain the address of a function or data item in the newly attached object module.

In some implementations, object files attached through *dlopen()* may directly reference symbols in the base program; in other implementations they may not. In any case, object files cannot directly reference symbols defined by objects that have been placed into the program by previous calls to *dlopen()* (only, if at all, indirectly by calling *dlsym()*). Thus, these dynamic linker interfaces are clearly inferior to incremental loading, as they lack the important capability to load a set of object files *incrementally*. Vendors who have replaced "/bin/ld -A" by a *dlopen*-style library in their UNIX systems, or who intend to do so, do not seem to be aware of the fact that this change will break applications that rely on incremental loading.

For Elk, the consequence of being restricted to dynamic linker interfaces of that kind is that, except for the simplest applications, one must pre-link all possible combinations of extensions that are not completely independent of each other. In general, given a set of n extensions each of which can be based on one out of m other extensions, this means having to prepare and keep around $n \times m$ pre-linked object files; not to mention the contortions one has to go through when the hierarchy of extensions has a depth greater than two (not an unlikely scenario in practice). If the number of extensions and relations between them is larger than trivial, or if the extensions are large or require large libraries, keeping around all pre-linked combinations of object modules will waste a considerable amount of disk space.

Another, although minor, problem with these dynamic linker interfaces is that they usually offer only a simple-minded function (such as *dlsym()*) to look up the address of a specific symbol of a newly accessed object module (typically some kind of module initialization function); but they do not provide a way to scan all newly defined symbols. This functionality is insufficient to implement extension initialization in Elk, where a dynamically loadable extension often is composed from a number of small modules, each defining its own initialization function. Requiring a single, common initialization function name for the entire object file implies that (often configuration-dependent) “glue code” must be added to call all the individual initialization functions, including the C++ static constructors.

Non-Standard Language Features

As the current version of the Scheme standard (deliberately) does not specify several important language issues, such as error handling or syntactic extensions, we have added a number of non-standard language features to the Scheme interpreter of Elk to fill some of the holes.

A proposal for a macro extension has only recently been added as an addendum to the *Revised^4 Report on the Algorithmic Language Scheme* [Clinger et al. 1991] and is still being discussed controversially within the Scheme community. To avoid having to wait for a final version of a macro system to evolve and be included in the Scheme standard, we implemented a simple-minded macro mechanism in Elk that resembles the macro facilities offered by various existing Scheme and Lisp systems.

One area where the Scheme standard does not specify any language features yet is error and exception handling; the standard merely states which error situations a conforming implementation is required to detect and report. Since it is essential for a non-trivial application to be able to gracefully handle error situations (such as failures in interactions with the operating system) and other exceptional conditions, we have added a simple error and exception handling facility to Elk.

When an error is detected by the interpreter, a user-supplied error handling procedure is invoked with arguments identifying the type and source of the error. The standard top-level of Elk provides a default error handler that prints an error message and then resumes the main read-eval-print loop by means of a *reset* primitive. Most primitives of Elk and the extensions use this error handling facility to signal an error, as opposed to indicating failure by a distinctive return value (which would be prone to being ignored). To by-pass the standard error handler and “catch” failure of a particular primitive, programs may enclose the call to the primitive by *call-with-current-continuation* and use *fluid-let* to dynamically bind the error handler to the continuation (as shown in listing 3).

```
(define (new-open-input-file name)
  (call-with-current-continuation
    (lambda (return)
      (fluid-let ((error-handler
                    (lambda args (return #f))))
        (open-input-file name)))))
```

Listing 3: A version of open-input-file that returns the newly opened port on success, #f on error

Elk provides a similar facility to handle an *interrupt* exception: a user-supplied interrupt handler is invoked when a SIGINT signal is sent to the interpreter (usually by typing the interrupt character on the keyboard). Support for other exceptions, such as timer interrupts, may be provided in future versions.

Another non-standard primitive that facilitates handling of errors is *dynamic-wind*, a generalization of the *unwind-protect* form offered by many Lisp dialects. *dynamic-wind* is used to implement the *fluid-let* special form (to create *fluid* or dynamic variable bindings). Both *dynamic-wind* and *fluid-let* are also provided by several other Scheme dialects [MIT 1984, Dybvig 1987].

The current version of the Scheme standard does not provide any language features that would make it possible to implement a useful Scheme debugger (apart from a debugger based on source code instrumentation). To compensate this shortcoming, we have added a few primitives that aid the implementation of a simple interactive debugger, among them an *eval* primitive (although, theoretically, *eval* could be implemented by writing an expression into a temporary file and then loading this file). In addition, Elk, like a few other Scheme dialects, provides lexical environments as first class (but immutable) objects. Other non-standard primitives that are helpful in debugging are *procedure-lambda* to obtain the lambda expression that evaluated to a given procedure, and a primitive that returns the list of currently active procedures together with their actual arguments and the lexical environments in which the procedure calls took place (a *backtrace*).

Garbage Collection

The garbage collector of Elk is based on the *stop-and-copy* algorithm; a description of this algorithm can be found, among other places, in [Abelson et al. 1985]. An incremental, generational garbage collector for Elk has recently been adapted from Yip's garbage collector [Yip 1991] and is now being integrated into Elk as an alternative to the stop-and-copy garbage collector.

Extensions to Elk can register *before-GC* and *after-GC* functions with the interpreter; these functions are invoked by the garbage collector immediately before and after each garbage collection run. Within *after-GC* functions, extension can determine whether or not a particular Scheme object has been moved by the preceding garbage collection run. An object has not been moved by the garbage collector if no references to the object exist any longer, i. e. if it has become garbage. In this case, an extension may perform some kind of clean-up action; for example, if the now unreferenced object contains a handle to an open file, close this file.

The Elk distribution contains a library based on this mechanism that enables extensions to register a *termination function* for objects of a particular type. The termination function associated with an object is then invoked by the garbage collector automatically when this object has been detected to be unused. The Xlib extension of Elk uses this library to perform suitable finalization operations on objects created by the extensions, for example, close windows, unload fonts, and free colormap objects that have become unreferenced. This mechanism is slightly complicated by the fact that objects may have to be terminated in a predefined order; for instance, when an X11 display becomes garbage, all objects associated with this display must be terminated before the display itself is finally closed.

5. Practical Experiences with Elk

Elk and ISOTEXT

In developing the ODA-based document processing system ISOTEXT, Elk proved to be a major asset [Bormann 1991]. Scheme was used as the implementation language for all user interface aspects of ISOTEXT. Apart from providing extensibility to users of ISOTEXT, using Elk as the base for ISOTEXT made it possible to write the shell code in a high level language with all its amenities, e. g. automatic storage reclamation. As no recompilation and relinking is necessary, it is a quick operation to apply and test changes to the user interface.

Elk provides for a strong "firewall" in the ISOTEXT system: bugs in the Scheme code give rise to errors at the Scheme level, which can easily be debugged using the (primitive, but functional) built-in debugger of Elk, while conditions such as core dumps always are the result of bugs in the ISOTEXT kernel implementation.

All this assistance for the development of ISOTEXT could be achieved without sacrificing the performance of the ISOTEXT kernel system, which is still written in efficient C++.

Elk also allowed to isolate the ISOTEXT kernel from the choice of an X toolkit: The ISOTEXT kernel is unaware of the toolkit being used ("Xt" with OSF/Motif). The Scheme code builds a user interface using the Motif library interface and provides X windows to the ISOTEXT kernel. Input is processed by the Scheme code which calls editor primitives provided by the ISOTEXT kernel and schedules redisplay operations. Replacing Xt and OSF/Motif by e. g. *Xview* would require no changes in the ISOTEXT kernel.

During the development of ISOTEXT it turned out that the extension writer's interface of Elk could benefit from a number of improvements. The main problem is that it is difficult to write non-trivial extensions, because too much of the inner workings of the interpreter is exposed to and must be dealt with by the extension writer.

In particular, as Elk can trigger a garbage collection run at any time a chunk of heap space is requested, extensions must register local or temporary Scheme object with the garbage collector to protect them from being discarded during a subsequent GC run. While this scheme has the advantage that maximum utilization of the available heap space is guaranteed, it imposes a strict discipline on the extension programmer. Failure to properly protect temporary Scheme objects usually results in delayed crashes of the application that hard to trace back to the actual source of the problem. In fact, when developing the X11 extensions to Elk, most of the time spent for debugging was due to GC-related bugs.

Library Extensions

The problems we encountered when designing and implementing Elk's interfaces to the C libraries of X11 are likely to be applicable to a wide range of similar APIs. The X11 libraries, especially Xlib, are quite complex; the core Xlib alone exports more than 600 functions and macros, within which numerous different mechanisms are employed for passing arguments and for manipulating objects, some of which are perceived to be rather verbose and error-prone by many programmers. This complexity is, at least partly, caused by the semantic restrictiveness of the C programming language. Thus, when designing the Scheme language interface, we had the opportunity to eliminate some of the "warts".

If integration of a library with an extension language (or interactive language in general) is not anticipated at the time the programmer's interface of the library is designed, writing a properly functioning extension language interface to this library can become quite difficult or even impossible. This problem is exemplified by the "Xt" toolkit intrinsics library of X11, in particular by earlier versions of this library. The following example illustrates a typical difficulty caused by the "static" nature of the programmer's interface to "Xt":

Each class of graphical objects (*widgets* in "Xt" terminology) exports a list of attributes (*resources*) that are associated with objects of this class. A function is provided by "Xt" to obtain the list of resources of a widget class together with the name and C type (integer, string, pixmap, color, etc.) of each resource. On this basis, operations like setting the value of a widget's resource from within Scheme can be implemented in a straightforward way. The "Xt" extension just has to check if the user-supplied Scheme value can be converted into a C object of the resource's type, perform this conversion, and call the Xt-function to set the resource, or complain to the user if the value is not suitable for this resource. However, until recently, some classes of widgets had a subset of resources (the *constraint resources*) whose names and types could not be obtained by an "Xt" application. While this omission is usually not perceived as a problem for C programmers (who would know each widget's resources *a priori* from reading the documentation), it had a dramatic effect on Elk's "Xt" extension, as now the knowledge about these resources had to be hard-wired into the extension. As a result, the extension's source code had to be modified for each new widget set to be made usable from within Scheme code.

This particular problem has been remedied in recent releases of X11, though several similar problems remain; even in the UNIX C library. While design flaws of library interface often go unnoticed or are considered minor when writing C or C++ programs (e. g. the fact that implementations of the *qsort()* functions are non-reentrant), they become crucial when these libraries are made accessible to an extension language. As the importance of extension languages is growing, it is essential that future library interfaces will be designed with the particular requirements of extensions languages in mind.

Conclusions

Since the Elk project was started, both the research community and significant industry projects have generated increasing numbers of "embeddable language" implementations. While many such languages inherit the syntactic flavor of BASIC, those projects that focus on the ability to build non-trivial extensions recently seem to almost exclusively turn to the Scheme language.

Scheme has proven to be an effective language for extension language purposes. In the beginning of the ISOTEXT project, there were concerns that an implementation of the full Scheme language would be both too large and too slow. These reservations proved to be unfounded: the code of Elk has less than half the size of medium size applications such as *vi*. While the performance of Elk may be uninspiring (no compiler is available), this has proven not to be a critical issue, as any bottlenecks can easily be replaced by a primitive recoded in C or C++.

While Elk has been used in the ISOTEXT project since 1987, legal issues prevented making it publicly available until the fall of 1989. Since, Elk has gained acceptance, in fact sufficient momentum to encourage others to contribute software. We know of projects that make use of Elk at research institutions such as ... as well as in industry work at Autodesk, Computervision, HP, IBM, Intel, NEC, and SNI.

Availability

Elk is available in legally unencumbered status. The current version as of September 1992 is 1.5. The newest version of Elk is available via anonymous FTP from export.lcs.mit.edu (/contrib) and ftp.cs.tu-berlin.de (/pub).

6. References

[Abelson et al. 1985]

Harold Abelson and Gerald J. Sussman with Julie Sussman, *Structure and Interpretation of Computer Programs*, MIT Press, Cambridge, Mass., 1985.

[Bormann et. al. 1988]

Ute Bormann, Carsten Bormann, C. Bathe, *SDE – A WYSIWYG Editing and Formatting System for ODA and SGML Documents*, ESPRIT ’88, Proceedings of the 5th Annual ESPRIT Conference, Brussels, November 14-17, 1988.

[Bormann 1991]

Carsten Bormann, *Open Document Processing and the ISOTEXT System*, Doctoral Dissertation, TU-Berlin, 1991.

[CFI 1991a]

CAD Framework Initiative, CFI Extension Language Sub-Committee, *CFI Extension Language Selection Document*, CFI Document Number 87, CAD Framework Initiative Inc., Austin, Texas, 1991.

[CFI 1991b]

CAD Framework Initiative, Extension Language Working Group: Architecture Technical Sub-Committee, *Extension Language: Core Language Selection*, Draft Proposal Version 0.7, CFI Document Number ARCH-91-G-1, CAD Framework Initiative Inc., Austin, Texas, 1991.

[Clinger et al. 1991]

William Clinger and Jonathan Rees (Editors), *Revised^4 Report on the Algorithmic Language*

Scheme, November 2, 1991. Available per FTP from altdorf.ai.mit.edu.

[CLX 1991]
CLX − Common LISP X Interface, 1991. (Part of the X11 Release 5 distribution available from the MIT software distribution center.)

[Dybvig 1987]
R. Kent Dybvig, *The Scheme Programming Language*, Prentice Hall, Englewood Cliffs, NJ, 1987.

[Hansen 1990]
Wilfred J. Hansen, *Enhancing documents with embedded programs: How Ness extends insets in the Andrew ToolKit*, Proceedings of IEEE Computer Society 1990 International Conference on Computer Languages, March 12-15, 1990, New Orleans.

[IEEE Std 1178-1990]
IEEE Standard for the Scheme Programming Language, New York, May 28, 1991.

[Joy 1980]
Bill Joy, *Changes in the VAX system in the Fourth Berkeley Distribution*, Computer Systems Research Group, University of California, Berkeley, November 1980.

[Lewis et al. 1990]
Bil Lewis, Dan LaLiberte, the GNU Manual Group, *GNU Emacs Lisp Reference Manual*, Edition 1.03, Free Software Foundation, Cambridge, Mass., December 1990.

[MIT 1984]
MIT Scheme Manual, Seventh Edition, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, Mass., September 1984.

[Ossanna 1979]
J. F. Ossanna, *Nroff/Troff User's Manual*, UNIX Programmer's Manual, Seventh Edition, vol. 2, Bell Telephone Laboratories, Murray Hill, NJ, January 1979.

[Ousterhout 1990]
John K. Ousterhout, *Tcl: An Embeddable Command Language*, Proceedings of the USENIX 1990 Winter Conference, January 1990, pp. 133-146.

[Scheifler et al. 1986]
Robert W. Scheifler and Jim Gettys, *The X Window System*, ACM Transactions on Graphics, vol. 5, no. 2, pp. 79-109, 1986.

[Springer et al. 1989]
George Springer and Daniel O. Friedman, *Scheme and the Art of Programming*, MIT Press, Cambridge, Mass., 1989.

[Stallman 1981]
Richard M. Stallman, *EMACS − The Extensible, Customizable, Self-documenting Display Editor Production System*, SIGPLAN Notices, vol. 16, no. 6, pp. 147-156, Association for Computing Machinery, New York, 1981.

[Yip 1991]
G. May Yip, *Incremental, Generational Mostly-Copying Garbage Collection in Uncooperative Environments*, WRL Research Report 91/8, DEC Western Research Laboratory, Palo

Alto, California, 1991.

Appendix A: Extending Elk – An Example

The "ndbm" Library Extension

The extensibility mechanisms of Elk can be demonstrated best by a simple library extension. Consider the *ndbm* library that is available on most versions of UNIX. This library implements functions to maintain a simple database file of key/contents pairs.

As shown in Listing 4, both the keys and the data to be stored are described by the type *datum*; it consists of the data (a string of bytes) and the length of the data. *dbm_open()* opens a database file and returns a handle to that file to be used in subsequent operations on that database (a pointer to an opaque data type, similar to the *fopen* and *readdir* interfaces); it returns a null pointer if the file could not be opened. A database is closed by a call to *dbm_close()*. The data stored under a given key is accessed by the function *dbm_fetch()*; it returns an object of type *datum* (with a null *dptr* if the key could not be found). *dbm_store* is used to insert an entry into a database and to modify an existing entry; it returns zero on success and a non-zero value on error.

```
#include <ndbm.h>

typedef struct {
    char *dptr;
    int dsize;
} datum;

DBM *dbm_open(char *file, int flags, int mode);

void dbm_close(DBM *db);

datum dbm_fetch(DBM *db, datum key);

int dbm_store(DBM *db, datum key, datum data, int flags);
```

Listing 4: The *ndbm* library

Note: For simplicity, several functions have been omitted.
The *flags* and *mode* arguments of *dbm_open* are that of the *open* system call. The *flags* argument of *dbm_store* can be DBM_INSERT to insert a new entry into the database or DBM_REPLACE to change an existing entry.

The straightforward way to write an *ndbm* extension to Elk is to provide a new Scheme data type *dbm-file* together with the obligatory type predicate *dbm-file?* and the Scheme primitive procedures *dbm-open*, *dbm-close*, *dbm-fetch* and *dbm-store* that operate on objects of type *dbm-file*.

dbm-open receives the filename (a string or a symbol); the second argument is one of the symbols *reader* (open the file read-only), *writer* (read and write access), and *create* (read and write access, create new file if it does not exist). The optional filemode argument is an integer. *dbm-open* returns an object of type *dbm-file* or #f (false) if the file could not be opened. *dbm-close* closes the database file associated with its argument of type *dbm-file*. It returns a nonprinting object.

dbm-fetch expects a *dbm-file* and a string argument (the key to be searched) and returns a string (the data stored under the key) or #f if the key does not exist. Note that in Elk strings may contain arbitrary 8-bit characters, including the null byte. *dbm-store* is called with a *dbm-file*, two strings (key and data) and one of the symbols *insert* and *replace*. Its integer return value is the return value of *dbm_store()*.

These procedures and the new *dbm-file* type can be used by application programmers to manipulate database files in those parts of their applications that are written in Scheme. Listing 5 shows a small example.

```
(define expand-mail-alias
  (lambda (alias)
    (let ((d (dbm-open "/etc/aliases" 'reader)))
      (if (not d)
          (error 'expand-mail-alias "cannot open database"))
    (unwind-protect
      (dbm-fetch d alias)
      (dbm-close d)))))

(define address-of-staff (expand-mail-alias "staff"))
```

Listing 5: Using the ndbm extension

Note: The *unwind-protect* and the *error* form are not present in standard Scheme.

The Anatomy of a Scheme Type

Listing 6 shows the part of the extension that deals with the new data type *dbm-file* and the extension initialization function. The variable *T_Dbm* will hold the unique identifier of the newly defined type. The structure *S_Dbm* defines the C representation of the type; one such C structure is declared for each composite Scheme type. Its main component is the handle of the database file that is contained in each object of type *dbm-file*.

Scheme objects can usually live longer than their underlying C objects. In case of the *dbm-file* type, a Scheme object of that type can apparently still be used after its database handle has been closed by a call to *dbm-close*. As Elk extensions must not crash the application, we must prevent such stale objects from being used in further calls to *dbm-fetch*, *dbm-store*, and *dbm-close*. One way to achieve this is to record in each Scheme object whether the underlying C object is still alive or has been terminated. The boolean component *alive* in the *dbm-file* type serves this purpose. It is initialized with true and is set to false in *dbm-close*. Further operations on objects with *alive* being false are rejected.

The interpreter stores all Scheme objects in variables of type *Object*. An *Object* is typically a 32-bit value; it is composed of a *tag* part and a *pointer* part. The *tag* part indicates the type of the object, and the remaining bits hold the actual memory address of the object (they point into the

```
#include <scheme.h>
#include <ndbm.h>

int T_Dbm;

struct S_Dbm {
    DBM *dbm;
    char alive;   /* 0 or 1 */
};

#define DBMF(obj) ((struct S_Dbm *)POINTER(obj))

int Dbm_Equal(a, b) Object a, b; {
    return DBMF(a)->alive && DBMF(b)->alive && DBMF(a)->dbm == DBMF(b)->dbm;
}

void Dbm_Print(d, port) Object d, port; {
    Printf(port, "#[dbm-file %lu]", DBMF(d)->dbm);
}

Object P_Is_Dbm(x) Object x; {
    return TYPE(x) == T_Dbm ? True : False;
}

void init_dbm() {
    Define_Primitive(P_Is_Dbm,    "dbm-file?", 1, 1, EVAL);
    Define_Primitive(P_Dbm_Open,  "dbm-open",  2, 3, VARARGS);
    Define_Primitive(P_Dbm_Close, "dbm-close", 1, 1, EVAL);
    Define_Primitive(P_Dbm_Store, "dbm-store", 4, 4, EVAL);
    Define_Primitive(P_Dbm_Fetch, "dbm-fetch", 2, 2, EVAL);

    T_Dbm = Define_Type("dbm-file", sizeof(struct S_Dbm),
        Dbm_Equal, Dbm_Equal, Dbm_Print, NOFUNC);
}
```

Listing 6: Skeleton of the ndbm extension

Note: For simplicity some details have been omitted in this listing, and the calling interface of some functions has been simplified; the program would not compile in this form. A working *gdbm* (GNU dbm) extension is included in the Elk distribution.

interpreter's heap). The macros *TYPE* and *POINTER* are provided to extract the fields of an *Object*. Each type definition must define a macro to extract the object's memory address from an *Object* (by means of *POINTER*) and then cast it into a pointer to the underlying C structure (see *#define DBMF* in Listing 6).

Dbm_Equal() implements both the *eqv?* and the *equal?* predicates for *dbm-file* objects; it returns true if both objects being compared are alive and contain identical *DBM* handles.

Dbm_Print() is called by the interpreter each time an object of type *dbm-file* is to be printed; it is invoked with the object and the Scheme port to which the output is to be sent.

P_Is_Dbm() implements the primitive procedure *dbm-file?* (the type predicate). As with all primitives, it receives arguments of type *Object* and returns an *Object*, and it has a name beginning with "P_".

The definition of the initialization function *init_dbm()* is straightforward; it invokes *Define_Primitive()* once for each primitive procedure and finally *Define_Type()* to make the new type known to the interpreter.

Primitive Procedures – The Details

Listing 7 gives the definitions of the primitives *dbm-open* and *dbm-close*.

dbm-open, as it has an optional argument, is a function with *VARARGS* calling discipline (not to be confused with the C language feature of the same name). This is indicated by the last argument to the corresponding call to *Define_Primitive* in the extension initialization. Primitives of this type receive an array of *Objects* and a count.

The initial call to the macro *Make_C_String* checks if the first argument to *dbm-open* is a string (or a symbol) and converts it to a C string. To obtain the second argument to *dbm_open()*, the symbol passed to the Scheme primitive (*reader*, *writer*, etc.) has to be mapped to a corresponding flags combination (*O_RDONLY*, *O_RDWR*, etc.). This is accomplished by the function *Symbols_To_Bits()*; it is invoked with a Scheme symbol, a flag indicating whether a single symbol or a list of symbols (a mask) is to be converted, and a table of pairs of symbol names and C integers. The third argument to *dbm_open* is the filemode; *Get_Integer()* converts a Scheme number to a C integer. *dbm-open* finally allocates a new Scheme object of type *T_Dbm* on the heap, initializes the components of the object, and returns it.

The auxiliary function *Check_Dbm()* is used by the remaining primitives to check whether a given object is of type *dbm-file* and if so, whether it is stale. In this case an error is signaled; *Primitive_Error()* enters the error handler of Elk.

P_Dbm_Close() just marks the object as stale by setting *alive* to false and closes the database file.

Listing 8 shows the implementation of *dbm-store* and *dbm-fetch*. *Make_Integer()* is the pendant to *Get_Integer()*; it converts a C integer into a Scheme number. Likewise, *Make_String()* converts a C string into a Scheme string.

```
static SYMDESCR Flag_Syms[] = {
    { "reader", O_RDONLY },
    { "writer", O_RDWR },
    { "create", O_RDWR|O_CREAT },
    { 0, 0 }
};

Object P_Dbm_Open(argc, argv) int argc; Object *argv; {
    char *p;
    DBM *dp;
    Object d;

    Make_C_String(argv[0], p);
    dp = dbm_open(p, Symbols_To_Bits(argv[1], 0, Flag_Syms),
        argc == 3 ? Get_Integer(argv[2]) : 0644);
    if (dp == 0)
        return False;
    d = Alloc_Object(sizeof(struct S_Dbm), T_Dbm, 0);
    DBMF(d)->dbm = dp;
    DBMF(d)->alive = 1;
    return d;
}

void Check_Dbm(d) Object d; {
    Check_Type(d, T_Dbm);
    if (!DBMF(d)->alive)
        Primitive_Error("invalid dbm-file: ~s", d);
}

Object P_Dbm_Close(d) Object d; {
    Check_Dbm(d);
    DBMF(d)->alive = 0;
    dbm_close(DBMF(d)->dbm);
    return Void;
}
```

Listing 7: ndbm extension – implementation of *dbm-open* and *dbm-close*

```
static SYMDESCR Store_Syms[] = {
    { "insert",  DBM_INSERT },
    { "replace", DBM_REPLACE },
    { 0, 0 }
};

Object P_Dbm_Store(d, key, content, flag) Object d, key, content, flag; {
    datum k, c;
    int result;

    Check_Dbm(d);
    Check_Type(key, T_String);
    Check_Type(content, T_String);
    k.dptr = STRING(key)->data;       k.dsize = STRING(key)->size;
    c.dptr = STRING(content)->data;   c.dsize = STRING(content)->size;
    result = dbm_store(DBMF(d)->dbm, k, c,
        Symbols_To_Bits(flag, 0, Store_Syms));
    return Make_Integer(result);
}

Object P_Dbm_Fetch(d, key) Object d, key; {
    datum k, c;

    Check_Dbm(d);
    Check_Type(key, T_String);
    k.dptr = STRING(key)->data;       k.dsize = STRING(key)->size;
    c = dbm_fetch(DBMF(d)->dbm, k);
    return c.dptr ? Make_String(c.dptr, c.dsize) : False;
}
```

Listing 8: ndbm extension – implementation of *dbm-store* and *dbm-fetch*