Building Extensible Applications with Elk —

C/C++ Programmer's Manual

Oliver Laumann

ABSTRACT

Elk (*Extension Language Kit*) is a Scheme implementation designed as an embeddable, reusable extension language subsystem for integration into existing and future applications written in C or C++. The programmer's interface to Elk provides for a close interworking of the C/C++ parts of Elk-based, *hybrid* applications with extensible Scheme code. This manual describes the facilities of the C/C++ programmer's interface that can be used by authors of extensible applications and Scheme extensions. Topics range from the architecture of Elk-based applications and the definition of application-specific Scheme types and primitives to more advanced subjects such as weak data structures and interacting with the garbage collector. Many examples throughout the text illustrate the facilities and techniques discussed in this manual.

1. Additional Documentation

The official specification of the Scheme programming language is the ''R^{4}RS'' (William Clinger and Jonathan Rees (editors), *Revised^4 Report on the Algorithmic Language Scheme*, 1991). A slightly modified version of an earlier revision of this report was adopted as an IEEE an ANSI standard in 1990 (IEEE Std 1178-1990, *IEEE Standard for the Scheme Programming Language*, 1991).

The dialect of Scheme implemented by Elk (a superset of the official language) is described in the *Reference Manual for the Elk Extension Language Interpreter* that is included in the Elk distribution as troff source and preformatted PostScript files. Reference manuals for the various predefined Elk extensions (such as the UNIX and X11 extensions) are also part of the distribution; see the file ''doc/README'' for an overview of the available documentation.

This manual supersedes the document *Interfacing Scheme to the ''Real World''* that was included in earlier versions of Elk.

An article about Elk has appeared in USENIX Computing Systems in 1994 (Oliver Laumann and Carsten Bormann, Elk: The Extension Language Kit, *USENIX Computing Systems*, vol. 7, no. 4, pp. 419–449).

A recent example of an application that uses Elk as its extension language implementation is freely available in source and binary form as *http://www.informatik.uni-bremen.de/˜net/unroff*. *unroff* is a programmable, extensible troff translator with Scheme-based back-ends for the

Hypertext Markup Language. The source code shown in Appendix B has been directly taken from the *unroff* source; authors of Elk-based applications are encourage to reuse this and other parts of the *unroff* source for their own projects.

2. Introduction

This manual can be roughly divided into two parts. The first part (chapters 3 to 6) describes the architectural aspects of Elk-based applications and Elk extensions. Facilities and tools for building extensible applications with Elk are introduced here. Readers who are already familiar with the concepts explained in this part of the document may want to skip it and begin reading at chapter 7 or later. The second part (covering chapters 7 to 12) specifies the C functions and types available to application programmers and describes techniques for building data structures that can be interfaced to Scheme in an efficient way. Appendix C briefly summarizes all the functions, macros, types, and variables exported by the Elk kernel to the C/C++ programmer.

Here is a short overview of the remaining chapters of this manual. Chapter 3 discusses the architecture of extensible applications based on Elk and their relation to Elk extensions. Chapter 4 provides an overview of the two basic methods for integrating an application (or extensions) with Elk: dynamic loading and static linking. Chapter 5 describes use of dynamic loading in more detail; topics include automatic extension initialization and C++ static constructors embedded in dynamically loaded modules. Chapter 6 describes several forms of linking user-supplied code with Elk statically and how these affect the structure of an application's *main()* function.

The remaining chapters are a complete specification of the functions and types of the C/C++ programmer's interface to Elk. Chapter 7 provides introductory notes and advice for programmers of C/C++ code interfacing to Elk (use of include files, predefined preprocessor symbols, etc.). Chapter 8 describes the anatomy of Scheme objects from the C/C++ programmer's point of view. Chapter 9 explains how applications and extensions can define new Scheme primitives. Chapter 10 presents the standard, built-in Scheme types implemented by Elk (numbers, pairs, vectors, etc.) and functions for creating and accessing Scheme objects of these types from within C/C++ code. The facilities for defining new, first-class Scheme data types are described in chapter 11. Finally, chapter 12 deals with a number of more advanced topics, such as functions for interacting with the garbage collector, automatic finalization of inaccessible objects, definition of user-supplied reader functions, error handling, etc.

A note on the naming conventions followed by the C identifiers used throughout this document: the names of all functions, macros, types, and variables exported by Elk have their components separated by underscores and capitalized (as in *Register_Object()*, for example). In contrast, the names defined by examples shown in this manual only use lower case letters, so that they can be distinguished easily from predefined functions exported by Elk.

3. The Architecture of Extensible Applications

Extensible applications built with Elk are *hybrid* in that they consist of code written in a mixture of languages—code written in the application's *implementation language* (C or C++) and code written in the *extension language* (Scheme). An application of this kind is usually composed of two layers, a low-level C/C++ layer that provides the basic, performance-critical functionality of the application, and on top of that a higher-level layer which is written in Scheme and interpreted at runtime.

The Scheme-language portion of an Elk-based application may range from just a few dozen lines of Scheme code (if a simple form of customization is sufficient) to fifty percent of the application or more (if a high degree of extensibility is required). As Scheme code is interpreted at runtime by an interpreter embedded in the application, users can customize and modify the application's Scheme layer or add and test their own Scheme procedures; recompilation, access to the C/C++ source, or knowledge of the implementation language are not required. Therefore, an application can achieve highest extensibility by restricting its low-level part to just a small core of time-critical C/C++ code.

To enable extensions to "work on" an application's internal data structures and state, the application core exports a set of new, application-specific Scheme data types and primitives operating on them to the Scheme layer. These types and primitives can be thought of as a "wrapper" around some of the C/C++ types and functions used by the application's core. For example, the core of an Elk-based newsreader program would export first-class Scheme types representing *newsgroups*, *subscriptions*, and *news articles*; these types would encapsulate the corresponding low-level C "structs" or C++ classes. In addition, it would export a number of Scheme primitives to operate on these types—to create members of them (e. g. by reading a news article from disk), to present them to the user through the application's user-interface, etc. Each of these primitives would recur on one or more corresponding C or C++ functions implementing the functionality in an efficient way.

Another job of the low-level C/C++ layer of an application is to hide platform-specific or system-specific details by providing suitable abstractions, so that the Scheme part can be kept portable and simple. For example, in case of the newsreader program, extension writers should not have to care about whether the news articles are stored in a local file system or retrieved from a network server, or about the idiosyncrasies of the system's networking facilities. Most of these system-specific details can be better dealt with in a language oriented towards systems programming, such as C, than in Scheme.

To decide whether to make a function part of the low-level part of an application or to write it in the extension language, you may ask yourself the following questions:

- *Is the function performance-critical?*

  If the answer to this question is *yes*, put the function into the C/C++ core. For example, in case of the newsreader application, a primitive to search all articles in a given newsgroup for a pattern is certainly performance-critical and would therefore be written in the implementation language, while a function to ask the user to select an item from a list of newsgroups is not time-critical and could be written Scheme.

- *Does the function have to deal with platform-specific details?*

  For example, a function that needs to allocate and open a UNIX pseudo-tty or to establish a network connection needs to care about numerous system-specific details and different kinds of operating system facilities and will therefore be written in C/C++ rather than in Scheme.

- *In which language can the function be expressed more "naturally"?*

  A function that parses and tokenizes a string can be expressed more naturally (that is, in a significantly more concise and efficient way) in a language such as C than in Scheme. On the other hand, functions to construct trees of news articles, to traverse them, and to apply a function to each node are obvious candidates for writing them in a Lisp-like language (Scheme).

- *Are customizability and extensibility important?*

  If it is likely that the application's users will want to customize or augment a function or even replace it with their own versions, write it in the extension language. If, for some reason, this is impossible or not practicable, at least provide suitable "hooks" that enable users to influence the function's operation from within Scheme code.

3.1. Scheme Extensions

In addition to the Scheme interpreter component, Elk consists of a number of *Scheme extensions*. These extensions are not specific to any kind application and are therefore reusable. They provide the "glue" between Scheme and a number of external libraries, in particular the X11 libraries and the UNIX C library (exceptions are the record extension and the bitstring extension which provide a functionality of their own). The purpose of these extensions is to make the functionality of the external libraries (for example, the UNIX system calls) available to Scheme as Scheme data types and primitives operating on them.

While the Scheme extensions are useful for writing freestanding Scheme programs (e. g. for rapid prototyping of X11-based Scheme programs), their main job is to help building applications that need to interface to external libraries on the extension language level. The X11 extensions, for instance, are intended to be used by applications with a graphical user interface based on the X window system. By linking the X11 extensions (in addition to the Scheme interpreter) with an Elk-based application, the application's user interface can be written entirely in Scheme and will therefore be inherently customizable and extensible. As the Scheme extensions are reusable and can be shared between applications, extension language code can be written in a portable manner.

3.2. Applications versus Extensions

As far as the C/C++ programmer's interface to Elk (that is, the subject of this manual) is concerned, there is not really a technical difference between Scheme *extensions* on the one hand (such as the X11 extensions), and Elk-based, extensible *applications* on the other hand. Both are composed of an efficient, low-level C/C++ core and, above that, a higher-level layer written in Scheme. In both cases, the C/C++ layer exports a set of Scheme types and primitives to the Scheme layer (that is, to the Scheme *programmer*) and thus needs to interact with the Scheme interpreter. Because of this analogy, the rest of the manual will mostly drop the distinction between applications and extensions and concentrate on the interface between C/C++ and Elk.

The only noteworthy difference between applications and extensions is that the former tend to have their own *main()* function that gains control on startup, while Scheme extensions do not have a *main()* entry point—they are usually loaded into the interpreter (or application) during runtime. This distinction will become important in the next chapter, when the different ways of joining Elk and C/C++ code will be discussed.

4. Linking Applications and Extensions with Elk

There are two different mechanisms for integrating compiled C/C++ code (extensions or an application) with Elk: *static linking* and *dynamic loading*. The object files that make up an Elk-based application are usually linked statically with the Scheme interpreter in the normal way to produce an executable program. Compiled extensions, on the other hand, are usually dynamically loaded into the running Scheme interpreter as they are needed. These conventions reflect the normal case; Scheme extensions may as well be linked statically with the interpreter

- to produce a "specialized" instance of the interpreter (for example, when developing X11-based Scheme code, an extended version of the interpreter may be produced by linking it statically with the X11 extensions);
- if a particular extension is required by an application from the beginning (an application with an X-based user-interface would be linked with the X11 extensions statically, as loading on-demand would not be useful in this case);
- on the (few) platforms where dynamic loading is not supported or where dynamic loading has a large performance overhead.

Likewise, dynamic loading is not only useful for on-demand loading of reusable Scheme extensions; *applications* can benefit from this facility as well. To reduce the size of the final executable, parts of an application may loaded dynamically rather than linked statically if they are used infrequently or if only a few of them are used at a time. Dynamic loading enables the author of an extensible application to decompose it into an arbitrary number of individual parts as an alternative to combining them statically into a large, monolithic executable. An extensible newsreader program, for example, may include a separate spelling check module that is dynamically loaded the first time it is needed (i. e. when a newly written news article is to be spell-checked).

The capability to dynamically load compiled C/C++ code into a running application enables users to write *hybrid extensions* which consist of a low-level C/C++ part and a high-level part written in Scheme. As a result, extensions can execute much faster (extensions to the Emacs editor, for example, must be entirely written in Emacs-Lisp and can therefore become slow if sufficiently complex); and extensions can deal more easily with low-level, platform-specific details.

5. Dynamic Loading

Object files (compiled C/C++ code) are loaded by means of the standard *load* primitive of Scheme, just like ordinary Scheme files. All you need to do is to compile your C or C++ source file, apply the *makedl* script that comes with the Elk distribution to the resulting object file, and load it into the interpreter or application. *makedl* prepares object files for dynamic loading (which is a no-op on most platforms) and combines several object files into one to speed up loading; arguments are the output file and one or more input files or additional libraries (input and output file may be identical):

```
% cc -c -I/usr/elk/include file.c
% /usr/elk/lib/makedl file.o file.o
% scheme
> (load 'file.o)
>
```

(This examples assumes that Elk has been installed under "/usr/elk" on your site. Additional arguments may be required for the call to *cc*.)

Elk does not attempt to discriminate object code and Scheme code based on the files' contents; the names of object files are required to end in ".o", the standard suffix for object modules in UNIX. Scheme files, on the other hand, end in ".scm" by convention. This convention is not enforced by Elk—everything that is not an object file is considered to be a Scheme file. A list of object files may be passed to the *load* primitive which may save time on platforms where a call to the system linker is involved.

Loading object files directly as shown above is uncommon. Instead, the Scheme part of a hybrid extension usually loads its corresponding object file (and all the other files that are required) automatically, so that one can write, for example,

```
(require 'unix)
```

to load the UNIX extension. This expression causes the file *unix.scm* to be loaded, which then loads the object file *unix.o*—the UNIX extension's low-level part—automatically on startup. Additional *load-libraries* (as explained in the next section) may be set by the Scheme file immediately before loading the extension's object file.

When an object file is loaded, unresolved references are resolved against the symbols exported by the running interpreter or by the combination of an application and the interpreter (the *base program*). This is an essential feature, as dynamically loaded extensions must be able to reference the elementary Scheme primitives defined by the interpreter core and all the other functions that are available to the extension/application programmer. In addition, references are resolved against the symbols exported by all previously loaded object files. The term *incremental loading* is used for this style of dynamic loading, as it allows building complex applications from small components incrementally.

5.1. Load Libraries

Dynamically loadable object files usually have unresolved references into one or more libraries, most likely at least into the standard C library. Therefore, when loading an object file, references are resolved not only against the base program and previously loaded object files, but also against a number of user-supplied *load libraries*. The X11 extensions of Elk, for instance, need to be linked against the respective libraries of the X window system, such as *libX11* and *libXt*. These load libraries can be assigned to the Scheme variable *load-libraries* which is bound in the top-level environment of Elk. Typically, *load-libraries* is dynamically assigned a set of library names by means of *fluid-let* immediately before calling *load*. For example, the Xlib extension (*xlib.scm*) contains code such as

```
(fluid-let
  ((load-libraries
     (string-append "-L/usr/X11/lib -lX11 " load-libraries)))
  (load 'xlib.o))
```

to load the accompanying object file (*xlib.o*), linking it against the system's X library in addition to whatever libraries were already in use at that point. The default value of *load-libraries* is ‘‘−lc’’ (i. e. the C library), as extensions are likely to use functions from this library in addition to those C library functions that have already been linked into the base program or have been pulled in by previously loaded object files. By using *string-append* in the example above, the specified libraries are added to the default value of *load-libraries* rather than overwriting it. The exact syntax of the load libraries is platform specific. For instance, ‘‘−L/usr/X11/lib’’ as used above is recognized by the system linker of most UNIX variants as an option indicating in which directory the libraries reside on the system, but different options or additional libraries are required on certain platforms (as specified by the platform's ‘‘config/site’’ file in the Elk distribution).

5.2. Extension Initializers and Finalizers

When loading an object file, Elk scans the file's symbol table for the names of extension initialization functions or *extension initializer*s. These extension initializers are the initial entry points to the newly loaded extension; their names must have the prefix ‘‘elk_init_’’ (earlier the prefix ‘‘init_’’ was used; it was changed in Elk 3.0 to avoid name conflicts). Each extension initializer found in the object file is invoked to pass control to the extension. The job of the extension initializers is to register the Scheme types and primitives defined by the extension with the interpreter and to perform any dynamic initializations.

As each extension may have an arbitrary number of initialization functions rather than one single function with a fixed name, extension writers can divide their extensions into a number of independent modules, each of which provides its own initialization function. The compiled modules can then be combined into one dynamically loadable object file without having to lump all initializations into a central initialization function.

In the same manner, extension can define an arbitrary number of *extension finalization functions* which are called on termination of the Scheme interpreter or application. The names of finalization functions begin with ‘‘elk_finit_’’. Extension finalization functions are typically used for clean-up operations such as removing temporary files.

The extension initializers (as well as the finalizers) are called in an unspecified order.

5.3. C++ Static Constructors and Destructors

In addition to calling extension initialization functions, the *load* primitives invokes all C++ static constructors that are present in the dynamically loaded object file in case it contains compiled C++ code. Likewise, C++ static destructors are called automatically on termination. The constructors and destructors are called in an unspecified order, but all constructors (destructors) are called before calling any extension initializers (finalizers). Elk recognizes the function name prefixes of static constructor and destructor functions used by all major UNIX C++ compilers; new prefixes can be added if required.

6. Static Linking

Linking user-supplied code with Elk statically can be used as an alternative to dynamic loading on platforms that do not support it, for applications with their own *main()*, and to avoid the overhead of loading frequently used Elk extensions. Dynamic loading and static linking may be used in combination— additional object files can be loaded in a running executable formed by linking the Scheme interpreter with extensions or with an application (or parts thereof).

When making the Scheme interpreter component of Elk, these executables and object files get installed (relative to your *install_dir* which usually is ''/usr/elk'' or ''/usr/local/elk''):

bin/scheme
: The freestanding, plain Scheme interpreter.

lib/standalone.o
: The Scheme interpreter as a relocatable object file which can be linked with user-supplied object files to form an executable. This object file contains a *main()* function; thus the Scheme interpreter starts up in the normal way when the executable is invoked.

lib/module.o
: Like *standalone.o*, except that the object file does not export its own *main()* function. Therefore, the object files linked with it have to supply a *main()*.

The object file *standalone.o* is typically linked with a number of Elk extensions (e. g. the X11 extensions), while *module.o* is used by Elk-based applications which contribute their own *main()* and need to be ''in control'' on startup.

6.1. Linking the Scheme Interpreter with Extensions

A shell script *linkscheme* (installed as ''lib/linkscheme'') simplifies combining the Scheme interpreter with a number of—user-supplied or predefined—extensions statically. This script is called with the name of the output file (the resulting executable) and any number of object files and libraries. It basically links the object files and libraries with ''standalone.o'' and supplies any additional libraries that may be required by the interpreter. In general, this can be done just as well by calling the linker or compiler directly, but *linkscheme* also takes care of additional processing that needs to be performed on at least one platform (currently AIX).

To create an instance of Elk including the Xlib, Xt, and Xaw extensions, *linkscheme* would be used as follows (again assuming you have installed the software under ''/usr/elk''):

```
% cd /usr/elk
% lib/linkscheme x11scheme runtime/obj/xlib.o runtime/obj/xt.o \
     runtime/obj/xaw/*.o -lXaw -lXmu -lXt -lSM -lICE -lX11 -lXext
```

The exact form of the libraries depends on your platform and X11 version; for example, additional options may be required if X11 is not installed in a standard location at your site. *xlib.o* is the Xlib extension, *xt.o* is the X toolkit intrinsics (Xt) extension, and the subdirectory *xaw* holds the object files for all the Athena widgets. The executable *x11scheme* can now be used to run arbitrary X11 applications using the Athena widgets without requiring any runtime loading of object files belonging to the X11 extensions:

```
% x11scheme
> (load '../examples/xaw/dialog.scm)
[Autoloading xwidgets.scm]
[Autoloading xt.scm]
[Autoloading siteinfo.scm]
...
```

In the same way, *linkscheme* can be used to link the Scheme interpreter with any new, user-supplied extensions, with parts of an Elk-based application, or with any combination thereof.

6.1.1. Automatic Extension Initialization

When linking Elk with extensions, it is *not* necessary to add calls to the extension initializers to the Scheme interpreter's *main()* function and recompile the interpreter; all extensions are initialized automatically on startup. To accomplish this kind of automatic initialization, Elk scans its own symbol table on startup, invoking any ''elk_init_'' functions and C++ static constructors, in the same way the symbol table of object files is scanned when they are dynamically loaded. Extension finalizers and C++ static destructors are saved for calling on exit. Automatic extension initialization only works if

- the executable file has a symbol table (i. e. you must not strip it)
- the executable file can be opened for reading
- the interpreter can locate its executable file by scanning the shell's directory search path.

The performance overhead caused by the initial scanning of the symbol is small; the program's symbol table can be read or mapped into memory efficiently (it it has not been automatically mapped into the address space by the operating system in the first place).

6.2. Linking the Scheme Interpreter with an Application

Elk-based applications that have their own *main()* are linked with the Scheme interpreter installed as *module.o* which, unlike *standalone.o*, does not export a *main()* function. No special *linkscheme* script is required to link with *module.o*; application writers usually will add ''/usr/elk/lib/module.o'' (or whatever the correct path is) to the list of object files in their Makefile. To simplify linking with Elk, a trivial script *ldflags* (which lives in ''lib'' along with *linkscheme*) is supplied that just echoes any additional libraries required by the Scheme interpreter. Application developers may use *ldflags* in their Makefiles.

As *module.o* does not have a *main()* entry point, an application using it must initialize the interpreter from within its own *main()*. This is done by calling . *Elk_Init()*:

```
void Elk_Init(int argc, char **argv, int init_flag, char *filename);
```

Elk_Init() is only defined by *module.o* and is essentially a ''wrapper'' around the Scheme interpreter's *main()*. *argc* and *argv* are the arguments to be passed to the Scheme interpreter's *main()*. These may or may not be the calling program's original arguments; however, *argv[0]* must be that from the calling program in any case (because its address is used by Elk to determine the program's stack base). If *init_flag* is nonzero, the interpreter scans its symbol table to invoke extension initializers as described in 6.1.1. C++ static constructors, however, are never invoked by *module.o* (regarless of *init_flag*), because they are already taken care of by the runtime startup in this case. If *filename* is nonzero, it is the name of Scheme file to be loaded by *Elk_Init()*.

6.2.1. An Example "main()" Function

Figure 1 shows a realistic (yet somewhat simplified) example *main()* function of an application using Elk.

```
char *directory;

int main(int ac, char **av) {
    char **eav;
    int eac = 1, c;

    Set_App_Name(av[0]);
    eav = safe_malloc((ac+2+1) * sizeof(char *));    /* ac + −p xxx + 0 */
    eav[0] = av[0];
    while ((c = getopt(ac, av, "gh:o")) != EOF) switch (c) {
        case ’o’:
            process option...
        case ’g’:
            eav[eac++] = "−g"; break;
        case ’h’:
            eav[eac++] = "−h"; eav[eac++] = optarg; break;
        case ’?’:
            usage(); return 1;
    }
    if ((directory = getenv("APP_DIR")) == 0)
        directory = DEFAULT_DIR;
    eav[eac++] = "−p";
    eav[eac] = safe_malloc(strlen(directory) + 11);
    sprintf(eav[eac++], ".:%s/elk/scm", directory);
    eav[eac] = 0;
    Elk_Init(eac, eav, 0, 0);

    initialize application’s modules...

    boot_code();

    application’s main loop (if written in C)
    ...
```

Figure 1: Example *main()* of an Elk-based application (simplified)

The code shown in the example must construct a new argument vector to be passed to *Elk_Init()*, because the application has command line options of its own (just −*o* in the example).

Two Elk-options (−*g* and −*h*) are handed to *Elk_Init()* if present, so that a mixture of Elk-specific and application-specific options can be given (see the manual page for the Scheme interpreter for the meaning of Elk's options). (*safe_malloc()* is assumed to be a wrapper around *malloc()* with proper error-checking.) *Set_App_Name()* is provided by Elk and is called with a name to be displayed in front of fatal error messages by the interpreter.

When all the options have been parsed, an additional option −*p* is synthesized to provide a minimal initial *load-path* for Elk. This load-path consists of the current directory and a subdirectory of the directory under which the application expects its files that are needed during runtime. An environment variable can be used to set this directory. Defining a load-path like this has the benefit that a minimal, self-contained Elk runtime environment (e. g. a toplevel and the debugger) can be shipped with binary distributions of the application so that users are not required to have Elk installed at their sites.

When Elk has been initialized by calling *Elk_Init()*, the application may initialize all its other modules and finally load an initial Scheme file that "boots" the Scheme part of the application (which may involve loading further Scheme files). This initial Scheme file may be quite simple and just define a few functions used later, or it main contain the application's entire "driving logic" or interactive user-interface. This is accomplished by a function *boot_code()* which may as simple as this:

```
void boot_code(void) {
     char *fn = safe_malloc(strlen(directory) + 30);

     sprintf(fn, "%s/scm/app.scm", directory);
     Set_Error_Tag("initial load");
     Load_File(fn);
     free(fn);
}
```

Load_File() is defined by Elk and loads a Scheme file whose name is supplied as a C string. *Set_Error_Tag()* may be used by extensions and applications to define the symbol that is passed as the first argument to the standard error handler when a Scheme error is signaled (see section 12.5).

6.3. Who is in Control?

When an application's object files are loaded into the interpreter dynamically or are linked with the interpreter using *linkscheme*, control initially rests in the interpreter. In contrast, when the application is linked using *module.o* and *Elk_Init()* as shown in the previous section, it defines its own *main()* function, and hence the application is "in control" on startup.

From a technical point of view, it does not really make a difference whether control rests in the interpreter or in the application initially. In the first case, the main "driving logic" (or "main loop") of the application can simply be wrapped in a Scheme primitive which is then called by the Scheme toplevel on startup to pass control back to the application, if this is desired. In any case, control usually changes frequently between the Scheme interpreter and the actual application anyway—the Scheme interpreter invokes callback functions or Scheme primitives provided by the application, which may in turn invoke Scheme procedures or load Scheme files, and so on.

The Tcl-like style of use, where control rests in the C-part of the application most of the time, and where this C code ‘‘calls out’’ to the interpreter occasionally by passing it an extension language expression or a small script, is not typical for Elk. It is supported, though; Elk provides a simple extension to pass a Scheme expression to the interpreter as a C string and receive the result in the same form, similar to what *Tcl_Eval()* does in Tcl (see section 12.2). In a typical Elk-based application the extension language serves as the ‘‘backbone’’ of the application: the application’s driving logic or main loop is written entirely in Scheme, and this Scheme code calls out to the application’s C layer, using the data types, primitives, and other callbacks exported to the extension language by the application. With the help of the X11 extensions, the entire (graphical) user interface of an application can be written in Scheme easily; control can then passed to the application’s C/C++ layer whenever an Xt callback is triggered. In this case, the application’s ‘‘main loop’’ consists of a call to the Scheme primitive corresponding to the X toolkit function *XtAppMainLoop()* (the main event dispatch loop).

7. Notes for Writing C/C++ Code Using Elk

This chapter describes general conventions and usage notes for Elk-based C/C++ code and introduces a few useful facilities that are not directly related to Scheme.

7.1. Elk Include Files

Every C or C++ file using functions, macros, or variables defined by Elk must include the file *scheme.h*:

```
#include <scheme.h>        or:        #include "scheme.h"
```

This include file resides in a subdirectory *include* of the directory where Elk has been installed on your system. You must insert a suitable −I option into your Makefiles to add this directory to the C compiler’s search path. ‘‘scheme.h’’ includes several other Elk-specific include files from the same directory and, in addition, the standard C include files *<stdio.h>* and *<signal.h>*.

7.2. Standard C and Function Prototypes

All the examples shown in this manual are written in ANSI/ISO C. This assumes that the Elk include files have been installed with function prototypes enabled. Whether or not function prototypes are enabled is controlled by a definition in the platform- and compiler-specific ‘‘config/system’’ file that has been selected for configuring Elk. However, if the include files have function prototypes disabled, prototypes are enable automatically if you are compiling your code with a C compiler that defines the symbol ‘‘__STDC__’’ as non-zero, or with a C++ compiler that defines ‘‘__cplusplus’’[1].

---

[1] Although the public include files provided by Elk can be used by C++ code, Elk itself cannot be compiled with a C++ compiler. The interpreter has been written in C to maximize portability.

Elk include files that have been installed with function prototypes disabled can also be “upgraded” by defining the symbol “WANT_PROTOTYPES” before including “scheme.h”. Similarly, include files installed without function prototypes can be used with a non-ANSI C compiler by defining the symbol “NO_PROTOTYPES” before including “scheme.h”.

7.3. External Symbols Defined by Elk

As extensions or applications are linked with Elk (regarless of whether dynamic loading or static linking is used), they can in general reference all external symbols exported by Elk. Of these, only the symbols described in this manual may be used safely. Use of other (private) symbols results in non-portable code, as the symbols may change their meaning or may even be removed from future releases of Elk. The same restriction applies to the macros and types defined by the include files of Elk.

In addition to the symbols defined by the Scheme interpreter kernel, those exported by other Scheme extensions that are present in the same executable (or have been loaded earlier) can be referenced from within C/C++ code. These extensions are not subject of this manual; you should refer to the relevant documentation and the public include files that are part of the extensions.

If Elk is linked with an application that has its own *main()* function, none of the functions exported by Elk must be used before the initial call to *Elk_Init()* (except *Set_App_Name()*).

7.4. Calling Scheme Primitives

A large subset of the symbols exported by the Scheme interpreter is the set of functions implementing the Scheme primitives. These may be used safely by extensions and applications. There exists one C function for each Scheme primitive. Its name is that of the corresponding primitive with the following conversions applied:

- dashes are replaced by underscores, and the initial letters of the resulting word components are capitalized;
- the prefix “P_” is prepended;
- “−>” is replaced by “_To_” (as in *vector−>list*);
- a trailing exclamation mark is deleted, except for *append!* and *reverse!*, where “_Set” is appended;
- a trailing question mark is replaced by the letter ‘p’ (except for *eq?, eqv?, equal?* and the string and character comparison primitives, where it is deleted);

The names of a few functions are derived differently as shown by this table:

Scheme Primitive	C Function
<	P_Generic_Less()
>	P_Generic_Greater()
=	P_Generic_Equal()
<=	P_Generic_Eq_Less()
>=	P_Generic_Eq_Greater()
1+	P_Inc()
1− and −1+	P_Dec()
+	P_Generic_Plus()
−	P_Generic_Minus()
*	P_Generic_Multiply()
/	P_Generic_Divide()
let*	P_Letseq()

According to these rules, the primitive *exact−>inexact* can be used from within C as *P_Exact_To_Inexact()*, the predicate *integer?* is available as *P_Integerp()*, etc. Authors of reusable Scheme extensions are encouraged to follow these (or similar) naming conventions in their code.

All the functions implementing Scheme primitives (as well as special forms, which are treated as primitives in Elk) receive Scheme objects or arrays thereof as their arguments and return Scheme objects as their values. The underlying C type will be described in the next chapter. For the semantics of the non-standard Scheme primitives defined by Elk refer to the Reference Manual for the interpreter.

7.5. Portable alloca()

Elk provides a portable variant of *alloca()* as a set of macros that can be used by extensions and applications. *alloca()*, which is supported by most modern UNIX systems and C compilers, allocates memory in the caller's stack frame; the memory is automatically released when the function returns. Elk simulates this functionality on the (rare) platforms where *alloca()* is not available.

To allocate memory, the macro *Alloca()* is called with a variable to which the newly allocated memory is assigned, the type of that variable, and the number of bytes that are requested. The macro *Alloca_End* must be called (without an argument list) before returning from a function or block that uses *Alloca()*; this macro is empty on those platforms that support the ordinary *alloca()*. Finally, a call to the macro *Alloca_Begin* must be placed in the function's declarations. *Alloca()* usually is more efficient than *malloc()* and *free()*, and the memory need not be freed when the function is left prematurely because of an interrupt or by calling a continuation.

As an example, here is the skeleton of a function that is called with a filename prefix and a suffix, concatenates them (separated by a period), and opens the resulting file:

```
int some_function(char *prefix, char *suffix) {
    char *name;
    int len, fd;
    Alloca_Begin;

    len = strlen(prefix) + 1 + strlen(suffix) + 1;
    Alloca(name, char*, len);
    sprintf(name, "%s.%s", prefix, suffix);
    fd = open(name, ...);
    ...
    Alloca_End;
}
```

7.6. Other Useful Macros and Functions

The preprocessor symbols ELK_MAJOR and ELK_MINOR expand to the major and minor version number of the current release of Elk. They did not exist in versions older than Elk 3.0.

index(), *bcopy()*, *bcmp()*, and *bzero()* are defined as suitable macros on systems that do not have them in their C library; they may be used by source files that include "scheme.h", regardless of the actual platform.

Code linked with Elk may use the two functions

```
char *Safe_Malloc(unsigned size);
char *Safe_Realloc(char *old_pointer, unsigned size);
```

as alternatives to *malloc()* and *realloc()*. If the request for memory cannot be satisfied, the standard Elk error handler is called with a suitable error message.

8. The Anatomy of Scheme Objects

All Scheme objects, regarless of their Scheme type, are represented as instances of the type *Object* in C. *Object* is implemented as a small C *struct* in newer Elk releases and was an integral type earlier. However, code using Elk should not assume a specific representation, as it may change again in future revisions. An *Object* consists of three components:

- the type of the corresponding Scheme object as a small integer (the "type field" or "tag field"),
- the contents of the object, either directly (for small objects) or as a pointer into the Scheme heap (the "pointer field"),
- a "const bit" which, if set, indicates that the object is read-only and cannot be modified by destructive Scheme primitives.

Elk defines a few macros to retrieve and modify the fields of an *Object* independent of its representation:

```
TYPE(obj)           ISCONST(obj)        SET(obj,t,ptr)
POINTER(obj)        SETCONST(obj)
```

TYPE() returns the contents of the type field of an *Object*; *POINTER()* returns the contents of the pointer field as an *unsigned long* (different macros are provided for types which have their values stored directly in the *Object* rather than in the heap); *ISCONST()* returns the value of the const bit; and *SETCONST()* sets the const bit to 1 (it cannot be cleared once it has been set). *ISCONST()* and *SETCONST()* may only be applied to *Objects* that have their value stored on the heap (such as vectors, strings, etc.); all other types of Scheme objects are *ipso facto* read-only. Another macro, *SET()*, can be used to set both the type and pointer field of a new object.

Two objects can be compared by means of the macro *EQ()*, which is also used as the basis for the Scheme predicate *eq?*:

```
EQ(obj1,obj2)
```

EQ() expands to a non-zero value if the type fields and the pointer fields of the two objects are identical, else zero (regardless of whether the pointer field really holds a pointer or the object's actual value). As *EQ()* may evaluate its arguments twice, it should not be invoked with function calls or complex expressions.

8.1. Type-specific Macros

For each predefined Scheme type, there exists a preprocessor symbol that expands to the integer value of that type (the contents of the type field of members of the type). The name of each such symbol is the name of the type with the prefix "T_":

```
T_Boolean     T_Pair     T_Vector     etc...
```

These symbols are typically used as case labels in switch-statements to discriminate the possible types of a given object, or in if-statements to check whether a Scheme object is of a given type:

```
if (TYPE(obj) == T_Vector)
     ...
```

In addition, each type defines a macro to extract the contents of an object of that type and to convert it to the correct C type. For example, the macro

```
CHAR(obj)
```

is used to fetch the character value (a C *int*) from members of the Scheme type *character*, that is, from objects whose type field contains the value *T_Character*. Similarly, the macro

```
VECTOR(obj)
```

gets the heap pointer conveyed in objects of the Scheme type *vector*. For objects such as vectors, pairs, and procedures, the heap address is coerced to a pointer to a C *struct* defining the layout of the object. There exists one structure type declaration for each such Scheme type; their names are that of the type with "S_" prepended. For example, *VECTOR()* returns a pointer to a structure with the components *size* (the number of elements in the vector) and *data* (the elements as an array of *Objects*). These can be used from within C code like this:

```
int i, num = VECTOR(obj)->size;

for (i = 0; i < num; i++)
     VECTOR(obj)->data[i] = ...;
```

Similarly, the structure underlying the Scheme type *pair* is defined as:

```
struct S_Pair { Object car, cdr; };
```

and the macro *PAIR()* returns a (heap) pointer to a member of the structure *S_Pair*. Macros such as *VECTOR()* and *PAIR()* just convert the contents of the pointer field to a pointer of the correct type:

```
#define VECTOR(obj)    ((struct S_Vector *)POINTER(obj))
#define PAIR(obj)      ((struct S_Pair   *)POINTER(obj))
```

Authors of Scheme extensions and Elk-based applications are encouraged to follow these conventions in their code and, for each new type *xyz*, store the new type value (which is allocated by the interpreter when the type is registered) in a variable *T_Xyz*, and define a structure or class *S_Xyz*, and a macro *XYZ()* that makes a pointer to this structure from a member of the type. Capitalization may vary according to personal preference.

9. Defining New Scheme Primitives

In Elk, there exists a one-to-one relationship between Scheme primitives and C functions: each Scheme primitive—whether predefined or user-defined—is implemented by a corresponding C function. This includes special forms, which are treated as a special kind of primitives in Elk. Extensions and applications use the function *Define_Primitive()* to register a new Scheme primitive with the interpreter, supplying its name and the C function that implements it. In case of dynamically loadable extensions or application modules, the calls to *Define_Primitive()* are placed in the extension initialization functions that are called automatically as the object file is loaded. *Define_Primitive()* is declared as

```
void Define_Primitive((Object (*func)()), const char *name,
                      int minargs, int maxargs,
                      enum discipline disc);
```

The arguments are:

func a pointer to the C function implementing the new primitive;

name the name of the primitive as a null-terminated C string;

minargs
the minimum number of arguments accepted by the primitive;

maxargs
the maximum number of arguments (identical to *minargs* in most cases);

disc the *calling discipline* (usually *EVAL*).

Define_Primitive() creates a Scheme variable of the specified name in the current (i. e. the caller's) lexical environment and binds it to the newly created procedure. Each C function that implements a primitive has a return type of *Object* and, for a calling discipline of *EVAL*, zero or more arguments of type *Object* which are bound to the evaluated arguments passed to the Scheme primitive when it is called. The calling discipline must be one of the following:

EVAL
: The primitive expects a fixed number of arguments; *minargs* and *maxargs* must be identical[2].

VARARGS
: The primitive has a variable number of arguments, and the underlying C function is called with an argument count and an array of arguments. Defining primitives with a variable number of arguments will explained in more detail in section 9.2.

NOEVAL
: The arguments are passed as a Scheme list of unevaluated objects—a single argument of the type *Object*. Primitives using this discipline will then use *Eval()* as described in section 12.2 to evaluate some or all of the arguments. *NOEVAL* is only rarely used (with the exception of the built-in special forms of Elk); extensions and applications mostly use macros as a more convenient way to defined new syntactical forms.

Figure 2 shows a simple example for defining a new Scheme primitive.

```
#include "scheme.h"

Object p_vector_reverse(Object vec) {
     Object tmp, *s, *t;

     Check_Type(vec, T_Vector);
     for (s = VECTOR(vec)->data, t = s+VECTOR(vec)->size; --t > s; s++)
          tmp = *s, *s = *t, *t = tmp;
     return vec;
}

void elk_init_vector(void) {
     Define_Primitive(p_vector_reverse, "vector-reverse!", 1, 1, EVAL);
}
```

Figure 2: Defining a new Scheme Primitive

The primitive *vector-reverse!* defined by the example extension reverses the elements of a Scheme vector in place and returns its argument (note the final exclamation mark indicating the destructive operation). *Check_Type()* is a simple macro that compares the type field of the first argument (an *Object*) with the second argument and signals and error if they do not match. This macro is used primarily for type-checking the arguments to Scheme primitives. A call to the

[2] Because of a limitation in the C language, primitives of type *EVAL* can only have a fixed maximum number of arguments (currently 10). If more arguments are required, *VARARGS* must be used instead.

macro *Check_Mutable()* with the vector as an argument could have been inserted before the loop to check whether the vector is read-only and to automatically raise an error if this is the case. The example code forms a complete extension including an extension initialization function and could be linked with the interpreter, or loaded dynamically into the interpreter as follows:

```
% cc -c -I/usr/elk/include vec.c; makedl vec.o vec.o
% scheme
> (load 'vec.o)
> (define v '#(hello word))
v
> (vector-reverse! v)
#(world hello)
> v
#(world hello)
>
```

9.1. Making Objects Known to the Garbage Collector

Consider the non-destructive version of the primitive *vector-reverse* shown in Figure 3, which returns a new vector instead of altering the contents of the original vector.

```
Object p_vector_reverse(Object vec) {
     Object ret;
     int i, j;

     Check_Type(vec, T_Vector);
     ret = Make_Vector(VECTOR(vec)->size, False);
     for (i = 0, j = VECTOR(vec)->size; --j >= 0; i++)
          VECTOR(ret)->data[i] = VECTOR(vec)->data[j];
     return ret;
}
```

Figure 3: Non-destructive Scheme primitive *vector-reverse*

The code in Figure 3 is identical to that shown in Figure 2, except that a new vector is allocated, filled with the contents of the original vector in reverse order, and returned as the result of the primitive. *Make_Vector()* is declared by Elk:

```
Object Make_Vector(int size, Object fill);
```

size is the length of the vector, and all elements are initialized to the Scheme object *fill*. In the example, the predefined global variable *False* is used as the *fill* object; it holds the boolean Scheme constant #f (any *Object* could have been used here).

Although the C function may look right, there is a problem when it comes to garbage collection. To understand the problem and its solution, it may be helpful to have a brief look at how the garbage collector[3] works (the following description presents a simplified view; the real algorithm is more complex). In Elk, a garbage collection is triggered automatically whenever a request for heap space cannot be satisfied because the heap is full, or explicitly by calling the primitive *collect* from within Scheme code. The garbage collector traces all ‘‘live’’ objects starting with a known *root set* of pointers to reachable objects (basically the interpreter’s global lexical environment and its symbol table). Following these pointers, all accessible Scheme objects are located and copied to a new heap space in memory (‘‘forwarded’’), thereby compacting the heap. Whenever an object is relocated in memory during garbage collection, the contents of the pointer field of the corresponding C *Object* is updated to point to the new location. After that, any constituent objects (e. g. the elements of a vector) are forwarded in the same way.

As live objects are relocated in memory, *all* pointers to an object need to be updated properly when that object is forwarded during garbage collection. If a pointer to a live object were not in the root set (that is, not reachable by the garbage collector), the object would either become garbage erroneously during the next garbage collection, or, if it had been reached through some other pointer, the original pointer would now point to an invalid location.[4] This is exactly what happens in the example shown in Figure 3.

The call to *Make_Vector()* in the example triggers a garbage collection if the heap is too full to satisfy the request for heap space. As the *Object* pointer stored in the argument *vec* is invisible to the garbage collector, its pointer field cannot be updated when the vector to which it points is forwarded during the garbage collection started inside *Make_Vector()*. As a result, all further references to *VECTOR(vec)* will return an invalid address and may cause the program to crash (immediately or, worse, at a later point). The solution is simple: the primitive just needs to add *vec* to the set of initial pointers used by the garbage collector. This is done by inserting the line

```
GC_Link(vec);
```

at the beginning of the function before the call to *Make_Vector()*. *GC_Link()* is a macro. Another macro, *GC_Unlink*, must be called later (e. g. at the end of the function) without an argument list to remove the object from the root set again. In addition, a call to *GC_Node* (again without an argument list) must be placed in the declarations at the beginning of the enclosing function or block. Figure 4 shows the revised, correct code.

Appendix A lists the C functions which can trigger a garbage collection. Any local variable or argument of type *Object* must be protected in the manner shown above if one of these functions is called during its lifetime. This may sound more burdensome than it really is, because most of the ‘‘dangerous’’ functions are rarely or never used from within C/C++ extensions or

---

[3] Elk actually employs two garbage collectors, one based on the traditional stop-and-copy strategy, and a generational, incremental garbage collector which is less disruptive but not supported on all platforms.

[4] The problem of managing an ‘‘exact root set’’ can be avoided by a technique called *conservative* garbage collection. A conservative garbage collector treats the data segment, stack, and registers of the running program as *ambiguous roots*. If the set of ambiguous roots is a superset of the *actual* roots, then a pointer that looks like a heap pointer can safely be considered as pointing to an accessible object that cannot be reclaimed. At the time Elk was designed, conservative GC was still in its infancy and sufficient experience did not exist. For this reason, and because of the implied risks on certain machine architectures, the inherent portability problems, and the inability to precisely determine the actual memory utilization, a traditional GC strategy was chosen for Elk.

```
Object p_vector_reverse(Object vec) {
     Object ret;
     int i, j;
     GC_Node;

     GC_Link(vec);
     Check_Type(vec, T_Vector);
     ret = Make_Vector(VECTOR(vec)->size, False);
     for (i = 0, j = VECTOR(vec)->size; --j >= 0; i++)
          VECTOR(ret)->data[i] = VECTOR(vec)->data[j];
     GC_Unlink;
     return ret;
}
```

Figure 4: Non-destructive Scheme primitive *vector-reverse*, corrected version

applications in practice. Most primitives that require calls to *GC_Link()* use some function that creates a new Scheme object, such as *Make_Vector()* in the example above.

To simplify GC protection of more than a single argument or variable, additional macros *GC_Link2()*, *GC_Link3()*, and so on up to *GC_Link7()* are provided. Each of these can be called with as many arguments of type *Object* as is indicated by the digit (separate macros are required, because macros with a variable number of arguments cannot be defined in C). A corresponding macro *GC_Node2*, *GC_Node3*, and so on, must be placed in the declarations. Different *GC_Link*()* calls cannot be mixed. All local variables passed to one of the macros must have been initialized. GC protection is not required for ‘‘pointer-less’’ objects such as booleans and small integers, and for the arguments of primitives with a variable number of arguments (as described in section 9.2). Section 12.3 will describe how global (external) *Object* variables can be added to the root set.

Here is how the implementation of the primitive *cons* uses *GC_Link2()* to protect its arguments (the car and the cdr of the new pair):

```
Object P_Cons(Object car, Object cdr) {
     Object new_pair;
     GC_Node2;

     GC_Link2(car, cdr);
     new_pair = allocate heap space and initialize object;
     GC_Unlink;
     return new_pair;
}
```

There are a few pitfalls to be aware of when using ‘‘dangerous’’ functions from within your C/C++ code. For example, consider this code fragment which fills a Scheme vector with the program’s environment strings that are available through the null-terminated string array *environ[]*:

```
Object vec = new vector of the right size;
int i;
GC_Node;

GC_Link(vec);
for (i = 0; environ[i] != 0; i++)
     VECTOR(vec)->data[i] = Make_String(environ[i], strlen(environ[i]));
```

(*Make_String()* creates and initializes a new Scheme string.) The body of the for-loop contains a subtle bug: depending on the compiler used, the left hand side of the assignment (the expression involving *vec*) may be evaluated before *Make_String()* is invoked. As a result, a copy of the contents of *vec* might be, for instance, stored in a register before a garbage collection is triggered while evaluating the right hand side of the assignment. The garbage collector would then move the vector object in memory, updating the—properly GC-protected—variable *vec*, but not the temporary copy in the register, which is now a dangling reference. To avoid this, the loop must be modified along these lines:

```
for (i = 0; environ[i]; i++) {
     Object temp = Make_String(environ[i], strlen(environ[i]));
     VECTOR(vec)->data[i] = temp;
}
```

A related pitfall to watch out for is exemplified by this code fragment:

```
Object obj;
...
GC_Link(obj);
...
some_function(obj, P_Cons(car, cdr));
```

Here, the call to *P_Cons()*—just like *Make_String()* above—can trigger a garbage collection. Depending on the C compiler, the properly GC-protected object pointer *obj* may be pushed on the argument stack before *P_Cons()* is invoked, as the order in which function arguments—just like the operands of the assignment operator—are evaluated is undefined in the C language. In this case, if a garbage collection takes place and the heap object to which *obj* points is moved, *obj* will be updated properly, but the copy on the stack will not. Again, the problem can be avoided easily by assigning the result of the nested function call to a temporary *Object* variable and use this variable in the enclosing function call:

```
temp = P_Cons(car, cdr);
some_function(obj, temp);
```

9.2. Primitives with Variable-Length Argument Lists

Primitives with a variable number of arguments are registered with the interpreter by calling *Define_Primitive()* with the calling discipline *VARARGS* and with different values for *minargs* and *maxargs*. The special symbol *MANY* can be given as the maximum number of arguments to indicate that there is no upper limit on the primitive's number of actual arguments. The C/C++ function implementing a primitive with a variable number of arguments is called with two arguments: an integer count that specifies the number of actual arguments, and the Scheme arguments as an array of *Objects* (that is, a pointer to *Object*). The objects passed as the argument vector of *VARARGS* primitives are already registered with the garbage collector; calls to *GC_Link()* are not required. As an example for a primitive with an arbitrary number of arguments, here is the definition of a simplified variant of *append!* (which does not handle empty lists):

```
Object p_append_set (int argc, Object *argv); {
     int i;

     for (i = 0; i < argc-1; i++)
          (void)P_Set_Cdr (P_Last_Pair (argv[i]), argv[i+1]);
     return *argv;
}
```

The corresponding call to *Define_Primitive()* would read:

```
Define_Primitive(p_append_set, "append!", 0, MANY, VARARGS);
```

Besides implementing primitives with an indefinite maximum number of arguments, the *VARARGS* discipline is frequently used for primitives with an optional argument. For example, a primitive encapsulating the UNIX *open()* system call, which has two fixed arguments (filename, flags) and an optional third argument (the mode for newly created files, i. e. calls with the flag *O_CREAT*), could be defined as follows:

```
Object p_unix_open(int argc, Object *argv) {
     char *name = get_file_name(argv[0]);
     int flags = get_flags(argv[1]);
     mode_t mode;

     if (flags & O_CREAT) {
          if (argc < 3)
               error--too few arguments
          mode = get_mode(argv[2]);
          ...
```

The call to *Define_Primitive()* could then be written as:

```
Define_Primitive(p_unix_open, "unix-open", 2, 3, VARARGS);
```

10. Predefined Scheme Types

This chapter introduces the Scheme types predefined by Elk. It begins with the “pointerless” types such as boolean, whose values are stored directly in the pointer field of an *Object*; followed by the types whose members are C *structs* that reside on the Scheme heap.

10.1. Booleans (T_Boolean)

Objects of type *T_Boolean* can hold the values #t and #f. Two *Objects* initialized to #t and #f, respectively, are available as the external C variables *True* and *False*. The macro

```
Truep(obj)
```

can be used to check whether an arbitrary Scheme object is regarded as true. Use of *Truep()* is not necessarily equivalent to

```
!EQ(obj,False)
```

because the empty list may count as false in addition to #f if backwards compatibility to older Scheme language versions has been enabled. *Truep()* may evaluate its argument twice and should therefore not be invoked with a function call or a complex expression.

The two functions

```
int Eqv(Object, Object);
int Equal(Object, Object);
```

are identical to the primitives *P_Eqv()* and *P_Equal()*, except that they return a C integer rather than a Scheme boolean and therefore can be used more conveniently in C/C++.

10.2. Characters (T_Character)

The character value stored in an *Object* of type *T_Character* can be obtained by the macro

```
CHAR(char_obj)
```

as a non-negative *int*. A new character object is created by calling the function

```
Object Make_Char(int c);
```

The predefined external C variable *Newline* holds the newline character as a Scheme *Object*.

10.3. Empty List (T_Null)

The type *T_Null* has exactly one member—the empty list; hence all *Objects* of this type are identical. The empty list is available as the external C variable *Null*. This variable is often used to initialize *Objects* that will be assigned their real values later, for example, as the fill element for newly created vectors or to initialize *Objects* in order to *GC_Link()* them. A macro *Nullp()* is provided as a shorthand for checking if an *Object* is the empty list:

```
#define Nullp(obj)  (TYPE(obj) == T_Null)
```

This macro is used frequently in the termination condition of for-loops that scan a Scheme list:

```
Object tail;
...
for (tail = some_list; !Nullp(tail); tail = Cdr(tail))
     process_element(Car(tail));
```

(*Car()* and *Cdr()* essentially are shorthands for *P_Car()* and *P_Cdr()* and will be revisited in the section on pairs).

10.4. End of File (T_End_Of_File)

The type *T_End_Of_File* has one member—the end-of-file object—and is only rarely used from within user-supplied C/C++ code. The external C variable *Eof* is initialized to the end-of-file object.

10.5. Integers (T_Fixnum and T_Bignum)

Integers come in two flavors: *fixnums* and *bignums*. The former have their value stored directly in the pointer field and are wide enough to hold most C *ints*. Bignums can hold integers of arbitrary size and are stored in the heap. Two macros are provided to test whether a given signed (or unsigned, respectively) integer fits into a fixnum:

```
FIXNUM_FITS(integer)
UFIXNUM_FITS(unsigned_integer)
```

The former always returns 1 in Elk 3.0, but the range of integer values that can be represented as a fixnum may be restricted in future revisions. It is guaranteed, however, that at least two bits less than the machine's word size will be available for fixnums in future versions of Elk.

The value stored in a fixnum can be obtained as a C *int* by calling the macro

```
FIXNUM(fixnum_obj)
```

A macro

```
Check_Integer(obj)
```

can be used as a shorthand for checking whether an *Object* is a fixnum or a bignum and raising an error otherwise.

The following functions are provided to convert C integers to Scheme integers:

```
Object Make_Integer(int);
Object Make_Unsigned(unsigned);
Object Make_Long(long);
Object Make_Unsigned_Long(unsigned long);
```

Make_Integer() returns a fixnum object if *FIXNUM_FITS()* returns true for the argument, otherwise a bignum. Likewise, *Make_Long()* usually returns a fixnum but may have to resort to bignums on architectures where a C *long* is wider than an *int*. *Make_Unsigned()* returns a bignum if the specified integer is larger than the largest positive *int* that fits into a fixnum (*UFIXNUM_FITS()* returns zero in this case). Another set of functions convert a Scheme number to a C integer:

```
int Get_Integer(Object);
int Get_Exact_Integer(Object);

unsigned Get_Unsigned(Object);
unsigned Get_Exact_Unsigned(Object);

long Get_Long(Object);
long Get_Exact_Long(Object);

unsigned long Get_Unsigned_Long(Object);
unsigned long Get_Exact_Unsigned_Long(Object);
```

These functions signal an error if one of the following conditions is true:

- the argument is neither a fixnum, nor a bignum, nor a flonum (real number) with a fractional part of zero (more about flonums in the next section);
- the function is one of the "unsigned" variants and the argument is a negative number;
- the argument is a bignum too large for the respective return type;
- the function is one of the "exact" variants and the argument is neither a fixnum nor a bignum;
- the argument is a flonum that cannot be coerced to the respective return type.

As all of the above functions include suitable type-checks, primitives receiving integer arguments can be written in a simple and straightforward way. For example, a primitive encapsulating the UNIX *dup* system call (which returns an integer file descriptor pointing to the same file as the original one) can be written as:

```
Object p_unix_dup(Object fd) {
    return Make_Integer(dup(Get_Exact_Unsigned(fd)));
```

Note that if *Get_Unsigned()* (or *Get_Integer()*) had been used here in place of the "exact" conversion function, it would be possible to write expressions such as:

```
(define fd (unix-dup (truncate 1.2)))
```

10.6. Floating Point Numbers (T_Flonum)

Real and inexact numbers are represented as *Objects* of type *T_Flonum*. Each such object holds a pointer to a structure on the heap with a component *val* of type *double*, so that the expression

```
FLONUM(flonum_obj)->val
```

can be used to obtain the *double* value. To convert a Scheme number to a *double* regardless of its type, the more general function

```
double Get_Double(Object);
```

can be used. It raises an error if the argument is not a fixnum, bignum, or flonum, or if it is a bignum too large to fit into a *double*.

The functions

```
Object Make_Flonum(double);
Object Make_Reduced_Flonum(double);
```

convert a C *double* to a flonum; the latter returns a fixnum if the *double* is small enough to fit into a fixnum and has a fractional part of zero. The macro

```
Check_Number(obj)
```

checks whether the given *Object* is a number (that is, a fixnum, bignum, or flonum in the current revision of Elk) and raises an error otherwise.

10.7. Pairs (T_Pair)

Pairs have two components of type *Object*, the car and the cdr, that can be accessed as:

```
PAIR(pair_obj)->car
PAIR(pair_obj)->cdr
```

Two macros *Car()* and *Cdr()* are provided as shorthands for these expressions, and another macro *Cons()* can be used in place of *P_Cons()* to create a new pair. The macro

```
Check_List(obj)
```

checks whether the specified *Object* is either a pair or the empty list and signals an error otherwise. The predefined function

```
int Fast_Length(Object list);
```

can be used to compute the length of the given Scheme list. This function is more efficient than the primitive *P_Length()*, because it neither checks the type of the argument nor whether the given list is proper, and the result need not be converted to a Scheme number. The function

```
Object Copy_List(Object list);
```

returns a copy of the specified list (including all its sublists).

As explained in section 9.1, care must be taken when mixing calls to these macros, because *Cons()* may trigger a garbage collection: an expression such as

```
Car(x) = Cons(y, z);
```

is wrong, even if *x* is properly “GC_Linked”, and should be replaced by

```
tmp = Cons(x, y);
Car(x) = tmp;
```

or a similar sequence.

10.8. Symbols (T_Symbol)

Objects of type *T_Symbol* have one public component—the symbol’s name as a Scheme string (that is, an *Object* of type *T_String*):

```
SYMBOL(symbol_obj)->name
```

A new symbol can be created by calling one of the functions

```
Object Intern(const char *);
Object CI_Intern(const char *);
```

with the new symbol's name as the argument. *CI_Intern()* is the case-insensitive variant of *Intern()*; it maps all upper case characters to lower case. *EQ()* yields true for all *Objects* returned by calls to *Intern()* with strings with the same contents (or calls to *CI_Intern()* with strings that are identical after case conversion). This is the main property that distinguishes symbols from strings in Scheme.

A symbol that is used by more than one function can be stored in a global variable to save calls to *Intern()*. This can be done using the convenience function

```
void Define_Symbol(Object *var, const char *name);
```

Define_Symbol() is called with the address of a variable where the newly-interned symbol is stored and the name of the symbol to be handed to *Intern()*. The function adds the new symbol to the garbage collector's root set to make it reachable (as described in section 12.3. Example:

```
static Object sym_else;
...
void elk_init_example(void) {
     Define_Symbol(&sym_else, "else");
     ...
}
```

10.8.1. The Non-Printing Symbol

By convention, Scheme primitives that do not have a useful return value (for example the output primitives) return the "non-printing symbol" in Elk. The name of this symbol consists of the empty string; it does not produce any output when it is printed, for example, by the toplevel read-eval-print loop. In Scheme code, the non-printing symbol can be generated by using the reader syntax "#v" or by calling *string−>symbol* with the empty string. On the C language level, the non-printing symbol is available as the external variable *Void*, so that primitives lacking a useful return value can use

```
return Void;
```

10.9. Strings (T_String)

Objects of type string have two components—the length and the contents of the string as a pointer to *char*:

```
STRING(string_obj)->size
STRING(string_obj)->data
```

The *data* component is not null-terminated, as a string itself may contain a null-byte as a valid character in Elk. A Scheme string is created by calling the function

```
Object Make_String(const char *init, int size);
```

size is the length of the newly-created string. *init* is either the null-pointer or a pointer to *size* characters that are copied into the new Scheme string. For example, the sequence

```
Object str;
...
str = Make_String(0, 100);
bzero(STRING(str)->data, 100);
```

generates a string holding 100 null-bytes.

Most primitives that receive a Scheme string as one of their arguments pass the string's contents to a C function (for example a C library function) that expects an ordinary, null-terminated C string. For this purpose Elk provides a function

```
char *Get_String(Object);
```

that returns the contents of the Scheme string argument as a null-terminated C string. An error is raised if the argument is not a string. *Get_String()* has to create a copy of the contents of the Scheme string in order to append the null-character. To avoid requiring the caller to provide and release space for the copy, *Get_String()* operates on and returns NUMSTRBUFS internal, cyclically reused buffers (the value of NUMSTRBUFS is 3 in Elk 3.0). Consequently, no more than NUMSTRBUFS results of *Get_String()* can be used simultaneously (which is rarely a problem in practice). As an example, a Scheme primitive that calls the C library function *getenv()* and returns #f on error can be written as

```
Object p_getenv(Object name) {
     char *ret = getenv(Get_String(name));
     return ret ? Make_String(ret, strlen(ret)) : False;
}
```

If more strings are to be used simultaneously, the macro *Get_String_Stack()* can be used instead. It is called with the Scheme object and the name of a variable of type "char*" to which the C string will be assigned. *Get_String_Stack()* allocates space by means of *Alloca()* (as explained in section 7.5); hence a call to *Alloca_Begin* must be placed in the declarations of the enclosing function or block, and *Alloca_End* must be called before returning from it.

An additional function *Get_Strsym()* and an additional macro *Get_Strsym_Stack()* are provided by Elk; these are identical to *Get_String()* and *Get_String_Stack()*, respectively, except that the Scheme object may also be a symbol. In this case, the symbol's name is taken as the string to be converted.

As an example for the use of *Get_String_Stack()*, here is a simple Scheme primitive *exec* that is called with the name of a program and one more more arguments and passes them to the *execv()* system call:

```
Object p_exec(int argc, Object *argv) {
     char **argp; int i;
     Alloca_Begin;

     Alloca(argp, char**, argc*sizeof(char *));
     for (i = 1; i < argc; i++)
          Get_String_Stack(argv[i], argp[i-1]);
     argp[i-1] = 0;
     execv(Get_String(*argv), argp);   /* must not return */
     error...
}

elk_init_example() {
     Define_Primitive(p_exec, "exec", 2, MANY, VARARGS);
}
```

The primitive can be used as follows:

```
(exec "/bin/ls" "ls" "-l")
```

Get_String() could not be used in this primitive, because the number of string arguments may exceed the number of static buffers maintained by *Get_String()*.

10.10. Vectors (T_Vector)

The layout of *Objects* of type vector is identical to that of strings, except that the *data* component is an array of *Objects*. A function *Make_Vector()* creates a new vector as has been explained in section 9.1 above.

10.11. Ports (T_Port)

The components of *Objects* of type *T_Port* are not normally accessed directly from within C/C++ code, except for

```
PORT(port_obj)->closefun
```

which is a pointer to a function receiving an argument of type “FILE*” (for example, a pointer to *fclose()*), provided that the port is a file port. It is called automatically whenever the port is closed, either because *close-input-port* or *close-output-port* is applied to it or because the garbage collector has determined that the port is no longer reachable.

A new file port is created by calling

```
Object Make_Port(int flags, FILE *f, Object name);
```

with a first argument of either zero (output port), *P_INPUT* (input port) or *P_BIDIR* (bidirectional port), the file pointer, and the name of the file as a Scheme string. The macros

```
Check_Input_Port(obj)
Check_Output_Port(obj)
```

check whether the specified port is open and is capable of input (or output, respectively); an error is raised otherwise.

To arrange for a newly-created port to be closed automatically when it becomes garbage, it must be passed to the function *Register_Object()* as follows:

```
Register_Object(the_port, 0, Terminate_File, 0);
```

Register_Object() will be described in section 12.4. The current input and output port as well as ports pointing to the program's initial standard input and output are available as four external variables of type *Object*:

```
Curr_Input_Port       Standard_Input_Port
Curr_Output_Port      Standard_Output_Port
```

The function

```
void Reset_IO(int destructive_flag);
```

clears any input queued at the current input port, then flushes the current output port (if *destructive_flag* is zero) or discards characters queued at the output port (if *destructive_flag* is non-zero), and finally resets the current input and current output port to their initial values (the program's standard input and standard output). This function is typically used in error situations to reset the current ports to a defined state.

In addition to the standard Scheme primitives for output, extensions and applications can use a function

```
void Printf(Object port, char *fmt, ...);
```

to send output to a Scheme port using C *printf*. The first argument to *Printf()* is the Scheme port to which the output will be sent (it must be an output port); the remaining arguments are that of the C library function *printf()*.

To output a Scheme object, the following function can be used in addition to the usual primitives:

```
void Print_Object(Object obj, Object port, int raw_flag,
           int print_depth, int print_length);
```

The arguments to *Print_Object()* are identical to the arguments of the "print function" that must be supplied for each user-defined Scheme type (as described in section 11: the *Object* to be printed, the output port, a flag indicating that the object should be printed in human-readable form (*display* sets the flag, *write* does not), and the "print depth" and "print length" for that operation. For debugging purposes, the macro

```
Print(obj);
```

may be used to output an *Object* to the current output port.

A function

```
void Load_Source_Port(Object port);
```

can be used to load Scheme expressions from a file that has already been opened as a Scheme port.

10.12. Miscellaneous Types

Other built-in Scheme types are lexical environments, primitive procedures, compound procedures, macros, continuations (also called ‘‘control points’’ at a few places in Elk), and promises. These types are not normally created or manipulated from within C or C++ code. If you are writing a specialized extension that depends on the C representation of these types, refer to the declarations in the public include file ‘‘object.h’’ (which is included automatically via ‘‘scheme.h’’).

Lexical environments are identical to pairs except that the type is *T_Environment* rather than *T_Pair*. The current environment and the initial (gobal) environment are available as the external C variables *The_Environment* and *Global_Environment*. The predefined type constants for primitives, compound procedures (the results of evaluating lambda expressions), and macros are *T_Primitive*, *T_Compound*, and *T_Macro*, respectively. The function

```
void Check_Procedure(Object);
```

checks whether the specified object is either a compound procedure or a primitive procedure with a calling discipline different from *NOEVAL* and raises an error otherwise. The type constant for continuations is *T_Control*. ‘‘Promise’’ is the type of object returned by the special form *delay*; the corresponding type constant is named *T_Promise*.

11. Defining New Scheme Types

A new, disjoint Scheme type is registered with Elk by calling the function *Define_Type()*, similar to *Define_Primitive()* for new primitives. Making a new type known to Elk involves passing it information about the underlying C/C++ representation of the type and a number of C or C++ functions that are ‘‘called back’’ by the interpreter in various situations to pass control to the code that implements the type. The prototype of *Define_Type()* is:

```
int Define_Type(int zero, const char *name,
     int (*size)(Object), int const_size,
     int (*eqv)(Object, Object),
     int (*equal)(Object, Object),
     int (*print)(Object, Object, int, int, int),
     int (*visit)(Object*, int (*)(Object*)));
```

The arguments to *Define_Primitive()* are in detail:

zero
: The first argument must be zero (in early versions of Elk it could be used to request a fixed, predefined type number for the new type);

name
: The name of the new type.

size, const_size
: The size of the corresponding C type (usually a *struct*) in bytes, given as one of two, mutually-exclusive arguments: *size*, a pointer to a function called by the interpreter to determine the size of an object (for types whose individual members are of different sizes, such as the *vector* type); and *const_size*, the size as a constant (for all other types). A null-pointer is

given for *const_size* if *size* is to be used instead.

eqv, equal

Pointers to (callback) functions that are invoked by the interpreter whenever the Scheme predicate *equal?*, or *eqv?* respectively, is applied to members of the newly defined type. As an application-defined type is opaque from the interpreter's point of view, the equality predicates have to be supplied by the application or extension. Each of these (boolean) functions is passed two objects of the new type as arguments when called back.

print

A pointer to a function that is used by the interpreter to print a member of this type. When calling the print function, the interpreter passes as arguments the Scheme object to be printed, a Scheme *port* to which the output is to be sent, a flag indicating whether output is to be rendered in human-readable form (*display* Scheme primitive) or machine-readable, read-write-invariance preserving form (*write*), and finally the current remainders of the maximum *print depth* and *print length*. The return value of this function is not used (the type is *int* for historical reasons).

visit

A pointer to a "visit" function called by the garbage collector when tracing the set of all currently accessible objects. This function is only required if other Scheme objects are reachable from objects of the newly defined type (a null pointer can be given otherwise). It is invoked with two arguments: a pointer to the object being visited by the garbage collector, and a pointer to another function to be called once with the address of each object accessible through the original object. For example, the implementation of pairs would supply a visit function that invokes its second argument twice—once with the address of the car of the original object, and once with the address of the cdr.

The return value of *Define_Type()* is a small, unique integer identifying the type; it is usually stored in a "T_*" (or "t_*") variable following the convention used for the built-in types.

In the current version of Elk, *Define_Type()* cannot be used to define new "pointer-less" types resembling built-in types such as *fixnum* or *boolean*.

The first component of the C structure implementing a user-defined Scheme type must be an *Object*; its space is used by the garbage collector to store a special tag indicating that the object has been forwarded. If you are defining a type that has several components one of which is an *Object*, just move the *Object* to the front of the *struct* declaration. Otherwise insert an additional *Object* component.

The Scheme primitive that instantiates a new type can request heap space for the new object by calling the function *Alloc_Object()*:

```
Object Alloc_Object(int size, int type, int const_flag);
```

The arguments to *Alloc_Object()* are the size of the object in bytes (usually obtained by applying *sizeof* to the underlying *struct*), the type of which the new object is a member (i.e. the return value of *Define_Type()*), and a flag indicating whether the newly created object is to be made read-only. The return value is a fully initialized *Object*.

11.1. Example for a User-Defined Scheme Type

Figure 5 shows the skeleton of an extension that provides a simple Scheme interface to the UNIX *ndbm* library; it can be loaded dynamically into the Scheme interpreter, or into an Elk-based application that needs access to a simple database from within the extension language. Please refer to your system's documentation if you are not familiar with *ndbm*. The extension defines a new, first-class Scheme type *dbm-file* corresponding to the *DBM* type defined by the C library. Again, note the naming convention to use lower-case for new identifiers (in contrast to the predefined ones).

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

int t_dbm;

struct s_dbm {
     Object unused;
     DBM *dbm;
     char alive;   /* 0: has been closed, else 1 */
};

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

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

int dbm_print(Object d, Object port, int raw, int length, int depth) {
     Printf(port, "#[dbm-file %lu]", DBMF(d)->dbm);
     return 0;
}

Object p_is_dbm(Object d) {
     return TYPE(d) == t_dbm ? True : False;
}

void elk_init_dbm(void) {
     t_dbm = Define_Type(0, "dbm-file", 0, sizeof(struct s_dbm),
          dbm_equal, dbm_equal, dbm_print, 0);

     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);
}
```

Figure 5: Skeleton of a UNIX ndbm extension

The code shown in Figure 5 declares a variable *t_dbm* to hold the return value of *Define_Primitive()*, and the C structure *s_dbm* that represents the new type. The structure is composed of the required initial *Object*, the *DBM* pointer returned by the C library function *dbm_open()*, and a flag indicating whether the database pointed to by this object has already been closed (in this case the flag is cleared). As a *dbm-file* Scheme object can still be passed to primitives after the *DBM* handle has been closed by a call to *dbm_close()*, the *alive* flag had to be added to avoid further use of a ''stale'' object: the ''dbm'' primitives include an initial check for the flag and raise an error if it is zero.

The macro *DBMF* is used to cast the pointer field of an *Object* of type *t_dbm* to a pointer to the correct structure type. *dbm_equal()* implements both the *eqv?* and the *equal?* predicates; it returns true if the *Objects* compared point to an open database and contain identical *DBM* pointers. The print function just prints the numeric value of the *DBM* pointer; this could be improved by printing the name of the database file instead, which must then be included in each Scheme object. The primitive *p_is_dbm()* provides the usual type predicate. Finally, an extension initialization function is supplied to enable dynamic loading of the compiled code; it registers the new type and three primitives operating on it. Note that a visit function (the final argument to *Define_Type()*) is not required here, as the new type does not include any components of type *Object* that the garbage collector must know of—the required initial *Object* is not used here and therefore can be neglected. The type constructor primitive *dbm-open* and the primitive *dbm-close* are shown in Figure 6.

The primitive *dbm-open* shown in Figure 6 is called with the name of the database file, a symbol indicating the type of access (*reader* for read-only access, *writer* for read/write access, and *create* for creating a new file with read/write access), and an optional third argument specifying the file permissions for a newly-created database file. A default of 0666 is used for the file permissions if the primitive is invoked with just two arguments. Section 12.1 will introduce a set of functions that avoid clumsy if-cascades such as the one at the beginning of *p_dbm_open()*. *Primitive_Error()* is called with a ''format string'' and zero or more arguments and signals a Scheme error (see section 12.5). *dbm-open* returns #f if the database file could not be opened, so that the caller can deal with the error.

Note that *dbm-close* first checks the *alive* bit to raise an error if the database pointer is no longer valid because of an earlier call to *dbm-close*. This check needs to be performed by all primitives working on *dbm-file* objects; it may be useful to wrap it in a separate function—together with the initial type-check. Ideally, database objects should be closed automatically during garbage collection when they become inaccessible; section 12.4 will introduce functions to accomplish this.

At least two primitives *dbm-store* and *dbm-fetch* need to be added to the database extension to make it really useful; these are not shown here (their implementation is fairly simple and straightforward). Using these primitives, the extension discussed in this section can be used to write Scheme code such as this procedure (which looks up an electronic mailbox name in the mail alias database maintained on most UNIX systems):

```
Object p_dbm_open(int argc, Object *argv) {
     DBM *dp;
     int flags = O_RDWR|O_CREAT;
     Object d, sym = argv[1];

     Check_Type(sym, T_Symbol);
     if (EQ(sym, Intern("reader")))
          flags = O_RDONLY;
     else if (EQ(sym, Intern("writer")))
          flags = O_RDWR;
     else if (!EQ(sym, Intern("create")))
          Primitive_Error("invalid argument: ~s", sym);
     if ((dp = dbm_open(Get_String(argv[0]), flags,
               argc == 3 ? Get_Integer(argv[2]) : 0666)) == 0)
          return False;
     d = Alloc_Object(sizeof(struct s_dbm), t_dbm, 0);
     DBMF(d)->dbm = dp;
     DBMF(d)->alive = 1;
     return d;
}

Object p_dbm_close(Object d) {
     Check_Type(d, t_dbm);
     if (!DBMF(d)->alive)
          Primitive_Error("invalid dbm-file: ~s", d);
     DBMF(d)->alive = 0;
     dbm_close(DBMF(d)->dbm);
     return Void;
}
```

Figure 6: Implementation of *dbm-open* and *dbm-close*

```
(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"))
```

12. Advanced Topics

12.1. Converting between Symbols, Integers, and Bitmasks

Symbols are frequently used as the arguments to Scheme primitives which call an underlying C or C++ function with some kind of bitmask or with a predefined enumeration constant or preprocessor symbol. For example, the primitive *dbm-open* shown in Figure 6 above uses symbols to represent the symbolic constants passed to *dbm_open()*. Similarly, a Scheme primitive corresponding to the UNIX system call *open()* could receive a list of symbols represending the logical OR of the usual *open()* flags, so that one can write Scheme code such as:

```
(let ((tty-fd (unix-open "/dev/ttya"    '(read write exclusive)))
      (tmp-fd (unix-open "/tmp/somefile '(write create))))
     ...
```

To facilitate conversion of symbols to C integers or enumeration constants and vice versa, these two functions are provided:

```
unsigned long Symbols_To_Bits(Object syms, int mask_flag,
    SYMDESCR *table);
Object Bits_To_Symbols(unsigned long bits, int mask_flag,
    SYMDESCR *table);
```

The type *SYMDESCR* is defined as:

```
typedef struct {
     char *name;
     unsigned long val;
} SYMDESCR;
```

Symbols_To_Bits() converts a symbol or a list of symbols to an integer; *Bits_To_Symbols()* is the reverse operation and is usually applied to the return value of a C/C++ function to convert it to a Scheme representation. Both functions receive as the third argument a table specifying the correspondence between symbols and C constants; each table entry is a pair consisting of the *name* of a symbol as a C string and an integer *val* (typically an enumeration constant or a *#define* constant). Each *SYMDESCR* array is terminated by an entry with a zero *name* component:

```
SYMDESCR lseek_syms[] = {
     { "set",      SEEK_SET },
     { "current",  SEEK_CUR },
     { "end",      SEEK_END },
     { 0, 0 }
};
```

The second argument to the conversion functions controls whether a single symbol is converted to an integer or vice versa (*mask_flag* is zero), or whether a list of symbols is converted to the logical OR of a set of matching values or vice versa (*mask_flag* is non-zero). *Symbols_To_Bits()* signals an error if the symbol does not match any of the names in the given table or, if *mask_flag* is non-zero, if any of the list elements does not match. The empty list is converted to zero. If *Bits_To_Symbols()* is called with a non-zero *mask_flag*, it matches the *val* components against the *bits* argument using logical AND. Regardless of *mask_flag*, *Bits_To_Symbols* returns the empty list if no match occurs. Figure 7 shows an improved version of *p_dbm_open()* using *Symbols_To_Bits()* in place of nested if-statements.

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

Object p_dbm_open(int argc, Object *argv) {
     DBM *dp;
     Object d;

     dp = dbm_open(Get_String(argv[0]),
         Symbols_To_Bits(argv[1], 0, flag_syms),
         argc == 3 ? Get_Integer(argv[2]) : 0666);
     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;
}
```

Figure 7: Improved version of *dbm-open* using *Symbols_To_Bits()*

A Scheme primitive calling the UNIX system call *access()* could use *Symbols_To_Bits()* with a non-zero *mask_flag* to construct a bitmask:

```
Object p_access(Object fn, Object mode) {
     access(Get_String(fn), (int)Symbols_To_Bits(mode, 1, access_syms));
     ...
```

where *access_syms* is defined as:

```
static SYMDESCR access_syms[] = {
     { "read",       R_OK },
     { "write",      W_OK },
     { "execute",    X_OK },
     { 0, 0 }
};
```

Note that in this example the empty list can be passed as the *mode* argument to test for existence of the file, because in this case *Symbols_To_Bits()* returns zero (the value of *F_OK*).

12.2. Calling Scheme Procedures, Evaluating Scheme Code

A Scheme procedure can be called from within C or C++ code using the function

```
Object Funcall(Object fun, Object argl, int eval_flag);
```

The first argument is the Scheme procedure—either a primitive procedure (*T_Primitive*) or a compound procedure (*T_Compound*). The second argument is the list of arguments to be passed to the procedure, as a Scheme list. The third argument, if non-zero, specifies that the arguments need to be evaluated before calling the Scheme procedure. This is usually not the case (except in some special forms). The return value of *Funcall()* is the result of the Scheme procedure.

Funcall() is frequently used from within C callback functions that can be registered for certain events, such as the user-supplied X11 error handlers, X11 event handlers, timeout handlers, the C++ *new* handler, etc. Here, use of *Funcall()* allows to register a user-defined Scheme procedure for this event from within a Scheme program. As an example, Figure 8 shows the generic signal handler that is associated with various UNIX signals by the UNIX extension.

```
void scheme_signal_handler(int sig) {
     Object fun, args;

     Set_Error_Tag("signal-handler");
     Reset_IO(1);
     args = Bits_To_Symbols((unsigned long)sig, 0, signal_syms);
     args = Cons(args, Null);
     fun = VECTOR(handlers)->data[sig];
     if (TYPE(fun) != T_Compound)
          Fatal_Error("no handler for signal %d", sig);
     (void)Funcall(fun, args, 0);
     Printf(Curr_Output_Port, "\n\7Signal!\n");
     (void)P_Reset();
     /*NOTREACHED*/
}
```

Figure 8: Using *Funcall()* to call a Scheme procedure

The signal handler shown in Figure 8 uses the signal number supplied by the system to index a vector of user-defined Scheme procedures (that is, *Objects* of type *T_Compound*). *Reset_IO()* is used here to ensure that the current input and output port are in defined state when the Scheme signal handler starts executing. The argument list is constructed by calling *Cons()*; it consists of a single element—the signal number as a Scheme symbol. *signal_syms* is an array of *SYMDESCR* records that maps the UNIX signal names (*sighup*, *sigint*, etc.) to corresponding Scheme symbols of the same names. The Scheme procedure called from the signal handler is not supposed to return (it usually invokes a continuation); therefore the result of *Funcall()* is ignored. In case the Scheme handler (and thus the call to *Funcall()*) does return, a message is printed and the primitive *reset* is called to return to the application's toplevel or standard Scheme toplevel.

An S-expression can be evaluated by calling the function

```
Object Eval(Object expr);
```

which is identical to the primitive *eval* (*P_Eval()* in C), except that no optional environment can be supplied. *Eval()* is very rarely used by extensions or applications, mainly by implementations of new special forms. Both *Eval()* and *Funcall()* can trigger a garbage collection; all local variables holding Scheme *Objects* with heap pointers must be properly registered with the garbage collector to survive calls to these functions.

Occasionally an S-expression needs to be evaluated that exists as a C string, for example, when a Scheme expression has been entered through a "text widget" in a graphical user interface. Here, evaluation requires calling the Scheme reader to parse the expression; therefore a straightforward solution is to create a string port holding the string and then just "load" the contents of the port:

```
void eval_string(char *expr) {
     Object port; GC_Node;

     port = P_Open_Input_String(Make_String(expr, strlen(expr)));
     GC_Link(port);
     Load_Source_Port(port);
     GC_Unlink;
     (void)P_Close_Input_Port(port);
}
```

If a more sophisticated function is required, the *eval-string* extension included in the Elk distribution can be used ("lib/misc/elk-eval.c"). This extension provides a function

```
char *Elk_Eval(char *expr);
```

that converts the result of evaluating the stringized expression back to a C string and returns it as a result. A null pointer is returned if an error occurs during evaluation.

Applications should not use this function as the primary interface to the extension language. In contrast to languages such as Tcl, the semantic concepts and data structures of Scheme are not centered around strings, and strings are not a practicable representation for S-expressions. Instead, applications should pass control to the extension language by calling Scheme procedures (using *Funcall()*) or by loading files containing Scheme code. The extension language then calls back into the application's C/C++ layer by invoking application-supplied Scheme primitives and other forms of callbacks as explained in section 6.3.

12.3. GC-Protecting Global Objects

Section 9.1 explained when—and how—to register with the garbage collector function-local *Object* variables holding heap pointers. Similarly, global variables must usually be added to the set of reachable objects as well if they are to survive garbage collections (a useful exception to this rule will be introduced in section 12.4). In contrast to local variables, global variables are only made known to the garbage collector once—after initialization—as their lifetime is that of the entire program. To add a global variable to the garbage collector's root set, the macro

```
Global_GC_Link(obj)
```

must be called with the properly initialized variable of type *Object*. The macro takes the address of the specified object. If that is a problem, an equivalent functional interface can be used:

```
void Func_Global_GC_Link(Object *obj_ptr);
```

This function must be supplied the address of the global variable to be registered with the garbage collector.

When writing extensions that maintain global *Object* variables, *Global_GC_Link()* (or *Func_Global_GC_Link()*) is usually called from within the extension initialization function right after each variable is assigned a value. For instance, the global Scheme vector *handlers* that was used in Figure 8 to associate procedures with UNIX signals is initialized and GC-protected as follows:

```
void elk_init_unix_signal(void) {
     handlers = Make_Vector(NSIG, False);
     Global_GC_Link(handlers);
     ...
}
```

NSIG is the number of UNIX signal types as defined by the system include file. The signal handling Scheme procedures that are inserted into the vector later need not be registered with the garbage collector, because they are now reachable through another object which itself is reachable.

12.3.1. Dynamic C Data Structures

Dynamic data structures, such as the nodes of a linked list containing Scheme *Objects*, cannot be easily registered with the garbage collector. The simplest solution is to build these data structures in Scheme rather than in C or C++ in the first place. For example, a linked list of Scheme objects can be built from Scheme pairs much more naturally and more straightforward than from C structures or the like, in particular if the list will be traversed and manipulated using Scheme primitives anyway. Besides, data structures programmed in Scheme benefit from automatic memory management, whereas use of *malloc()* and *free()* in C frequently is a source of memory leaks and related errors.

If for some reason a dynamic data structure must be built in C or C++ rather than in Scheme, reachability problems can be avoided by inserting all *Objects* into a global, GC-protected vector (such as *handlers* in Figure 8) and then use the corresponding vector indexes rather than the actual *Objects*. This sounds more difficult than it really is; Appendix B shows the complete source code of a small module to register *Objects* in a Scheme vector. The module exports three functions: *register_object()* inserts an *Object* into the vector and returns the index as an *int*; *deregister_object()* removes an *Object* with a given index from the vector; and *get_object()* returns the *Object* stored under a given index. *register_object()* dynamically grows the vector to avoid artificial limits.

A dynamic data structure (e. g. linked list) implementation using this module would call *register_object()* when inserting a new *Object* into the list and then use the integer return value in place of the *Object* itself. Similarly, it would call *deregister_object()* whenever a node is removed from the list. *get_object()* would be used to retrieve the *Object* associated with a given list element. Note that with these functions the same *Object* can be registered multiple times (each time under a new index) without having to maintain reference counts: the garbage collector does not care how often a particular *Object* is traversed during garbage collection, as long as it will be reached at least once.

12.4. Weak Pointers and Object Termination

A data structure implementation may deliberately use *Objects* that are not added to the global set of reachable pointers (as described in the previous section) and are thus invisible to the garbage collector. In this case, it becomes possible to determine whether or not garbage collection has found any *other* pointers to the same Scheme objects. This property can be exploited in several ways by extensions or applications using Elk.

Pointers that are not included in the garbage collector's reachability search are called ''weak pointers''. The memory occupied by a Scheme object that is only referenced by weak pointers will be reclaimed. The term *weak* expresses the notion that the pointer is not strong enough to prevent the object it points to from being garbage collected. Code using weak pointers can scan the pointers immediately after each garbage collection and check whether the target object has been visited by the just-finished garbage collection. If this is the case, normal (strong) pointers to the object must exist (which can therefore be considered ''live''), and the weak pointer is updated manually to point to the object's new location. On the other hand, if the object has not been visited, no more (normal) references to it exist and the memory occupied by it has been reclaimed.

Weak pointers are useful in implementing certain types of data structures where the sole existence of a (weak) pointer to an object from within this data structure should not keep the object alive (*weak sets*, *populations*, certain kinds of hash tables, etc.). Objects that are not reachable through strong pointers are then removed from the weak data structure after garbage collection. In this case, it is frequently useful to invoke a ''termination function'' for each such object, e. g. for objects that contain resources of which only a finite amount is available, such as UNIX file descriptors (or FILE structures), X displays and windows, etc. The termination function for Scheme ports closes the file pointer encapsulated in a port object if it is still open; likewise, the termination function for X windows closes the window and thereby removes it from the display, and so on. Thus, should an object holding some kind of resource go inaccessible before it was terminated ''properly'' by calling the respective Scheme primitive (*close-input-port*, *close-output-port*, *destroy-window*, etc.), then resource will be reclaimed after the next garbage collection run.

12.4.1. Using Weak Pointers

Code using weak pointers must scan the pointers immediately after each garbage collection, but *before* the interpreter resumes normal operation, because the memory referenced by the weak pointers can be reused the next time heap space is requested. This can be accomplished by registering a so-called ''after-GC function. Elk's garbage collector invokes all after-GC functions (without arguments) upon completion. To register an after-GC functions, the function

```
void Register_After_GC((void (*func)(void)));
```

is used, typically in an extension initializer. Similarly, extensions and applications can register "before-GC functions" using

```
void Register_Before_GC((void (*func)(void)));
```

These functions are called immediately before each garbage collection and may be used, for instance, to change the application's cursor to an hourglass symbol. After-GC and before-GC functions must not trigger another garbage collection.

An after-GC function scanning a set of weak pointers makes use of the three macros *IS_ALIVE()*, *WAS_FORWARDED()*, and *UPDATE_OBJ()*. For example, an after-GC function scanning a table of elements holding *Objects* with weak pointers could be written as shown in Figure 9.

```
void scan_weak_table(void) {
    int i;

    for (i = 0; i < table_size; i++) {
        Object obj = table[i].obj;
        if (IS_ALIVE(obj)) {            /* object is still reachable */
            if (WAS_FORWARDED(obj))
                UPDATE_OBJ(obj);
        } else {
            terminate_object(obj);  /* object is dead; finalize... */
            table[i] = 0;           /* and remove it from the table */
        }
    }
}
```

Figure 9: After-GC function that scans a table containing weak pointers

The function *scan_weak_table()* shown in Figure 9 can then be registered as an after-GC function by invoking

```
Register_After_GC(scan_weak_table);
```

The then-part of the if-statement in *scan_weak_table()* is entered if the just-completed garbage collection has encountered any pointers to the Scheme object pointed to by *obj*; in this case the pointer conveyed in *obj* is updated manually using *UPDATE_OBJ()* (when using the generational garbage collector included in Elk, reachability of an object does not necessarily imply that it was forwarded, hence the additional call to *WAS_FORWARDED()*). If *IS_ALIVE()* returns false, no more strong pointers to the object exist and it can be terminated and removed from the weak data structure. *terminate_object()* typically would release any external resources contained in the Scheme object, but it must neither create any new objects nor attempt to "revive" the dead object in any way (e. g. create a new strong pointer to it by inserting it into another, live object).

12.4.2. Functions for Automatic Object Termination

As automatic termination of Scheme objects using user-supplied termination functions is the most frequent use of weak pointers, Elk offers a set of convenience functions for this purpose. Extensions and applications can insert *Objects* into a weak list maintained by Elk and remove them from the list using the two functions

```
void Register_Object(Object obj, char *group,
                     (Object (*term)(Object)), int leader_flag);
void Deregister_Object(Object obj);
```

term is the termination function that is called automatically with *obj* when the object becomes unreachable (its result is not used); *group* is an opaque "cookie" associated with *obj* and can be used to explicitly terminate all objects with the same value for *group*; a non-zero *leader_flag* indicates that *obj* is the "leader" of the specified *group*. Elk automatically registers an after-GC function to scan the weak list maintained by these two functions and to call the *term* function for all objects that could be proven unreachable by the garbage collector, similar to the function shown in Figure 9.

Object termination takes place in two phases: first all objects registered with a zero *leader_flag* are terminated, after that the termination functions of the leaders are invoked. This group and leader notion is used, for example, by the Xlib extension to associate windows (and other resources) with an X display: the ID of the display to which a window belongs is used as the window's group, and the display is marked as the group leader. Thus, if a display becomes unreachable or is closed by the program, all its windows are closed before the display is finally destroyed[5].

Two additional functions are provided for explicitly calling the termination functions:

```
void Terminate_Type(int type);
void Terminate_Group(char *group);
```

Terminate_Type() invokes the termination function (if any) for all objects of a given type and deletes them from the weak list. For example, to close all ports currently held open by Elk (and thus apply *fclose()* to the FILE pointers embedded in them), one would call

```
Terminate_Type(T_Port)
```

Terminate_Group() calls the termination functions of all non-leader objects belonging to the specified *group*.

Finally, another function, *Find_Object()*, locates an object in the weak list:

---

[5] This interface has evolved in a slightly *ad hoc* way; the two-stage relationship expressed by groups and group leaders may not be sufficient for more complex hierarchies than those used in X.

```
Object Find_Object(int type, char *group,
            (int (*match_func)(Object, ...)), ...);
```

Arguments are a Scheme type, a group, and a match function called once for each object in the weak list that has the specified type and group. The match function is passed the *Object* and the remaining arguments to *Find_Object()*, if any. If the match function returns true for an object, this object becomes the return value of *Find_Object()*; otherwise it returns *Null*.

Complicated as it may seem, *Find_Object()* is quite useful—extensions can check whether a Scheme object with certain properties has already been registered with the weak list earlier and, if this is the case, return *this* object instead of creating a new one. This is critical for Scheme objects encapsulating some kind of external resource, such as file descriptors or X windows. Consider, for example, a Scheme primitive that obtains the topmost window on a given X display and returns it as a Scheme *window* object. If the primitive just were to instantiate a Scheme object encapsulating the corresponding X window ID for each call, it would become possible for two or more distinct Scheme *window* objects to reference the same real X window. This is not acceptable, because two Scheme objects pointing to the same X object should certainly be equal in the sense of *eq?*, not to mention the problems that would ensue if one of the Scheme *window* objects were closed (thereby destroying the underlying X window) and the second one were still be operated on afterwards. Example uses of *Find_Object()* can be found in the Xlib extension and in the Xt extension that are included in the Elk distribution.

12.5. Errors

User-supplied code can signal an error by calling *Primitive_Error()* with a format string and as many additional arguments (*Objects*) as there are format specifiers in the format string:

```
void Primitive_Error(char *fmt, ...);
```

Primitive_Error() calls the default or user-defined error handler as described in the Elk Reference Manual, passing it an "error tag" identifying the source of the error, the format string, and the remaining arguments. A special format specifier "~E" can be used to interpolate the standard error message text corresponding to the UNIX error number *errno*; this is useful for primitives that invoke UNIX system calls or certain C library functions (if "~e" is used, the first character of the text is converted to lower case). If this format specifier is used, the current *errno* must be assigned to a variable *Saved_Errno* prior to calling *Primitive_Error()* to prevent it from being overwritten by the next system call or C library function. *Primitive_Error()* does not return.

Applications that need to supply their own error handler by redefining *error-handler* usually do so in Scheme, typically at the beginning of the initial Scheme file loaded in *main()*.

If *Primitive_Error()* is called from within a C function that implements a Scheme primitive, an error tag is supplied by Elk (the name of the primitive). Applications may set the error tag explicitly at the beginning of sections of C/C++ code that reside outside of primitives, for example, before loading an initial Scheme file in the application's *main()*. Two functions are provided to set and query the current error tag:

```
void Set_Error_Tag(const char *tag);
char *Get_Error_Tag(void);
```

The following three functions can be used by primitives to signal errors with standardized messages in certain situations:

```
void Range_Error(Object offending_obj);
void Wrong_Type(Object offending_obj, int expected_type);
void Wrong_Type_Combination(Object offending_obj, char *expected_type);
```

Range_Error() can be used when an argument to a primitive is out of range (typically some kind of index). *Wrong_Type()* signals a failed type-check for the given *Object*; the second argument is the expected type of the *Object*. This function is used, for example, by *Check_Type()*. *Wrong_Type_Combination()* is similar to *Wrong_Type()*; the expected type is specified as a string. This is useful if an *Object* can be a member of one out of two or more types, e. g. a string or a symbol.

Fatal errors can be signaled using the functions

```
void Fatal_Error(char *fmt, ...);
void Panic(char *msg);
```

Fatal_Error() passes its arguments to *printf()* and then terminates the program. *Panic()* is used in situations that ''cannot happen'' (failed consistency checks or failed assertions); it prints the specified message and terminates the program with a core dump.

12.6. Exceptions

As explained in the Elk Reference Manual, a user-supplied Scheme procedure is called each time an *exception* is raised. Currently, the set of UNIX signals that are caught by the interpreter or an extension (at least *interrupt* and *alarm*) are used as exceptions. As signals occur asynchronously, extensions and applications must be able to protect non-reentrant or otherwise critical code sections from the delivery of signals. In particular, calls to external library functions are frequently not reentrant[6] and need to be protected from being disrupted.

Extensions may call the macros *Disable_Interrupts* and *Enable_Interrupts* (without arguments) to enclose code fragments that must be protected from exceptions. Calls to these macros can be nested, and they are also available as Scheme primitives on the Scheme-language level. As all modern UNIX versions provide a facility to temporarily block the delivery of signals, a signal that occurs after a call to *Disable_Interrupts* will be delayed until the outermost matching *Enable_Interrupts* is executed. Two additional macros, *Force_Disable_Interrupts* and *Force_Enable_Interrupts* can be used to enable and disable signal delivery regarless of the current nesting level. Extensions that use additional signals (such as the *alarm* signal) must register these with the interpreter core to make sure they are included in the *mask* of signals that is maintained by *Disable_Interrupts* and *Enable_Interrupts* (the interface for registering signals is still being revised; refer to the source code of the UNIX extension for an example).

---

[6] Fortunately, with the advent of multithreading, vendors are now beginning to provide reentrant versions of their system libraries.

The ability to protect code from exceptions is particularly useful for primitives that temporarily open a file or allocate some other kind of resource that must subsequently be released again. If the relevant code fragment were not enclosed by calls to *Disable_Interrupts* and *Enable_Interrupts*, an exception handler could abandon execution of the code section by calling a continuation, thus causing the file to remain open forever. While situations like this can be handled by *dynamic-wind* on the Scheme level, some form of *try/catch* facility is not available on the C-language level, and using the C function implementing the *dynamic-wind* primitive would be cumbersome.

The function

```
void Signal_Exit(int signal_number);
```

may be used as the handler for signals that must terminate the application; it ensures that the temporary files maintained by Elk are removed and calls the extension finalization functions in the normal way.

12.7. Defining Scheme Variables

User-supplied C/C++ code can define global Scheme variables that are maintained as corresponding *Object* C variables. The Scheme interpreter itself defines several such variables, for example, the variable *load-path* (see section 5) which can be modified and read both from Scheme and from C. The function *Define_Variable()* is used to define a Scheme variable and bind an initial value to it:

```
void Define_Variable(Object *var, const char *name, Object init);
```

var is the address of the C variable corresponding to the newly-created Scheme variable, *name* is the name of the Scheme variable, and *init* is its initial value. *Define_Variable()* calls *Intern()* to create the variable name included in the new binding and *Func_Global_GC_Link()* to properly register the C variable with the garbage collector.

The C side of a Scheme variable cannot be accessed directly; the functions

```
Var_Set(Object variable, Object value);
Var_Get(Object variable)
Var_Is_True(Object variable)
```

must be used instead to assign a value to the variable and to read its current value; the first argument to each function is the *Object* whose address was passed to *Define_Variable()*. *Var_Is_True()* is convenient for boolean variables and tests whether the contents of the variable is true in the sense of *Truep()*. As an example, Figure 10 shows how the Xt extension defines a Scheme variable that is associated with the user-defined ‘‘warning handler’’ called by the Xt library to output warning messages.

In the example in Figure 10, the function *Xt_Warning()* is registered as the Xt ‘‘warning handler’’ by passing it to *XtSetWarningHandler()*. It is invoked by Xt with a warning message. The message is converted to a Scheme string, and, if the Scheme variable *xt-warning-handler* has been assigned a procedure, this procedure is called with the string using *Funcall()*. Otherwise the string is just sent to the current output port. The call to *Define_Variable()* in the extension

```
Object V_Xt_Warning_Handler;

void Xt_Warning(char *msg) {
     Object args, fun;

     args = Cons(Make_String(msg, strlen(msg)), Null);
     fun = Var_Get(V_Xt_Warning_Handler);
     if (TYPE(fun) == T_Compound)
          (void)Funcall(fun, args, 0);
     else
          Printf(Curr_Output_Port, "%s\n", msg);
}

void elk_init_xt_error(void) {
     Define_Variable(&V_Xt_Warning_Handler, "xt-warning-handler", Null);
     XtSetWarningHandler(Xt_Warning);
}
```

Figure 10: The Xt extension defines a Scheme variable holding a "warning handler"

initialization function associates the Scheme variable *xt-warning-handler* with the C variable *V_Xt_Warning_Handler* (as a convention, Elk uses the prefix "V_" for variables of this kind).

12.8. Defining Readers

In addition or as an alternative to the constructor primitive for a new Scheme type, applications and extensions may define a *reader* function for each new type. The bitstring extension, for example, defines a reader to allow input of bitstring literals using the *#*10110001* syntax. Each user-defined read syntax is introduced by the '#' symbol followed by one more character, identifying the type of the object. To define a reader, the following function is called (typically from within an extension initialization function):

```
void Define_Reader(int c,
     (Object (*func)(Object port, int c, int const_flag)));
```

The arguments to *Define_Reader()* are the as yet unused character identifying the type (e. g. '*' for bitstrings) and a pointer to a *reader function* that is invoked by the Scheme parser whenever the newly defined syntax is encountered. This reader function is passed a Scheme input port from which it reads the next token, the character following the '#' symbol (to facilitate using the same reader for different types), and a flag indicating whether the newly-created object is expected to be made read-only (this is true when expressions are loaded from a file). The reader function must return a new object of the given type.

You may want to refer to the bitstring extension included in the Elk distribution for an example definition of a reader function ("lib/misc/bitstring.c"), and for the macros that can be used by reader functions to efficiently read characters from a port.

12.9. Fork Handlers

Extensions may need to be notified when a copy of the running interpreter (or application) is created by means of the *fork()* UNIX system call. For example, consider an extension that stores information in a temporary file and removes this file on termination of the program. If another extension created a copy of the running interpreter by calling *fork()*, the child process would remove the temporary file on exit—the file would not be available to the original instance of the interpreter (i. e. the parent process) any longer. To prevent premature removal of the file, the extension that owns it can define a *fork handler* by calling *Register_Onfork()* with a pointer to a C function:

```
void Register_Onfork((void (*func)(void)));
```

The function could create an additional link to the file, so that a child process would just remove this link on exit, leaving the original link intact.

Extensions that use *fork()* without executing a new program in the child process (e. g. the UNIX extension which defines a *unix-fork* primitive) are required to call the function *Call_Onfork()* in the newly created child process to invoke all currently defined fork handlers:

```
void Call_Onfork(void);
```

Appendix A: Functions that can Trigger a Garbage Collection

This appendix lists the functions exported by Elk that may trigger a garbage collection. Within C/C++ code, local Scheme objects must be protected as shown in section 9.1 when one of these functions is called during the objects' lifetime.

The C functions corresponding to the following Scheme primitives can cause a garbage collection:

`append`	`load`	`read-string`
`apply`	`macro-body`	`require`
`autoload`	`macro-expand`	`reverse`
`backtrace-list`	`make-list`	`string`
`call-with-input-file`	`make-string`	`string->list`
`call-with-output-file`	`make-vector`	`string->number`
`call/cc`	`map`	`string->symbol`
`command-line-args`	`oblist`	`string-append`
`cons`	`open-input-file`	`string-copy`
`dump`	`open-input-output-file`	`substring`
`dynamic-wind`	`open-input-string`	`symbol-plist`
`eval`	`open-output-file`	`tilde-expand`
`for-each`	`open-output-string`	`type`
`force`	`port-line-number`	`vector`
`get-output-string`	`procedure-lambda`	`vector->list`
`list`	`provide`	`vector-copy`
`list->string`	`put`	`with-input-from-file`
`list->vector`	`read`	`with-output-to-file`

all special forms
all mathematical primitives except predicates
all output primitives if output is sent to a string port

In practice, most of these functions, in particular the special forms, are rarely or never used in extensions or Elk-based applications. In addition to these primitives, the following C functions can trigger a garbage collection:

`Alloc_Object()`	`Make_Reduced_Flonum()`	`Make_String()`
`Make_Port()`	`Make_Flonum()`	`Make_Const_String()`
`Load_Source_Port()`	`Define_Primitive()`	`Intern()`
`Load_File()`	`Printf()`	`CI_Intern()`
`Copy_List()`	`Print_Object()`	`Define_Variable()`
`Const_Cons()`	`General_Print_Object()`	`Define_Symbol()`
`Make_Integer()`	`Format()`	`Bits_To_Symbols()`
`Make_Unsigned()`	`Eval()`	`Make_Vector()`
`Make_Long()`	`Funcall()`	`Make_Const_Vector()`
`Make_Unsigned_Long()`		

Note: *Make_Integer()*, *Make_Unsigned()*, *Make_Long()*, and *Make_Unsigned_Long()* can only trigger a garbage collection if *FIXNUM_FITS()* (or *UFIXNUM_FITS()*, respectively) returns zero for the given argument.

Appendix B: Convenience Functions for GC-Safe Data Structures

Figure 11 shows the source code for a set of functions to insert Scheme objects into a vector that has been registered with the garbage collector, to delete objects from the vector, and to retrieve the object stored under a given vector index. These functions help building dynamic data structures (such as linked lists or hash tables) containing Scheme objects. There is nothing application-specific in the code; if you find it useful, you can directly include it in your Elk extension or Elk-based application without any changes. See section 12.3 for a detailed description.

```
static int max_objects = 32;      /* initial size */
static int num_objects;
static Object objects;
static int inx;

int register_object(Object x) {
     Object v;
     int n;
     GC_Node;

     if (num_objects == max_objects) {
          max_objects *= 2;
          GC_Link(x);
          v = Make_Vector(max_objects, Null);
          GC_Unlink;
          memcpy(VECTOR(v)->data, VECTOR(objects)->data,
               num_objects * sizeof(Object));
          objects = v;
          inx = num_objects;
     }
     for (n = 0; !Nullp(VECTOR(objects)->data[inx]);
               inx++, inx %= max_objects) {
          n++;
          assert(n < max_objects);
     }
     VECTOR(objects)->data[inx] = x;
     num_objects++;
     return inx;
}

void deregister_object(int i) {
     VECTOR(objects)->data[i] = Null;
     --num_objects;
     assert(num_objects >= 0);
}

Object get_object(int i) {
     return VECTOR(objects)->data[i];
}

void elk_init_gcroot(void) {
     objects = Make_Vector(max_objects, Null);
     Global_GC_Link(objects);
}
```

Figure 11: Functions to map Scheme objects to indexes into a GC-safe vector

Appendix C: Summary of Functions, Macros, Types, and Variables

This appendix provides a quick overview of the functions and other definitions exported by the Elk kernel. The list is divided in groups of definitions with related functionality; the entries are presented in roughly the same order in which they are introduced in the above chapters. Full function prototypes are given for functions; in some prototypes, arguments are given names for clarification. The initial keywords **function**, **macro**, **typedef**, and **variable** indicate the type of each entry (function, preprocessor symbol with or without arguments, type definition, and external variable defined by Elk, respectively). The functions corresponding to Scheme primitives (as described in section 7.4) have been omitted from the list.

Accessing the Scheme Object Representation

typedef `Object`

macro `TYPE(obj)`
macro `POINTER(obj)`
macro `ISCONST(obj)`
macro `SETCONST(obj)`
macro `SET(obj, type, ptr)`
macro `EQ(obj1, obj2)`

Defining Scheme Primitives

function `void Define_Primitive((Object (*func)()), const char *name,`
`int minargs, int maxargs, enum discipline disc);`

Making Objects Known to the Garbage Collector

macro `GC_Node, GC_Node2, ...`
macro `GC_Link(obj), GC_Link2(obj1, obj2), ...`
macro `GC_Unlink`
macro `Global_GC_Link(obj)`
function `void Func_Global_GC_Link(obj_ptr);`

Booleans

macro `T_Boolean`
macro `Truep(obj)`

variable `Object True`
variable `Object False`

function `int Eqv(Object, Object);`
function `int Equal(Object, Object);`

Characters

macro `T_Character`
macro `CHAR(char_obj)`
function `Object Make_Char(int);`
variable `Object Newline`

Pairs and Lists

macro `T_Null`
macro `Nullp(obj)`
variable `Null`

macro `T_Pair`
macro `PAIR(pair_obj)`
macro `Car(obj)`
macro `Cdr(obj)`
macro `Cons(obj1, obj2)`

macro `Check_List(obj)`
function `int Fast_Length(Object);`
function `Object Copy_List(Object);`

Integers (Fixnums and Bignums)

macro `T_Fixnum`
macro `T_Bignum`
macro `FIXNUM_FITS(integer)`
macro `UFIXNUM_FITS(unsigned_integer)`
macro `FIXNUM(fixnum_obj)`
macro `BIGNUM(bignum_obj)`

macro `Check_Integer(obj)`
macro `Check_Number(obj)`

function `Object Make_Integer(int);`
function `Object Make_Unsigned(unsigned);`
function `Object Make_Long(long);`
function `Object Make_Unsigned_Long(unsigned long);`

function `int Get_Integer(Object);`
function `unsigned Get_Unsigned(Object);`
function `long Get_Long(Object);`
function `unsigned long Get_Unsigned_Long(Object);`

function `int Get_Exact_Integer(Object);`
function `unsigned Get_Exact_Unsigned(Object);`
function `long Get_Exact_Long(Object);`
function `unsigned long Get_Exact_Unsigned_Long(Object);`

Floating Point Numbers (Reals)

macro `T_Flonum`
macro `FLONUM(flonum_obj)`
function `Object Make_Flonum(double);`
function `Object Make_Reduced_Flonum(double);`
function `double Get_Double(Object);`

Symbols

macro `T_Symbol`
macro `SYMBOL(symbol_obj)`
function `Object Intern(const char *);`
function `Object CI_Intern(const char *);`
function `void Define_Symbol(Object *var, const char *name);`
variable `Object Void`

typedef `SYMDESCR`
function `unsigned long Symbols_To_Bits(Object syms, int mask_flag,`
`SYMDESCR *table);`
function `Object Bits_To_Symbols(unsigned long bits, int mask_flag,`
`SYMDESCR *table);`

Strings

macro `T_String`
macro `STRING(string_obj)`
function `Object Make_String(const char *init, int size);`
function `char *Get_String(Object);`
function `char *Get_Strsym(Object);`
macro `Get_String_Stack(obj, char_ptr)`
macro `Get_Strsym_Stack(obj, char_ptr)`

Vectors

macro `T_Vector`
macro `VECTOR(vector_obj)`
function `Object Make_Vector(int size, Object fill);`

Ports

macro `T_Port`
macro `PORT(port_obj)`
function `Object Make_Port(int flags, FILE *f, Object name);`
function `Object Terminate_File(Object port);`
macro `Check_Input_Port(obj)`
macro `Check_Output_Port(obj)`

variable `Object Curr_Input_Port, Curr_Output_Port`
variable `Object Standard_Input_Port, Standard_Output_Port`
function `void Reset_IO(int destructive_flag);`
function `void Printf(Object port, char *fmt, ...);`
function `void Print_Object(Object obj, Object port, int raw_flag, int print_depth, int print_length);`
macro `Print(obj)`
function `void Load_Source_Port(Object port);`
function `void Load_File(char *filename);`

Miscellaneous Types

macro `T_End_Of_File`
variable `Object Eof`

macro `T_Environment`
variable `Object The_Environment, Global_Environment`

macro `T_Primitive`
macro `T_Compound`
function `void Check_Procedure(Object);`

macro `T_Control_Point`
macro `T_Promise`
macro `T_Macro`

Defining Scheme Types and Allocating Objects

function
```
int Define_Type(int zero, const char *name,
            int (*size)(Object), int const_size,
            int (*eqv)(Object, Object),
            int (*equal)(Object, Object),
            int (*print)(Object, Object, int, int, int),
            int (*visit)(Object*, int (*)(Object*)));
```
function `Object Alloc_Object(int size, int type, int const_flag);`

Calling Scheme Procedures and Evaluating Scheme Code

function `Object Funcall(Object fun, Object argl, int eval_flag);`
function `Object Eval(Object expr);`
function `char *String_Eval(char *expr);`

Weak Pointers and Object Termination

function `void Register_Before_GC((void (*func)(void)));`
function `void Register_After_GC((void (*func)(void)));`

macro `IS_ALIVE(obj)`

macro `WAS_FORWARDED(obj)`
macro `UPDATE_OBJ(obj)`

function `void Register_Object(Object obj, char *group,`
`(Object (*term)(Object)), int leader_flag);`
function `void Deregister_Object(Object obj);`
function `void Terminate_Type(int type);`
function `void Terminate_Group(char *group);`
function `Object Find_Object(int type, char *group,`
`(int (*match_func)(Object, ...)), ...);`

Signaling Errors

function `void Primitive_Error(char *fmt, ...);`
function `void Set_Error_Tag(const char *tag);`
function `char *Get_Error_Tag(void);`
function `void Set_App_Name(char *name);`
function `void Range_Error(Object offending_obj);`
function `void Wrong_Type(Object offending_obj, int expected_type);`
function `void Wrong_Type_Combination(Object offending_obj,`
`char *expected_type);`
function `void Fatal_Error(char *fmt, ...);`
function `void Panic(char *msg);`
variable `int Saved_Errno`

Exceptions (Signals)

macro `Disable_Interrupts, Enable_Interrupts`
macro `Force_Disable_Interrupts, Force_Enable_Interrupts`
function `void Signal_Exit(int signal_number);`

Defining and Using Scheme Variables

function `void Define_Variable(Object *var, const char *name, Object init);`
function `void Var_Set(Object var, Object val);`
function `Object Var_Get(Object var);`
function `int Var_Is_True(Object var);`

Defining Reader Functions

function `void Define_Reader(int c,`
`(Object (*func)(Object port, int c, int const_flag)));`

Fork Handlers

function `void Register_Onfork((void (*func)(void)));`
function `void Call_Onfork(void);`

Allocating Memory

function `char *Safe_Malloc(unsigned size);`
function `char *Safe_Realloc(char *old_pointer, unsigned size);`

macro `Alloca_Begin, Alloca_End`
macro `Alloca(char_ptr, type, size)`

Initializing Elk from an Application's main()

function `void Elk_Init(int argc, char **argv, int init_flag,`
`char *filename);`

Miscellaneous Macros

macro `ELK_MAJOR, ELK_MINOR`
macro `NO_PROTOTYPES, WANT_PROTOTYPES`

Index

X

_

Table of Contents