libscheme: Scheme as a C Library

Brent W. Benson Jr.
Harris Computer Systems
`Brent.Benson@mail.csd.harris.com`

Abstract

Because of its small size and simplicity, Scheme is often seen as an ideal extension or scripting language. While there are many Scheme implementations available, their interfaces are often complex and can get in the way of using the implementation as part of a larger software product. The `libscheme` library makes the Scheme language available as a C library. Its interface is through a single C header file and it is easily extended with new primitive procedures, new primitive types, and new syntax. It is portable to any system that has an ANSI C compiler and to which Hans Boehm's popular conservative garbage collector [1] has been ported. It has been used to build a variety of special purpose data manipulation tools, and as an extension language for an ethernet monitor.

1 Introduction

There is a long tradition of scripting languages in the Unix community, the canonical example being `/bin/sh` [2]. Scripting languages allow the programmer to express programming ideas at a high level, and can also be designed in such a way that the language interpreter can be included as an extension language inside of other programs. When a program provides a powerful extension language to end users, it often increases the utility of the program by orders of magnitude (consider GNU Emacs and GNU Emacs Lisp as an example). While in recent years there has been an explosion of general purpose extension and scripting languages (e.g., Python [6] and Elk [4]), one language has had a dramatic increase in popularity and seems to have become the de facto extension language. That language is Tcl [5].

It is the author's opinion that the popularity of Tcl is primarily due to the ease with which it can be embedded into C applications (its interface is through a single C header file and a standard C library archive file) and the ease with which it can be extended with new primitive commands. The `libscheme` library attempts to learn from Tcl's success by making Scheme [3] available as a C library and by providing simple ways to extend the language with new procedures and syntax. While Scheme is not as convenient as Tcl in the role of an interactive shell program, it has several advantages over Tcl with respect to writing scripts:

1. Lexical Scope
2. Nested procedures
3. A richer set of data types
4. Extensible syntax

In addition, `libscheme` allows the application writer to extend the interpreter with new data types that have the same standing as built in types. It also provides a conservative garbage collector that can be used by application and extension writers.

2 Scheme

Scheme is a small, lexically scoped dialect of Lisp that is based on the principle that a programming language should not include everything but the kitchen sink, but rather it should provide a framework in which it is easy to build the kitchen sink.

> Programming languages should be designed not by piling feature on top of feature, but by removing the weaknesses and restrictions that make additional features appear necessary. Scheme demonstrates that a very small number of rules for forming expressions, with no restrictions on how they are composed, suffice to form a practical and efficient programming language that is flexible enough to support most of the major programming paradigms in use today. [3]

These properties make Scheme a good general purpose programming language and also an ideal extension language. Its conceptual simplicity allows it to be implemented with a relatively small number of lines of code, while providing powerful high level programming constructs such as procedures that can be nested and used as first class values, and high-level data structures like lists, vectors and strings. It also removes the burden of memory management from the programmer, usually through some sort of garbage collector.

Scheme supports all major programming paradigms in use today including functional, procedural, and object oriented. It scales well from small applications to large software systems.

2.1 An Example Procedure

Let us examine a small Scheme procedure to get a feel for the language. The procedure in figure 1 splits a string of characters into a list of strings based on a delimiter character.

```
(define (split-string string delimiter)
  (let ((len (string-length string)))

    ; Collect characters until we reach a delimiter character,
    ; at which time we extract a field and start looking for
    ; the next delimiter.
    ;
    (define (collect start end)
      (cond
       ((= end len)
        (list (substring string start end)))
       ((char=? (string-ref string end) delimiter)
        (cons (substring string start end)
              (collect (+ end 1) (+ end 1))))
       (else (collect start (+ end 1)))))

    ; Start at the beginning of the string.
    ;
    (collect 0 0)))
```

Figure 1: A sample Scheme function

In this example we have a top-level definition of the `split-string` function and an *internal definition* of the `collect` function. An internal, or *nested* function like `collect` has access to all lexical variables in the scope of its definition. More specifically, `collect` has access to the lexical variables `string`, `delimiter` and `len`. The `let` special form establishes a series of local variables and initial bindings. In `split-string` the `let` establishes only the binding of `len`. The `cond` special form evaluates the test in each of its clauses, performing the clauses action(s) when it finds the first test to succeed. The `else` clause of a `cond` form is executed if no other clause succeeds. Scheme also has other standard control constructs like `if` and `case`.

```
> (split-string "brent:WgG6SfAUnX51Q:5359:100:Brent Benson" #\:)
("brent" "WgG6SfAUnX51Q" "5359" "100" "Brent Benson")
>
```

2.2 Closures

Procedures are first class in Scheme, meaning that procedures can be assigned to variables, passed as arguments to other procedures, be returned from procedures, and be applied to arbitrary arguments. When they are created, procedures "capture" variables that are free in their bodies.[1] Because of this "closing over," procedures are also known as *closures* in Scheme.

Closures can be used for a wide variety of purposes. They can be used to represent objects in an object-oriented framework, to model actors, to represent partially computed solutions, etc.

Unnamed procedures are created with the `lambda` special form. The existence of unnamed procedures frees the programmer from making up names for simple helper functions. For example, many Scheme implementations have a `sort` procedure that accepts a list or vector of elements, and a comparison procedure that is used to establish an ordering on the elements. It is often useful to use an unnamed procedure as the comparison function:

```
> (sort '(1 6 3 4) (lambda (n1 n2) (< n1 n2)))
(1 3 4 6)
> (sort #("jim" "brent" "jason" "todd")
        (lambda (s1 s2) (string<? s1 s2)))
#("brent" "jason" "jim" "todd")
```

Note that `(1 2 3 4)` is a list of numbers and `#("jim" ...)` is a vector of strings.

The next example shows the definition of a procedure `make-counter` that returns another procedure created with `lambda` that closes over the `count` variable.

```
> (define (make-counter)
    (let ((count 0))
      (lambda ()
        (set! count (+ count 1))
        count)))
make-counter
> (define c1 (make-counter))
c1
> (c1)
1
> (c1)
2
```

Since no one else has access to `count` it becomes private to the closure and can be modified and used within its context. In this particular case, `count` is incremented and its value returned when the procedure returned from `make-counter` is called.

2.3 Syntax

As you have probably already noticed, Scheme's syntax is Lisp-like. All function applications are in fully-parenthesized prefix form. While some find this sort of syntax unwieldy, it has the

[1] A variable is free in a procedure if it is not contained in the parameter list.

advantage that Scheme forms are actually lists which can be easily manipulated with standard list primitives. The `libscheme` library supports the `defmacro` special form that can be used by end users to create new special forms. A special form is a form that is evaluated by special rules. For example, the `if` special form only evaluates its "then" condition if its test expression evaluates to true, otherwise it evalutes its "else" expression.

These macros are much more powerful than the simple token-based substitution macros provided by languages like C.

3 An Application that Uses `libdwarf`

The `libscheme` library makes Scheme available as a C library. Its interface is through a single C header file. Scheme expressions can be read from an arbitrary input stream, evaluated with respect to a particular environment, and printed to an arbitrary output stream. The user can extend the interpreter by adding bindings to the global environment. Each binding can provide a new primitive written in C, a new syntax form, a new type, a constant, etc.

3.1 An Example

DWARF is a full-featured and complex debugging information format [7]. Our example program, `dwarfscheme`, is an interface that allows the user to browse DWARF information in an object file by providing stubs to the `libdwarf` [8] library. Figure 2 shows a sample `dwarfscheme` dialogue.

In this example the user invokes `dwarfscheme`, opens the file `"a.out"` for DWARF reading, defines a function for printing out debugging information entries (DIEs), and prints out the first DIE. This example shows how `dwarfscheme` would be used by an end user. Next, we will examine the way that the `dwarfscheme` executable was created using `libscheme` and the `libdwarf` libraries.

The program `dwarfscheme` is an executable that was produced by linking `libscheme` with a set of DWARF manipulating primitives, a read-eval-print loop that initializes the primitives, and the `libdwarf` library that is provided as a system library. The main routine for `dwarfscheme` appears in figure 3.

This main routine is a boiler-plate routine that is used when the application writer wants to make the application a Scheme read-eval-print loop. The thing that differentiates the main routines in different applications is the initializations that are done on the environment. In this case, we create a basic environment containing the standard `libscheme` bindings, and then add the DWARF specific bindings to it by calling `init_dwarf()`. The rest of the routine takes care of the business of establishing an error handler, printing out a prompt, reading an expression, evaluating the expression, and printing out the result.

The application writer can also embed `libscheme` in an application that is not structured as a read-eval-print loop. For example, at program startup a windowing application might initialize a `libscheme` environment, read and evaluate Scheme expressions representing configuration information from a user configuration file, and then enter its event loop. The user might bring up a dialog box in which she can evaluate Scheme expressions to further configure and query the system's state.

The major part of the DWARF initialization routine, `init_dwarf()` appears in figure 4. It consists of calls to `scheme_make_type()` to establish new data types, and then several calls to `scheme_add_global()` to add new global bindings to the environment provided as an argument. Each call to `scheme_add_global()` provides the Scheme name for the global, the initial value for the variable (in this case a new primitive that points to its C implementation), and an environment to which the global should be added. All routines and variables that are part of the `dwarfscheme` interface begin with a `dw` prefix, while routines and variables from the system-supplied `libdwarf` library begin with a `dwarf` prefix.

This practice of calling an initialization routine with the environment for each logical piece of code is only a convention, but is a helpful way of organizing `libscheme` code. The `libscheme` library itself is organized this way. Each file contains an initialization function that establishes that file's primitives.

```
$ ./dwarfscheme
> (define dbg (dwarf-init "a.out"))
dbg
> (define die1 (dwarf-first-die dbg))
die1
> (define (dwarf-print-die die)
    (display (dwarf-tag-string (dwarf-tag die)))
    (newline)
    (for-each
      (lambda (attr)
        (display "  ")
        (display (dwarf-attr-name-string (dwarf-attr-name attr)))
        (display " ")
        (write (dwarf-attr-value attr))
        (newline))
      (dwarf-attributes die)))
dwarf-print-die
> (dwarf-print-die die1)
DW_TAG_compile_unit
  DW_AT_comp_dir "CX/UX:/jade2/ccg/brent/misc/package/dwarf"
  DW_AT_identifier_case 0
  DW_AT_low_pc 198304
  DW_AT_high_pc 198344
  DW_AT_language 2
  DW_AT_name "t.c"
  DW_AT_producer "Harris C Compiler - Version build_c_7.1p1_03"
  DW_AT_stmt_list 0
#t
> (exit)
$
```

Figure 2: `dwarfscheme` dialogue

```
#include <stdio.h>
#include "scheme.h"
#include "dwarfscheme.h"

main()
{
  Scheme_Env *env;
  Scheme_Object *obj;

  env = scheme_basic_env ();
  init_dwarf (env);
  scheme_default_handler ();
  do
    {
      printf ("> ");
      obj = scheme_read (stdin);
      if (obj == scheme_eof)
        {
          printf ("\n; done\n");
          exit (0);
        }
      obj = scheme_eval (obj, env);
      scheme_write (stdout, obj);
      printf ("\n");
    }
  while ( 1 );
}
```

Figure 3: `dwarfscheme` read-eval-print loop

```
static Scheme_Object *dw_debug_type;
static Scheme_Object *dw_die_type;
static Scheme_Object *dw_first_die (int argc, Scheme_Object *argv[]);
...
void
init_dw (Scheme_Env *env)
{
  dw_debug_type = scheme_make_type ("<debug>");
  dw_die_type = scheme_make_type ("<die>");
  dw_attribute_type = scheme_make_type ("<attribute>");
  scheme_add_global ("dwarf-init",
                     scheme_make_prim (dw_init), env);
  scheme_add_global ("dwarf-first-die",
                     scheme_make_prim (dw_first_die), env);
  scheme_add_global ("dwarf-next-die",
                     scheme_make_prim (dw_next_die), env);
  scheme_add_global ("dwarf-tag",
                     scheme_make_prim (dw_tag), env);
  ...
}
```

Figure 4: The DWARF primitive initialization routine

A sample primitive is shown in figure 5. Each `libscheme` primitive accepts an argument count and a vector of evaluated arguments. Each primitive procedure is responsible for checking the number and type of its arguments. All Scheme objects are represented by the C type `Scheme_Object` (see section 4.1). The types `Dwarf_Debug` and `Dwarf_Die` are foreign to `libscheme` and are provided by the `libdwarf` library.

```
static Scheme_Object *
dw_first_die (int argc, Scheme_Object *argv[])
{
  Dwarf_Debug dbg;
  Dwarf_Die die;

  SCHEME_ASSERT ((argc == 1),
                 "dwarf-first-die: wrong number of args");
  SCHEME_ASSERT (DWARF_DEBUGP (argv[0]),
                 "dwarf-first-die: arg must be a debug object");
  dbg = (Dwarf_Debug) SCHEME_PTR_VAL (argv[0]);
  die = dwarf_nextdie (dbg, NULL, NULL);
  if (! die)
    {
      return (scheme_false);
    }
  else
    {
      return (dw_make_die (die));
    }
}
```

Figure 5: A `dwarfscheme` primitive

The `SCHEME_ASSERT()` macro asserts that a particular form evaluates to true, and signals an error otherwise. The `dw_first_die()` routine first checks for the correct number of arguments, then it checks that the first argument is an object with type `dw_debug_type`. Next, it extracts the pointer value representing the DWARF information in the file from the first argument, a `Scheme_Object`. It then calls a `libdwarf` function, `dwarf_nextdie()` and returns an appropriate value—a new `dw_die_type` object if there is another DIE, the Scheme false value otherwise. The `dw_make_die()` routine accepts a `Dwarf_Die` as an argument and returns a `libscheme` object of type `dw_die_type` that contains a pointer to the `Dwarf_Die` structure.

Now that we have a feel for the way that `libscheme` is extended, we will take a closer look at the design of `libscheme` itself.

4 `libscheme` Architecture

This section describes some specifics of `libscheme`'s implementation. An important feature of its design is that beyond a small kernel of routines for memory management, error handling, and evaluation, all of its Scheme primitives are implemented in the same way as non-`libscheme` extensions. This is similar to Tcl's implementation strategy.

4.1 Object Representation

Every object in `libscheme` is an instance of the C structure `Scheme_Object`. Each instance of `Scheme_Object` contains a pointer to a type object (that also happens to be a `Scheme_Object`) and two data words. If an object requires more than two words of storage or if the object is some other type of foreign C structure, it is stored in a separate memory location and pointed to by the `ptr_val` field. The actual definition of `Scheme_Object` appears in figure 6.

```
struct Scheme_Object
{
  struct Scheme_Object *type;
  union
    {
      char char_val;
      int int_val;
      double double_val;
      char *string_val;
      void *ptr_val;
      struct Scheme_Object *(*prim_val)
        (int argc, struct Scheme_Object *argv[]);
      struct Scheme_Object *(*syntax_val)
        (struct Scheme_Object *form, struct Scheme_Env *env);
      struct { struct Scheme_Object *car, *cdr; } pair_val;
      struct { int size; struct Scheme_Object **els; }
               vector_val;
      struct { struct Scheme_Env *env;
               struct Scheme_Object *code; } closure_val;
    } u;
};
```

Figure 6: The definition of `Scheme_Object`

While many Scheme implementations choose to represent certain special objects as immediate values (e.g., small integers, characters, the empty list, etc.) this approach was not used in `libscheme` because the "everything is an object with a tag approach" is simpler and easier to debug. A side effect of this decision is that `libscheme` code that does heavy small integer arithmetic will allocate garbage that must be collected, in contrast to higher performance Scheme implementations that only dynamically allocate very large integers.

4.2 Primitive Functions

Primitive functions in Scheme are implemented as C functions that take two arguments, an argument count and a vector of `Scheme_Objects`. Each primitive is responsible for checking for the correct number of arguments—allowing maximum flexibility for procedures of variable arity—and for checking the types of its arguments. All arguments to a primitive function are evaluated before they are passed to the primitive, following Scheme semantics. If the application writer wants to create a primitive that doesn't evaluate its arguments, she must use a syntax primitive. C functions are turned into `libscheme` primitives with the `scheme_make_prim()` function that accepts the C function as an argument and returns a new Scheme object of type `scheme_prim_type`.

4.3 Primitive Syntax

The user can add new primitive syntax and special forms to `libscheme`. A `libscheme` syntax form is implemented as a C function that takes two arguments, an expression and an environment in which the expression should be evaluated. The form is passed directly to the syntax form with no evaluation performed. This allows the syntax primitive itself to evaluate parts of the form as needed. Figure 7 shows the implementation of the `if` special form. Note that `if` cannot be implemented as a procedure because it must not evaluate all of its arguments. The `scheme_eval()` function evaluates a `libscheme` expression with respect to a particular environment.

```
static Scheme_Object *
if_syntax (Scheme_Object *form, Scheme_Env *env)
{
  int len;
  Scheme_Object *test, *thenp, *elsep;

  len = scheme_list_length (form);
  SCHEME_ASSERT (((len == 3) || (len == 4)),
                 "badly formed if statement");
  test = SCHEME_SECOND (form));
  test = scheme_eval (test, env);
  if (test != scheme_false)
    {
      thenp = SCHEME_THIRD (form);
      return (scheme_eval (thenp, env));
    }
  else
    {
      if (len == 4)
        {
          elsep = SCHEME_FOURTH (form);
          return (scheme_eval (elsep, env));
        }
      else
        {
          return (scheme_false);
        }
    }
}
```

Figure 7: The `if` special form

C functions that represent syntax forms are turned into Scheme objects by passing them to the `scheme_make_syntax()` procedure which returns a new object of type `scheme_syntax_type`.

4.4 Type Extensions

Scheme as defined in its standard has the following data types: boolean, list, symbol, number, character, string, vector, and procedure. While Scheme in its current form does not allow the creation of user-defined types, the `libscheme` system allows users to extend the type system with new types by calling the `scheme_make_type()` function with a string representing the name of the new type. This function returns a type object that can be used as a type tag in subsequently created objects. Normally, types are created in a file's initialization function and objects of the new type are created using a user-defined constructor function that allocates and initializes instances of the type.

In figure 8 we see the constructor for the `dw_debug_type` type from our `dwarfscheme` example. It accepts an object of type `Dwarf_Debug`, a pointer to a C structure defined in the `libdwarf` library, allocates a new `Scheme_Object`, sets the object type, and stores the pointer to the foreign structure into the `ptr_val` slot of the object.

```
static Scheme_Object *
dw_make_debug (Dwarf_Debug dbg)
{
  Scheme_Object *debug;

  debug = scheme_alloc_object ();
  SCHEME_TYPE (debug) = dw_debug_type;
  SCHEME_PTR_VAL (debug) = dbg;
  return (debug);
}
```

Figure 8: An object constructor

It is often convenient to define a macro that checks whether a `libscheme` object is of a specified type. The macro defined in `dwarfscheme` for the DWARF debug object looks like this:

```
#define DW_DEBUGP(obj) (SCHEME_TYPE(obj) == dwarf_debug_type)
```

The 'P' at the end of `DW_DEBUGP` indicates that the macro is a predicate that returns a true or false value. All of the builtin types have type predicate macros of this form (e.g., `SCHEME_PAIRP`, `SCHEME_VECTORP`, etc.).

4.5 Environment Representation

The state of the interpreter is contained in an object of type `Scheme_Env`. The environment contains both global and local bindings. The definition of the `Scheme_Env` structure is shown in figure 9. The global variable bindings are held in a hash table. The local bindings are represented by a vector of variables (symbols) and a vector of corresponding values. An environment that holds local variables points to the enclosing environment with its `next` field. Therefore, variable value lookup consists of walking the environment chain, looking for a local variable of the correct name. If no local binding is found, the variable is looked for in the global hash table.

```
struct Scheme_Env
{
  int num_bindings;
  struct Scheme_Object **symbols;
  struct Scheme_Object **values;
  Scheme_Hash_Table *globals;
  struct Scheme_Env *next;
};
```

Figure 9: The `Scheme_Env` structure

Table 1 lists the environment manipulation functions. Unless the user is adding special forms that create variable bindings, she usually only needs to worry about the `scheme_basic_env()` and `scheme_add_global()` functions. The `scheme_basic_env()` function is used to create a new environment with the standard Scheme bindings which can then be extended with new primitives, types, etc. using `scheme_add_global()`.

4.6 Interpreter Interface

The `libscheme` functions that are used for reading, evaluating and writing expressions are listed in table 2.

`scheme_basic_env ()`	Return a new `libscheme` env
`scheme_add_global (name, val, env)`	Add a new global binding
`scheme_add_frame (syms, vals, env)`	Add a frame of local bindings
`scheme_pop_frame (env)`	Pop a local frame
`scheme_lookup_value (sym, env)`	Lookup the value of a variable
`scheme_lookup_global (sym, env)`	Lookup the value of a global
`scheme_set_value (sym, val, env)`	Set the value of a variable

Table 1: Environment manipulation functions

`scheme_read (fp)`	Read an expression from stream
`scheme_eval (obj, env)`	Evaluate an object in environment
`scheme_write (fp, obj)`	Write object in machine readable form
`scheme_display (fp, obj)`	Write object in human readable form

Table 2: Interpreter functions

These functions can be used in the context of a read-eval-print loop or called at arbitrary times during program execution.

4.7 Error Handling

The `libscheme` library provides rudimentary error handling support. Errors are signaled using the `scheme_signal_error()` function, or by failing the assertion in a `SCHEME_ASSERT()` expression. If the default error handler is installed by calling `scheme_default_handler()`, then all uncaught errors will print the error message and `abort()`. Errors can be caught by evaluating an expression within a `SCHEME_CATCH_ERROR()` form. This macro evaluates its first argument. If an error occurs during the execution, the value second argument is returned, otherwise, the value of the first expression is returned.

```
obj = scheme_read (stdin);
result = SCHEME_CATCH_ERROR (scheme_eval (obj, env), 0);
if (result == 0)
   {
     /* error handling code */
   }
else
   {
     scheme_write (stdout, result);
   }
```

4.8 Memory Allocation/Garbage Collection

The `libscheme` library uses Hans Boehm and Alan Demers' conservative garbage collector [1]. It provides a replacement for the C library function `malloc()` called `GC_malloc()`. The storage that is allocated by `GC_malloc()` is subject to garbage collection. The collector uses a conservative approach, meaning that when it scans for accessible objects it treats anything that could possibly point into the garbage collected heap as an accessible pointer. Therefore, it is possible that an integer or a floating point number that looks like a pointer into this area could be treated as a root and the storage that it points to would not be collected. In practice, this rarely happens.

Users of `libscheme` can use the garbage collector in their own program and are strongly encouraged to make use of it when extending `libscheme`. Normally the user can simply call `scheme_alloc_object()` to allocate a new `Scheme_Object`, but occasionally other types of objects need to be allocated dynamically.

The Boehm/Demers collector is freely available and can run on most any 32-bit Unix machine.

5 Pros, Cons and Future Work

The `libscheme` library is simple to understand and use. It builds on the powerful semantic base of the Scheme programming language. The library also provides several powerful extensions to Scheme including an extensible type system and user defined structure types.

The `libscheme` interpreter is not very fast. The primary reason is an inefficient function calling sequence that dynamically allocates storage, creating unnecessary garbage. This issue is being addressed and future versions should be a great deal more efficient. In any case, `libscheme` is intended primarily for interactive and scripting use for which its performance is already adequate.

When compared to a language like Tcl, Scheme is not as well suited for interactive command processing. A possible solution is to add a syntax veneer on top of the parenthetical Scheme syntax for interactive use. On the other hand, Scheme's clean and powerful semantics provide a sizeable advantage over Tcl for writing large pieces of software. It also has the advantage of real data types rather than Tcl's lowest common denominator "everything is a string" approach.

The `libscheme` library has already been used in many small projects. The author plans to make `libscheme` even more useful by providing a variety of useful bindings including interfaces to the POSIX system calls, a socket library, a regular expression package, etc.

6 Conclusion

The `libscheme` library makes Scheme available as a standard C library and is easily extended with new primitives, types, and syntax. It runs on most Unix workstations including Harris Nighthawks, Suns, Intel 386/486 under Linux and OS/2, HP9000s, DECstations and DEC Alphas running OSF/1, IBM RS/6000s, NeXT machines and many others. Its simplicity and extensibility lends itself to use as an extension or scripting language for large systems. Currently it is being used by the DNPAP team at Delf University of Technology in the Netherlands as part of their ethernet monitor, and is being evaluated for use in a variety of other projects. The latest version of `libscheme` can be obtained from the Scheme Repository, `ftp.cs.indiana.edu`, in the directory `/pub/imp/`.

References

[1] Hans Boehm and M. Weiser. *Garbage Collection in an Uncooperative Environment*. Software Practice and Experience. pp. 807-820. September, 1988.

[2] Stephen Bourne. *An Introduction to the UNIX Shell*. Berkeley 4.3 UNIX User's Supplementary Documents. USENIX Association.

[3] William Clinger and Jonathan Rees (Editors). *Revised^4 Report on the Algorithmic Language Scheme*. Available by anonymous ftp from `altdorf.ai.mit.edu`. 1991.

[4] Oliver Laumann. *Reference Manual for the Elk Extension Language Kit*. Available by anonymous ftp from `tub.cs.tu-berlin.de`.

[5] John Ousterhout. *Tcl: an Embeddable Command Language*. Proceedings of the Winter 1990 USENIX Conference. USENIX Association. 1990.

[6] Guido van Rossum. *Python Reference Manual*. Release 1.0.2. Available by anonymous ftp from `ftp.cwi.nl`. 1994.

[7] *DWARF Debugging Information Format*. Unix International. Available by anonymous ftp from `dg-rtp.dg.com`. 1994.

[8] *DWARF Access Library (libdwarf)*. Unix International. 1994.

The Author

Brent Benson received a BA in Mathematics from the University of Rochester 1990 and completed the work for his MS in Computer Science at the University of New Hampshire in 1992. He has been a senior software engineer in the small but feisty compiler group at Harris Computer Systems since 1992.