Why Use Scheme?

Clinton Ebadi

January 2002

1 What is Scheme?

Scheme is a small dialect of LISP. The GNU version of Scheme is Guile[1]. The current stable release of Guile (1.4) is not fully compliant with the current standard, but the unstable versions of Guile (1.5 or 1.7) available from the Guile CVS repository are. Scheme has a small amount of syntax, and generally "gets out of your way." Unlike languages like C, you do not have to deal with complicated memory management; Scheme is garbage collected[2]. With Scheme (Guile in particular), you can even extend the core language with C (or the other way around). This article will discuss GNU Guile Scheme.

2 Getting Guile Scheme

Getting Guile Scheme is fairly easy. The easiest way to get it is to install packages from your distribution. To install Guile for Debian, simply
`apt-get install guile1.4`. This will get you Guile 1.4, which does not have as many features as Guile 1.5 or 1.7. You can download the Guile Source from http://www.gnu.org/software/guile. Compilation of Guile is just as easy as with any other Autoconf based program:

```
./configure
make
su -c 'make install'
```

You may want to pass `--with-threads` to configure to enable threading support. If you want to be bleeding edge, you can grab Guile 1.7 from CVS: (just hit enter when login asks for a password)

```
export CVSROOT=:pserver:anoncvs@\
subversions.gnu.org:/cvsroot/guile
cvs login
cvs co guile/guile-core
```

To compile you must first run `autogen.sh` from the source directory. Remember to add `--enable-maintainer-mode` to configure so it automatically updates your configure script and makefiles.

3 Learning Scheme

In order for the rest of the article to make sense, you need to actually know some Scheme. If you have never used Scheme before I recommend reading *Teach Yourself Scheme in Fixnum Days*[3] before you start the rest of the article. If you do not have to the time to do so, here is a quick overview of the language.

3.1 Data

Every in Scheme is a *Symbolic Expression* or simply a *S-expression*. The basic unit of a Scheme program is the list. A list is data contained between two

[1] http://www.gnu.org/software/Guile

[2] That is, the Scheme environment frees memory for you when the last reference to an object is destroyed

[3] http://www.cs.rice.edu/~dorai/t-y-Scheme

parenthesis. This is very powerful because the program is data and the data is the program. You can use the *eval* command to evaluate an S-expression. This is useful if you want to allow a user to enter data as a program. The basic built in data types are: *booleans*, *characters*, *strings*, *numbers*, *vectors*, and *lists*. Booleans have two possible values: `#t` and `#f`. Every value other than `#f` is true. Characters are written in the form `#\c` where "c" is an ASCII character. Strings are composed of characters, and are written like strings are in most other languages (between quotation marks). Numbers are just like numbers in any other language, except that they can represent any rational number exactly (e.g. 1/3). Scheme numbers can also be complex when written in the form a+bi. (e.g. 1+1i or 1-1i).Vectors contain other data elements. A vector is created using `make-vector` which takes an integer argument (the size of the vector) or `vector` which takes the elements of the vector as its arguments. Vectors cannot be resized after they have been created. Lists, however, can. A list is created using `make-list` or `list` (which have the same effect as the versions for vectors). Where accessing an element in a vector is a constant time operation, accessing an element is a linear time operation (where "n" is the number of the element in the list). To define a variable you use `define`. For example `(define foo 5)` will create a new variable named "foo" that is a number. To set the value of an existing variable you use `set!`.

3.2 Functions

Functions are created using the `lambda` keyword. A function is a normal variable, just like any other type. This makes Scheme extremely flexible. To create a function that takes a fixed number of arguments you write `(lambda (a b c) [body])` where you write the names of the variables (e.g. `(lambda (x y) [body])+)` takes two arguments). To take any number of arguments you simply write `(lambda x [body])` where "x" is any variable name. This will put the values of the variables that the function was called with into a list bound to the symbol "x." To create a function that has a certain number of required arguments and one or more optional arguments, use the notation `(lambda (x y . z) [body])`. This causes any arguments above the required two to be stored in the list "z." In Scheme, you always pass variables by value. There are no references. For vectors, however, you only copy the pointer to the vector so when you modify the vector you are modifying it in place. `Lambda` creates an anonymous function. You can call it immediately like this `((lambda (x y) (+ x y)) 5 6)`. That will call the function with the arguments "5" and "6." A function returns the value of its last expression, so this function would return 11. You can create a name function very easily: `(define foo (lambda (x y) (+ x y)))`. This will bind the function to the variable "foo." You can now call the function using the name foo: `(foo 5 6)`.

3.3 Symbols and Quoting

Symbols are variables too. A symbol is an identifier in Scheme (e.g. foo, list, vector, make-list). You can make a symbol not evaluate to whatever is bound to it (maybe nothing) by quoting it using `'` or `‘` (back quote). For example `'foo` evaluated to `foo` instead of the value that "foo" represents. The other type of quotation is called quasi-quotation. You use the back quote character for this. When you quasi quote something (e.g. a list) you can unquote parts of it using `,` and `,@`. For example `‘(1 2 ,@(3 4) ,foo foo)` will return this list (assuming foo equals 5): `(1 2 3 4 5 foo)`. Comma unquotes an elements in a quasi quoted list. Comma-at ("unquote and splice") unquotes a list and splices its elements into the current list.

3.4 Binding and Looping

To bind a variable to a value you can use `define`. To set the value of an existing variable you use

`set!`. To introduce a new scope and create new bindings for variables, you use `let`. `Let` is used like this: `(let ((var1 value) (var2 value) ... (varX value)) [body])`. This lets you temporarily create variables, use them, and have them destroyed when you exit the body of the `let`. Another use of `let` is "named let." "Named let" allows you to loop. Example:

```
(let loop ((i 0))
  (cond ((< i 5)
  (display i)
  (loop (+ i 1)))))
```

This will print `012345`.

3.5 Conditionals

The main conditional forms in Scheme are `if` and `cond`. `If` works like it does in C (e.g. `(if test exec-if-true exec-if-false)`). `Cond` takes the form

```
(cond
  (test execute-if-true)
  (test2 execute-if-true)
   ...
  (else execute-if-all-tests false))
```

3.6 Ready to Go

That wasn't much in the way of a tutorial, was it? It should provide you with just enough knowledge of Scheme to be able to read the rest of the article. I suggest that you read *Teach Yourself Scheme in Fixnum Days* if you want to actually use Scheme. I skipped a large portion of the language and didn't go into any kind of real depth for the rest. What I showed in this section was but a small glance into the world of Scheme. To properly to an overview of the essentials of the language I would have need at least 30 or so pages.

4 Should I Use Scheme?

Why use Scheme and not (say) C? A big reason is numbers. Another is the ability for variables to hold any type. There are also several reasons not to use Scheme. One is speed, Guile Scheme is a lot slower than C.

4.1 Yes

In Scheme, you can represent numbers like 1/3 exactly. All numbers are also complex. You don't have to deal with special data types like `_Complex` in C. This allows math to be done more simply. For example, you can write a simple program that allows the user to enter any number (complex or not) and a number to raise it to. For the source, see figure **??** for the source to this program.

Figure 1: Simple number program

```
(display  "Please enter the number")
(define abi (read))
(display "Please enter the power")
(define n (read))
(display  "Answer to numberñ")
(display (expt number n))
(newline)
```

You also don't have to deal with data types when declaring variables; all variables can hold *any* type. You can use type predicates (e.g. `vector?`) in your functions after you have written the first version of your program. This is a very powerful feature. You can flesh out a design without having to worry about types until you have to. This also frees you from having to do type-checking in functions that you know will not have incorrect types as arguments (usually because another function you wrote calls it). With Scheme variables you get the flexibility of a C `void*` with optional type safety and much easier usage. Functions are also normal variables. It would be very easy to create a list or vector of callback functions in Scheme; just add them into the data structure like you would other variables.

Memory management is one of Scheme's strong points. Scheme uses garbage collection, so you do not have to worry about it. Simply stop referring to an object (e.g. re-assigning a variable) and the value is marked as free and garbage collected. Languages like Java also do this. Garbage collection is a very powerful feature. The source of many bugs (in languages that you must manually manage memory in) is memory management. Why have every program deal with memory management manually when you can do it all in one place? Of course, many people will disagree, but whether garbage collection is the answer is usually something that is true for some apps but false for others.

Scheme's syntax rules are very simple. The basic rule is that lists are made up of atoms. Beyond that, there isn't much else. You can also extended Scheme's syntax using `let-syntax`. Being able to extend the syntax of Scheme is one of the more complicated features. `Let-syntax` is generally wrapped by `define-macro` is most Scheme systems to allow macros to be written more easily. Macros are much more powerful in Scheme than they are in C. One things that you can do with Scheme macros is generate new macros; something that cannot be done with the C macro system. In Guile, you can define a macro almost as easily as you would a function using `define-macro`. The big difference is that you have to return a form instead of a value. See figure **??** for an example macro. Figure **??** returns a new functions named `printname` where `name` is the name passed to the macro. It prints `name`. It is, as the name implies, a completely useless macro, but it does serve it's purpose of illustrating macros. What does this macro return? If called with `name` as `foo` the result will be what is show in figure **??**.

One problem you may face when using a language that doesn't have namespaces is a clash between named in two or more libraries that you use. The generally accepted method of naming functions in a library for C is to prefix the name of the library before the functions (e.g. `lt_dlopen` for GNU ltdl). This has one problem: you cannot rename the functions if two libraries clash. C++ has the `namespace` keyword that defines new namespaces. You cannot easily rename a namespace. To do so you must do something like this:

Figure 2: Simple (and useless) macro

```
(define-macro useless
  (lambda (name)
    (let ((sym
           (string-append "print" (symbol->string
name))))
      `(define ,(string->symbol sym)
         (lambda () (display ',name)
(newline))))))
```

Figure 3: Expansion of a Simple Macro

```
((lambda (name)
   (let ((sym
          (string-append "print" (symbol->string
foo))))
         (define printfoo
           (lambda () (display 'foo) (newline)))))
 foo)
```

```
namespace bar {
#include <foo.h> // contains namespace foo
using foo;
};
```

Clearly, this can get tedious. Although Standard Scheme does not not have a module system, Guile does. It is similar to the Common-LISP package system. To use a module you call `(use-modules (foo bar))`. This will search for foo/bar.x in your `%load-path`. `X` is an extension from the `-extensions` are variables in the Scheme environment can be modified by your `~/.guile`. The ability to load modules is a nice feature, but it still doesn't solve the name clashing issue. You can easily prefix the symbols from a module with anything you like so:

```
(use-modules ((foo bar)
              :rename (symbol-prefix-proc
'bar:)))
```

This will prefix everything in the `foo bar` module with `bar:`. This makes it easy to avoid name clashes. See the Guile reference manual for information on creating your own modules (it is actually quite easy).

Programs are data. This is a big advantage that Scheme (and LISP in general) has over other languages. Using `eval`, you can evaluate arbitrary data as a program. This allows the user to enter programs as data. A good example of this is the *listener* loop. The listener loop reads Scheme commands and evaluates them. This allows you to interactively enter programs. A very simple listener loop written in Scheme is shown in figure **??**. The `interaction-environment` is the current environment that the script is being executed in. What other use is there for `eval`? One good one would be to allow users to enter Scheme programs while they used your program. For example, you could have a text-mode audio editing application and control the operations on files using functions. You cannot do this in C using C. If you wanted something similar you can either invent your own language or use (say) libguile as an extension language (like glame[4] does).

Want to make your application easily extendable? With Scheme, it is easy. You could write a guile module for your program and have extensions load this. The extensions themselves would be put in a special extension dir to be loaded in a loop using `load` until `readdir` returns the `eof-object` (End Of File Object). This can be accomplished by factoring your applications innards into a library used by your program and the extensions and providing a function to hook into your program. The actual reading of the files is a very simple loop and can be seen in Figure **??**. To do the same in C would require dynamically loading libraries and would be somewhat more complicated and not portable (unless you used GNU ltdl). Of course, the way I show it (using `readdir`) would will only work with a Scheme system that supports POSIX (like guile).

Figure 4: Simple Listener Loop

```
(let listen ()
  (display "\nlistener> ")
  (display
   (eval (read) (interaction-environment)))
  (listen))
```

[4]http://www.glame.de

Figure 5: Extension Loading Loop

```
(let ((extdir (opendir
"/usr/local/share/myapp/ext")))
  (readdir extdir)
  (readdir extdir) ; ignore . and ..
  (let readfile ((file (readdir extdir)))
    (cond ((not (eof-object? file)) ; if !EOF,
continue reading dir
   (load file)
   (readfile (readdir extdir))))))
```

If you want to write a graphical application using Gtk+ or GNOME, then you can very easily use Guile. There are full bindings for Gtk+ and GNOME, although there are no bindings for the GtkExtra widgets. Wrapping new widgets is easy (in theory). All you have to do is write a def file for the widget. Getting Guile-GNOME and Guile-Gtk is a bit of a hassle with Guile 1.7, unless you download a few patches to the source[5]. Writing Gtk+ applications is generally easier in Scheme than in C. You don't have to use any of the many type-casting macros! If you don't like Gtk+, you can use the guile-tk TK bindings. At one point there were bindings for Mesa, but they no longer work.

4.2 No

If speed is a concern, then don't use Guile Scheme. Guile is currently interpreted. A byte-compiler has been partially written, but development appears to be stalled and it does not compile with the current unstable version of Guile. There is also a JIT[6] compiler, but that currently doesn't support much at all and it also appears to have made no progress in several months.

If you need low level access to memory, then Scheme probably won't work for you. You won't be writing the next kernel using Scheme[7]. If you want to do low level I/O, Guile does not provide very much for you.

[5]Available from http://lamer.hackedtobits.com/code/scheme.html

[6]Just In Time

[7]Unless that kernel is Vapour, http://vapour.sf.net

You can open file descriptors and convert them into Scheme ports, and then use Scheme `read`, `write`, and Guile `format` to manipulate them, but you do not have access to C `read`, `write`, or `ioctl` (to name a few). If you absolutely need those, you can use SWIG[8] to generate wrappers (except for `ioctl` because it is a variadic function). Most of the time if you are using low level I/O functions you want access to the memory layout of what you are reading, something you do not have access to in Scheme.

4.3 Working around Scheme's Limitations

One should not ask the question "Scheme or C?". Why choose? You can have the strengths of both languages with Guile. If you can't do something easily in Scheme, then you can use libguile to write an extension in C. If you would like to allow a user to write scripts for your C program, then you can use libguile to do the same thing. Writing Guile Scheme extensions may be the topic of a future article. For now, you should read the Guile Scheme reference manual for more information on writing extensions or extending your application with Scheme scripting. One goal of the Guile project is to create translators for other scripting languages into Scheme. This will allow the user to enter scripts in their favorite language and then use it to control your program. The HEAD branch of Guile (1.7) contains a translator for EMACS-LISP. It is currently the only translator. Extensions also provide a way to get the power of Scheme with the speed of C. You can write your most important functions in C and control them from Scheme. Of course, the only reason to do that would be for speed reasons (which could be mostly solved by a working byte-compiler).

[8] http://www.swig.org

5 Example

You could implement something similar to Qt's signal/slot system very easily in Scheme. Define three functions: `connect`, `emit`, and `remove-connections`. `Connect` connects a signal to a slot. A signal is represented by a symbol, and passed to connect just like any other variable. Connect then looks up the symbol in a hash table (a built in data type in Guile). If the symbol is not found, a new vector is added that contains the list of callbacks. Once the symbol has been found (or added to the table), the callback is be added to the `callbacks` list. `Emit` is be defined to take one or more arguments. The required argument is the symbol that will have all of its callbacks called. The optional arguments are the arguments that the callbacks are called with. If the symbol is not found in the table, an error will be printed. If the symbol is found, the list of optional arguments will be passed to the callback. To remove a connection call `remove-connections`. `Remove-connection` takes one argument: the signal to disconnect all of its callbacks from. There are two major flaws in this system: signals are created and stored globally, so callbacks have no way of telling what object called them. An easy way to work around this is to pass the object to the callback. The other is that you must disconnect all of the callbacks from a symbol, and cannot disconnect one signal at a time (if you were to use this in a production system, this would have to be fixed). Another smaller flaw is that the call to `emit` blocks the entire program while the callbacks execute. A better system would execute the callbacks in their own thread (but that introduces nasty thread safety issues). For the source to the system see figure **??**.

To help the code make more sense, I will explain a few of the procedures used here. `Apply` takes a procedure and a list as arguments. It them splices the list into its individual components and passes them as arguments to the procedure (e.g. `(apply foo '(1 2 3))` calls `foo` with `1 2 3` as its arguments). Hash Tables in Guile are a vector of pairs. Each pair contains a

key and the data. This is why I take the `cdr` of the value returns by `hash-get-handle` (this removes the key and gives me the callbacks). `For-each` takes at least two arguments, the first being a functions and the rest being lists. `For-each` then iterates through the list arguments and calls the function with them. I use it to `apply` the args list to each callback (using a wrapper function around `apply` that takes one argument, the callback to apply args to). The rest should make sense, if it doesn't consult the Guile reference manual.

What use is there for such a system? A threaded version that allowed you to disconnect individual callbacks from a signal and to associate signals with objects would be very useful for a GUI widget system (just like the one used in Qt). Even this limited version has some uses. Imagine you wrote a filter language in Scheme. It takes files as input, applies user selected transformations to its input, and then outputs the results. You could associate all of the user selected transformation with a signal, read the files, and then emit the signal for each file to apply the transformations.

Figure 6: Simple Signal and "Slot" system

```
;; simple signal and callback library
;; Copyright (C) 2002 Clinton Ebadi
;; Covered under the GPL Version 2 or (at your option) any later version

;; %*sig-table is the signal/callback table
;; it starts out at 50 entries, but grows automatically
(define %*sig-table (make-hash-table 50))
(define connect ; connects a signal to a callback (or "slot")
  (lambda (sig callback)
    (if (not (symbol? sig)) #f) ; return false if sig is not a signal
    (let ((found (hash-get-handle %*sig-table sig)))
      (if found
  (add-callback sig callback)
  (begin ; else
    (add-signal sig)
    (add-callback sig callback))))
    #t ))
(define add-callback ; adds a new callback to sig's callback list
  (lambda (sig callback)
    (hash-set! %*sig-table sig (append
(cdr (hash-get-handle %*sig-table sig))
(list callback)))))
(define add-signal ; adds a new signal to the signal table
  (lambda (sig)
    (hash-create-handle! %*sig-table sig '())))
(define remove-connections ; removes all of a signal's connections
  (lambda (sig)
    (if (not (symbol? sig)) #f)
    (if (hash-get-handle %*sig-table sig)
(hash-set! %*sig-table sig '())
#f ))) ; return false if the signal does not exist
(define emit ; emits a signal and calls all of it's callbacks
  (lambda (signal . args)
    (let ((callbacks (hash-get-handle %*sig-table signal)))
      (cond ((not callbacks)
     (display "Signal not found\n")
      #f ) ; display error and return false
    (else
      (set! callbacks (cdr callbacks)) ; get just callbacks
     (for-each
      (lambda (x) (apply x args))
      callbacks))))))
```

6 How You Can Help Guile

You can help Guile in a number of ways. The first is to go to the Project section of the Guile web site[9] and complete one of the listed project. You could also work on the (mostly complete for 1.7) reference manual. Several sections need to be rewritten and it can always use some proofreading. A big project would be to work on the guile byte compiler (available from the Guile CVS Repository as `guile/guile-vm`). The easiest way for you to help Guile is to tell others about it (like I have with this article). A very useful project would be to create bindings for popular C libraries. A few good libraries to write bindings for would be Xlib (for those times when you need raw X), Mesa (for 3D graphics), SDL (graphics), ALSA (audio), or KDE/Qt. KDE/Qt bindings could be generated from the C bindings, or using GOOPS[10] (which would require someone to rewrite parts of GOOPS in C).

7 Copyright Notice and Distribution Terms

[9] http://www.gnu.org/software/guile
[10] Guile Object Oriented Programming System