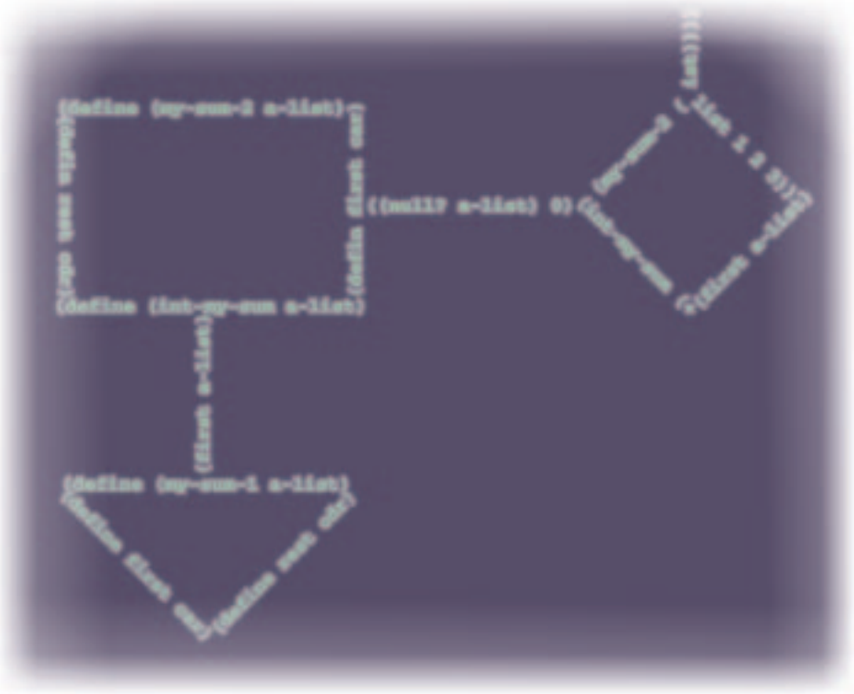
Shell Scripting with Scheme

A CHANGE ON THE BRIDGE

FRIEDRICH DOMINICUS



Thinking about scripting, the "usual suspects" like Bash, Korn Shell, Python, etc. will probably come to mind. However, scripting is also possible with Scheme. The focus of this article is therefore on shell programming, as well as "programming for the Internet".

The Scheme shell (Scsh; *http://www-swiss.ai.mit.edu/ftpdir/scsh/*) was designed specifically for Unix scripting; in its current form it is not nearly as suitable for interactive use as for instance the Bash or Z shell. There are other implementations in Lisp languages, die-hard Emacs fans could check out Eshell, a shell for Emacs in Emacs Lisp (*http://www.emacs.org/~johnw/eshell.tar.gz*).

Debian installation users have access to a DEB package of the Scheme shell, users of RPM-based distributions have to go via the following sources *ftp://ftp-swiss.ai.mit.edu/pub/su/scsh/scsh.tar.gz*. Other Scheme dialects can also be used for scripting, however, the Scsh offers the integration into the Unix tools. So why should you resort to Scheme for shell programming? The author of the Scheme shell, Olin Shivers, has the following to say on the subject:

"Unix shells are real programming languages. They have variables [...]. But they are terrible programming languages. The data structures typically consist only of integers and vectors of strings. The facilities for procedural abstraction are minimal to non-existent. The lexical and syntactic structures are multi-phased, unprincipled and baroque."

This fairly harsh criticism is aimed mainly at the programming language aspect. Also ignored are the many functions which make working on the command line so agreeable. This is the area in which the Scsh has great weaknesses. The length of

the Unix FAQ indicates that shell programming is not an easy subject.

Renaming a number of files

The question regarding the renaming of a number of files can also be found under 2.6 of the Unix FAQ, so it isn't a trivial question, otherwise it certainly wouldn't have ended up there. Here is a possibility for a Bourne shell:

```
function rename_1 (){
  from=$1
  to=$2
  mv $from $(echo $to |\
tr `[A-Z]' `[a-z]').html
}
for f in *.HTM; do
  base=`basename $f .HTM`
  rename_1 $f $base
done
```

Let's apply this to the following files: *T1.HTM, T2.HTM, "with blank.HTM"* and *T3.HTM*. The resulting output is: *mv: when moving several files the last argument must be a directory*. Well, this error message is not exactly helpful. Let's have a look at what might have gone wrong. Using *ls*, we can see that the renaming has worked for all files apart from the one with blanks. The problem is therefore one of blank spaces in filenames, and we can forget about the reference to a directory. So let's try it a different way:

```
function rename_2 (){
  from=$1;
  to=$2
  mv $from "$(echo $to |\
tr `[A-Z]' `[a-z]').html"
}
```

The following files exist: *t1.html, t2.html, with blank.html* and *t3.html*. The second method seems to work. However, to use umlauts or other strange characters in file names you need to know a lot about quoting, process substitution and special characters. Now here's the whole thing in the Scheme shell:

```
#!/usr/local/bin/scsh \
-l ./stringhax.scm -s
!#
(define (rename-them)
  (let ((file-list (glob "*.HTM")))
    (for-each
      (lambda(x) (rename-file x
         (replace-extension
           (downcase-string x) ".html")))
      file-list)))
(rename-them)
```

Applied to the same files we get the expected result directly. Now let's have a look at the structure of a Scsh script: as in every script, the first line specifies what sort of program is to process this script. The following lines are command line arguments for the Scsh, as listed in chapter 2 of the Scsh reference manual.

In the example we are first loading a file (*stringhax.scm*) that contains some extensions for handling character strings; the only thing required here is the procedure *downcase-string*. *-s* limits the extended command line and must always be the last option before *!#*.

We are going to define the procedure *rename-them*. *glob pattern* generates a list, whose elements correspond to *pattern*. Subsequently each individual file in the list is renamed in a loop. This requires several procedures: one to rename the files, one to replace the extension and the third to change the file names to lower case. The call to the newly defined procedure completes the script.

Scheme shell characteristics

Perhaps this example has managed to convince you that shell programming can be simplified by using a

Listing 1: Exception handling in the Scsh

```
(define (error-handling-test)
  (let ((dir "dir"))
    (create-directory "dir")
    (with-errno-handler
     ((e1 d1)
      ((errno/exist)
       (format #t "errno-msg = ~A~%
                   syscall = ~A~%data = ~A~%~%
                Directory exists~%"
               (nth d1 0) (nth d1 1) (nth d1 2)))
      (else (error "Do not know what to do now, giving up\n")))
     (create-directory dir))
    (with-errno-handler
     ((e2 d2)
      ((errno/isdir)
       (display "Oops, was a directory
                so running delete-directory instead\n")
       (delete-directory dir)))
     (delete-file dir))))
```

Listing 2: Awk, the Scheme way

```
#!/usr/local/bin/scsh -s
!#
;;; Add the numbers of kb of free and used disk space for all the volumes
;;; reported by df(1)
(define (df-free+used)
  (let ((df-read (field-reader))) ;df 1
    (exec-epf
     (| (df) ;df 2
        (begin
          (read-line) ; skip first line of df ;df 3
          (awk (df-read) (line fields) ;df 4
               ((used-kb 0) (free-kb 0)) ;df 5
               (#t  ;df 6
                 (values (+ used-kb (string->number (nth fields 2)))
                         (+ free-kb (string->number (nth fields 3)))))
               (after
                (values `(`used-mb ,(/ used-kb 1024.0)) ;df 7
                        `(`free-mb ,(/ free-kb 1024.0)))))))))
(receive
 (used-list free-list)
 (df-free+used) ; df 8
 (format #t " ~A MB are being used and ~A MB are still free.~%"
         (cadr used-list)
         (cadr free-list)))
```

fully-fledged programming language. For a link to the underlying system the Scsh offers:

- mechanisms for integrating data from Unix programs and the Shell
- a POSIX-API converted to Scsh.

Process control

Normally, every program started under Unix becomes a separate process. Shell programs often consist of a combination of several programs, linked, for instance, by pipes. The Scsh therefore requires channels for linking the programs. Here is a simple example:

```
(define (example-1-copy from to)
(run (cat ,@from)
      (> ,to)))
```

The procedure is expecting a list of files to be concatenated with *cat* and written to *to*. *run*, together with other process control elements, is implicitly backquoted. For example, a file *t1.txt* can be accessed in the three following ways:

```
* (run (cat "t1.txt"))
* (run (cat t1.txt))
* (define file "t1.txt") (run (cat ,file))
```

Capitalisation is important in the Scsh (unlike "standard Scheme"), as Unices differentiate between upper and lower case. This extension seems appropriate in order for the Scsh to adapt to that requirement. A mechanism such as *run* is not sufficient for integration, which is why there are additional procedures available for process control, such as procedures for diverting input/output, job control and pipes. Here is an example of a pipe with the Scsh:

```
(run (|(cat foo bar)
      (grep -n "\\<foo\\>"))
    (> foo_lines))
```

Exchange of data between service programs and Scheme

If you want to access all Unix programs you need the facility of making service program data available to the Scheme shell. This is achieved using the *run/xxx* procedures. If, for instance, you require a list of the files in the current directory, you enter *(run/strings (ls))*. The result is a list of character strings with the names of the files which can be processed by all Scheme programming elements.

Other *run/xxx* commands are used to write to ports or temporary files, or to convert the output into a single character string. For instance, *(run/string (cat foo bar))* gives you a character string consisting of all lines of the two files.

An interesting example for Linux is the command *run/port+proc*, which return additional information on the split process, such as output status, current status, and others.

```
(define (run/port+proc-test)
(receive (port process) (run/port+proc (ls -l))
         (let ((out (port->string port))
               (exit-status (wait process)))
      (display "the pid of the process was ")
          (display (proc:pid process))
          (display ", exit status = ")
          (display exit-status)
          (newline)
         out)))
```

The Scsh uses the form *receive* to bind several return values. The binding parameters are contained in the first list (here *port* and *process*), afterwards the procedure which will return these values is called (*run/port+proc*).

In our example the output from *ls -l* is written to the port, and, using *port->string*, this output is converted to a character string, which constitutes the return value of this procedure. The output status of the program started with *run/port+proc* is recorded using *exit-status*. *proc:pid* accesses the element *pid* in the *proc* structure, so the Scsh handles this differently from DrScheme.

Exceptions

Basically the Scheme shell wraps all POSIX system calls in Scheme syntax. Should error occur, exceptions are raised which can be processed (see listing 1).

Exceptions are processed as follows: The rump of the exception is executed first, such as, the procedure *create-directory* in the first error handling routine, and *delete-file* in the second.

Listing 3: Creating a colourful table

```
(load (string-append laml-dir "laml.scm"))
(laml-style "simple-html4.0-loose")
(generic-page-1
  "A page with a very colourful table"
  (con-par
     "This pages shows some elements from the
      HTML 4.0 convenience library"
     (em "Emphasis is not a problem"
         (b "bold") "also works.")

     "Here is another colourful table, keep your eyes open ;-)"
     (table-1
       2 ; border
       (list 70 100 100 100) ; four columns
       (list (make-color 106 90 205)
             (make-color 0 255 255)
             (make-color 127 255 212)
             (make-color 238 121 159)) ; "wonderful" colours;
       (list
        (list "Here" "a" "table" "with")
        (list "the" "most" "tasteful" "colours" "a")
        (list "work" "of" "enduring" "value")) ; text in table
       "middle")
     (b "Here ends this `nice' page, but not without")
     (con "referring to another `masterpiece'"
          (a-tag "simple.html" "simple")))  ; a link
  ;; colours for this page
  (make-color 255 255 191) (make-color 205 105 201) blue red
  )
(end-laml)
```

Should an exception occur, the first parameter of the error handling form (*e1*) is bound to the value of *errno*, the second parameter (*e2*) to a list. The latter consist of an error message which equates to the output *perror errno*, the failed system procedure (this parameter can be used to call the failed method again) as well as other information which differs according to the system procedure involved.

Afterwards the individual error handling cases are examined in turn, with the return value of the executed branch becoming the value of the entire error handling.

In our example we skip the creation of the directory and return an error message where a directory of the same name already exists. If it is not possible to delete the directory in the second part, an unsuitable procedure (*delete-file*) is called, which will inevitably generate an exception error. To remedy this the correct delete procedure (*delete-directory*) is then called.

System calls and additional procedures

The Scsh offers a great deals of other procedures for a range of different areas, which can only be mentioned briefly here:

- input/output (Scheme standard procedures and extensions)
- file/directory processing and information
- globbing
- other process control elements
- Signals and interruptions
- time-handling procedures
- environment variables
- terminal control
- regular expressions
- network programming
- bit manipulations

Especially interesting is the conversion of so-called "little" languages into Scheme syntax, for example the *awk* macro of the Scsh. In our example (see listing 2) the total used and free disk space on the hard disk is calculated. For individual partitions this could be done using *df*, however, there is no service program that will show you used and free disk space for an entire hard disk.

The returning of Scheme expressions (; df 7) follows a recommendation from Olin Shivers. He remarked that this makes it easier to process data in other procedures, as you can utilise the *read* functionality.

Let's examine the program a little more closely: the output of *df* is subject to a consistent format (file system "Total Size " "Bytes Used" "Bytes Free" ..\n"). These are several fields which are separated by spaces. In the Scheme shell the reading in of lines has been separated from the processing of the current line. The reading in is performed by what are called "field readers" (; df 1). The output from *df* is "piped" to Scheme (; df 2).

Since the first line of the *df* output contains text which is irrelevant for the capacity calculations, it is simply ignored (; df 3).

The Scsh "field reader" returns two values: the complete line as well as a list of individual elements. The lines are split into words using specific field separators. Where no particular characters are specified, white spaces are regarded as separators. We define two variables (; df 5), which will contain the total of used and free kilobyte. In order to understand the rest of *awk* you have to imagine the way in which the original *awk* works. An *awk* script is surrounded by a loop that reads in data line by line (or even several lines at a time, depending on the field separator). *awk* scripts are also based on an implicit case differentiation, in which the current line is compared with regular expressions.

In the Scsh equivalent of *awk* additional elements can be used: Boolean values, regular expressions and Scheme expressions. In our case we are interested in all lines, as each line lists the used and free bytes of the corresponding file system. Therefore we are simply using *#t* (; df 6) for testing, as that applies to each individual line.

The local variables are updated for each line that is read in. The third element of the list is added to *used-kb* (it is almost always equivalent to the used kilobytes in the file system that is currently being examined). The fourth list element and *free-kb* are dealt with in a similar way. Finally the kilobytes are converted to megabytes (; df 7) and two return values in the form of *(symbol ,(calculated Mbytes))* are returned.

The call to the newly defined procedure (; df 8) forms the conclusion of this script. The output on my computer is:

```
3976.94 MB are being used and 2885.82 MB ar↵
e still free,
```

so there is still a bit of space for a few more articles ;-).

Since the interaction of the procedures for reading in the data and processing by the *awk* macro is confusing at first glance I would recommend reading chapter 8 of the Scsh reference manual, where this interaction is documented in detail.

Network programming

The Scsh offers a few procedures for this as well. These can be divided into "basic procedures" and "usability procedures" based on the basic ones. Using the latter, it is pretty easy to create a Scheme server (see box Scheme server and client)

It really is a Scheme server in only 23 lines of program.

Scheme and the Internet

Functional or declarative languages like Scheme are tailor-made for use in today's Internet services. You have seen the basic elements in the previous example, however, based on this, Scheme enthusiasts have implemented a number of additional services. For example a HTTP server,

procedures for mail handling, modules for HTML-formatted output and CGI-programming. There are further service programs in the DrScheme net collection for SMTP, NNTP, DNS, POP3 and IMAP. So even for those programs you do not have to resort to other scripting languages.

Scheme as a markup language

The most prominent example of a markup language at the moment is HTML. The term "language" is misleading, however, as HTML is not a programming language but a tool for describing

Scheme server and client

Client

```
(define (send-to-server form)
 (let* ((sock (socket-connect
          protocol-family/internet
          socket-type/stream
          "localhost" test-port))
     (op (socket:outport sock)))
  (format #t "I'll send ~A~%" form)
  (cond
   ((number? form)
    (write (number->string form) op))
   (else
    (write form op)))
  (force-output op)
  ;(sleep 1)a
  (let ((result (read (socket:inport sock))))
    (format #t "I got ~A from the Server~%" result)
    (force-output (current-output-port)))
  (close-socket sock)))
```

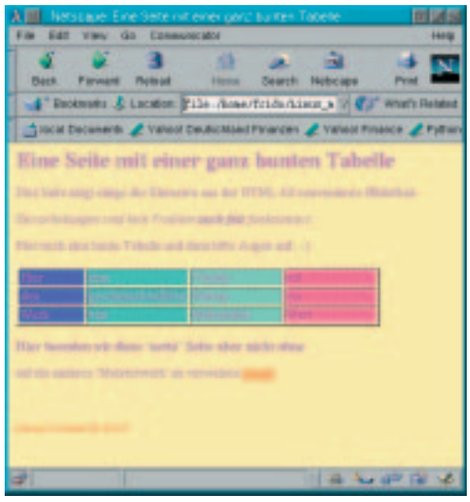
Server

```
(define test-port 1111) ;sk 1
(define (scheme-server)
  (let* ((log-file-name "./scheme-server.log") ;sk2
        (log-file (open-output-file
                  log-file-name
                  (bitwise-ior open/append open/create)))) ;sk2
   (bind-listen-accept-loop
    protocol-family/internet
    (lambda (sock addr)
      (let* ((op (socket:outport sock))
            (input (read (socket:inport sock))))
        (format log-file "Got:  ~A~%" input) ;sk 2
        (force-output log-file)
        (let ((output (eval input (interaction-environment))))
          (format log-file "Wrote: ~A~%" output) ;sk 2
          (force-output log-file)
          (cond
           ((number? output)
            (write (number->string output) op))
           (else
            (write output op)))
          (force-output op))))
    test-port)))
```

The server in this example waits for requests at port 1111 (; sk1). No error handling is provided, however, the entire data exchange is logged (; sk2). The procedure bind-listen-accept-loop is a wrapper around basic procedures such as create-socket, connect-socket, bind-socket and others. Extensions like receive-message and send-message are available for the output to sockets; or you could simply use standard Scheme procedures, like read and write in our case.

Some application examples:

```
(send-to-server "foo") -> I got foo from the Server
(sent-to-server `(+ 1 2)) -> I got 3 from the Server
(send-to-server `(map (lambda (x) (+ 1 x)) `(1 2 3)))
 -> I got (2 3 4) from the Server
```



[left]
Figure 1:
A table generated by LAML

[right]
Figure 2:
Lecture template

pages. For more advanced elements (such as loops and case differentiation) you will have to fall back on a real programming language.

How about adding markup elements to a programming language? That's exactly what Kurt Normark thought, and so he developed LAML (Lisp Abstracted Markup Language). LAML is divided into separate layers. The lowest one contains only "wrappers" around the different HTML tags. You can read about how the layers are structured in the comprehensive documentation of LAML. Here a first example of LAML:

LAML files are translated to HTML using a suitable Scheme implementation, while MzScheme is currently the LAML development Scheme. How you translate the files depends on the environment in which you are working. There is a *laml-mode* for the (X)Emacs, with which you can start the translation of LAML files directly from the editor. Alternatively you can call the LAML script (this is a MzScheme script) or load the appropriate files into a Scheme and call the LAML procedure *(laml "file.laml")*. Another example for the use of LAML can be found in listing 3.

I'm sure you will agree with me that the colour selection in the table is easy on the eye.

The LAML sources contain a number of examples and good documentation. Some of the examples are pretty clever templates, amongst other things a slide show, documentation tools for Scheme, a calendar and schedules.

Finally, please have a look at the following page, which is derived from a lecture template.

Summary

This part has introduced Scheme as a "scripting" and markup language. The examples show how flexibly Scheme can be used. Learning Scheme provides you with a tool that allows you to replace a number of other ones. The seamless conversion of the various elements is certainly another advantage of Scheme. You are always able to fall back on the full functionality of a programming language, there are practically no limitations such as you can encounter with other programs. In my opinion these are reasons for learning Scheme.

Outlook

In the next part about Scheme the focus is once again on the Internet. We will also be using Scheme for CGI programming and in connection with Java. There are Schemes that have been developed specially for the interaction with Java and which allow you seamless access to Java classes. As always I would be happy about any comments and suggestions, my email address is: *frido@q-software-solutions.com* ■

Info

"A Scheme Shell", Olin Shivers, SCSH Paper (part of the sources)
"Scsh Reference Manual", Olin Shivers (also part of the sources)
Unix FAQ: ftp://rtfm.mit.edu/pub/usenet-by-hierarchy/comp/unix/answers/unix-faq/
SCSH Home Page: http://www-swiss.ai.mit.edu/ftpdir/scsh/
LAML Home Page: http://www.cs.auc.dk/~normark/laml/

■

First steps in LAML

```
(load (string-append laml-dir "laml.scm"))

(laml-style "simple")

(generic-page-1

  "Greetings from LAML"

  "Especially for you one of the most impressive masterpieces

   of LAMLic programming ;-)")

(end-laml)
```