Running Scheme on Pluto

There are many implementations of Scheme language. Guile is the *GNU Ubiquitous Intelligent Language for Extensions*, the official extension language for the GNU operating system. Guile is an interpreter and compiler for the Scheme programming language, a clean and elegant dialect of Lisp. Guile is up to date with recent Scheme standards, supporting the Revised^5 and most of theRevised^6 language reports (gnu.org).

Running Guile Interactively

In its simplest form, Guile acts as an interactive interpreter for the Scheme programming language, reading and evaluating Scheme expressions the user enters from the terminal. Here is a sample interaction between Guile and a user; the user's input appears after the $ and guile> prompts:

```
$ guile                         ;start the guile interpreter
guile> (+ 1 2 3)                  ; add some numbers
6
guile> (define (factorial n)      ; define a function
         (if (zero? n) 1 (* n (factorial (- n 1)))))
guile> (factorial 20)           ;run the factorial
2432902008176640000
guile> C-d                      ;exit guile or use (quit)
$
```

Guile Scripts

Like AWK, Perl, or any shell, Guile can interpret script files. A Guile script is simply a file of Scheme code with some extra information at the beginning which tells the operating system how to invoke Guile, and then tells Guile how to handle the Scheme code.

Here is a trivial Guile script:

```
#!/usr/bin/guile -s
!#
(display "Hello, world!")
(newline)
```

The first line of a Guile script must tell the operating system to use Guile to evaluate the script, and then tell Guile how to go about doing that. Here is the simplest case:

- The first two characters of the file must be `#!'. The operating system interprets this to mean that the rest of the line is the name of an executable that can interpret the script. Guile, however, interprets these characters as the beginning of a multi-line comment, terminated by the characters `!#' on a line by themselves.
- Immediately after those two characters must come the full pathname to the Guile interpreter. On pluto, this would be `/usr/bin/guile'.

- Then must come a space, followed by a command-line argument to pass to Guile; this should be `-s'. This switch tells Guile to run a script, instead of soliciting the user for input from the terminal. There are more elaborate things one can do here; see section The Meta Switch.
- Follow this with a newline.
- The second line of the script should contain only the characters `!#' -- just like the top of the file, but reversed. The operating system never reads this far, but Guile treats this as the end of the comment begun on the first line by the `#!' characters.
- The rest of the file should be a Scheme program.

Guile reads the program, evaluating expressions in the order that they appear. Upon reaching the end of the file, Guile exits.

Continue the example above, if you put that text in a file called `hello' in the current directory, then you could make it executable and try it out like this:

```
$ chmod a+x hello
$ ./hello
Hello, world!
$ guile hello                  ; another way of running the saved script file
Hello, world!
```

Here is another script (saved as "fact") which prints the factorial of 5:

```
#!/usr/bin/guile -s
!#
(define (fact n)
  (if (zero? n) 1
    (* n (fact (- n 1)))))

;display the result

(display (fact 5))
(newline)
```

In action,

```
$ chmod a+x fact
$ ./fact
120
$guile fact
120
```

Yet another example in our lecture notes:

```
#!/usr/bin/guile -s
!#

(define foo
(lambda (L)
 (car (cdr L))
))

; apply it
(display (foo '(s c h e m e)))
(newline)
```

In action,

```
$ chmod a+x foo
$ ./foo
c
$guile foo
c
```

```
Note: You do not have to use the approach described below if you are not
comfortable with script file taking arguments from command line.
```

In all the examples, the function `command-line` returns the name of the script file and any command-line arguments passed by the user, as a list of strings. If the program invoked Guile with the `-s', `-c' or `--' switches, the function ignores everything up to and including those switches

Example 1 (fact_display):

```
#!/usr/bin/guile -s
!#
(define (fact n)
  (if (zero? n) 1
    (* n (fact (- n 1)))))

(display (fact (string->number (cadr (command-line)))))
(newline)
```

In action:

```
$ chmod a+x fact_display
$ ./fact_display 5
120
$guile fact_display 5
120
```

Example 2(fact_main):

```
#!/usr/bin/guile \
-e main -s
!#
(define (fact n)
  (if (zero? n) 1
    (* n (fact (- n 1)))))

(define (main command-line)
  (display (fact (string->number (cadr command-line))))
  (newline))
```

In action:

```
$ chmod a+x fact_main
$ ./fact_main 5
120
```

Example 3 (foo_main):

```
#!/usr/bin/guile \
-e main -s
!#

(define foo
(lambda (L)
 (car (cdr L))
))

(define (main command-line)
  (display (foo (cdr command-line)))
  (newline))
```

In action:

```
$ chmod a+x f00_main
$ ./foo s c h e m e
c
```

Reference:

http://www.cs.rit.edu/~afb/20013/plc/guile/guile_6.html#SEC9