CSE 341: Homework 2

due 01/29/01

All functions you are asked to implement and turn in should be typed and tested with the scheme interpreter. With each function turn in a hardcopy of the source code listing as well as sample output for a demonstrative set of tests. Do not forget to test *end cases*. Also, use the `turnin` program to turn in an electronic copy of your source code.

This homework also contains a small project. The end result of this project will be a simple web guest-book application. Typical web guest-books work by allowing people on the web to fill in a form on your web page and submit this form to a program running on the web server. This program can then save the data submitted to a file containing the guest-book information. The guest-book program has to communicate with the web server in order to get its input data, what the person that filled in the form typed, and to output data back to the server to be shown to the person after they submit their data. This is done via the *Common Gateway Interface* (CGI). CGI is a very simple interface that will be explained in more detail later in this assignment. You are to have your web application installed and running on `cubist.cs.washington.edu`. By now, you should have received email account information for `cubist`. An example of the guest-book application you will be writing is available at: `http://gurgle.cs.washington.edu/~hartline/cgi/show-guestbook.cgi`.

Part I: (Not to be turned in.)

Try these questions out before section on 01/18/01. First try to figure out the answers in your head. Then try typing them into the scheme interpreter.

1. Given the following definitions,

```
(define x 5)
(define y 4)
```

What would the values of the following scheme expressions?

(a)
```
(let ((x 3)
      (z (+ x 1))
     )
     (* x y z)
)
```

(b)
```
(let* ((x 3)
       (z (+ x 1))
      )
      (* x y z)
)
```

(c)
```
(let* ((z (+ x 1))
       (x 3)
      )
     (* x y z)
)
```

(d)
```
(list
   (let ((x 3)
         (z (+ x 1))
        )
        (* x y z)
   )
   x y
)
```

2. Given definitions

```
(define x 5)
(define y 4)
(define (foo x)
   (+ x y)
)
(define (bar y)
   (foo y)
)
```

What is the value of

```
(bar 10)
```

3. Write a function `flatten` that takes a *tree* and returns a list that contains the leaves of the tree in left to right order. For Example:

```
(flatten '(1 ((2) 3) (4)))        => (1 2 3 4)
```

Part II: (Not to be turned in.)

These are some sample problems that we will be discussing in section on 01/18/01. Use the `error` function for handling errors.

1. Define a tail-recursive version of `factorial`.
2. Partially evaluating a function by passing its arguments to it one argument at a time is called *currying* after Haskell B. Curry. The idea is that given an n argument function, you can specify one of the arguments and be left with an $n-1$ argument function. Implement the function `curry` that takes as its own arguments a two argument function f and an argument x and returns a new function g that takes only one argument, say y, such that $g(y) = f(x, y)$. This is illustrated with the following scheme code and example:

```
;; assuming f has no side-effects and that
;; x and y are defined and are valid arguments to f
(define g (curry f x))
(equal? (g y) (f x y))                  => #t
```

```
;; more examples
(define add5 (curry + 5))
(define add2 (curry + 2))
(add5 10)                              => 15
(add5 2)                               => 7
(add2 10)                              => 12
```

3. Define a function `filter` that takes, as arguments, a function `f` and a list l. It returns a new list that only contains elements of `l` for which `f` returns true. For Example:

   ```
   (filter number? '(a b 1 3 c 4 d))      => (1 3 4)
   (filter symbol? '(a b 1 3 c 4 d))      => (a b c d)
   (filter symbol? '())                   => ()
   ```

4. Use your `curry` function along with `filter` to implement `positive-only`, a function that takes a list and returns only the positive elements. Do not define any other new functions or *lambdas* explicitly. Use only `curry`, `filter`, and builtin scheme functions.

Part III: (Not to be turned in.)

Try these questions out before section on 01/25/01. First try to figure out the answers in your head. Then try typing them into the scheme interpreter.

Given definitions

```
(define a '(1 2 3))
(define b '(a b c))
(define c '(#t #f))
(define d '(4))

(define x (cons 10 a))
(define y (append a b))
(define z (list a c))

(set-car! a 99)
(set-car! b 'z)
(set-cdr! c '(9 10 11))
(set-cdr! d d)
```

What are the values of:

1. `x`
2. `y`
3. `z`

4. `c`
5. `(car d)`
6. `(cadr d)`
7. `(caddr d)`
8. `(cadddr d)`

Part IV: (Not to be turned in.)

These are some sample problems that we will be discussing in section on 01/25/01. Use the `error` function for handling errors.

1. Write a function `repeatr` and `repeatl` that take a function f and a number k. The result will be the list created by calling the f with no arguments k times and putting the results in a list. `repeatr` and `repeatl` will only do different things if the function f has side-effects. For `repeatr` the first call to f is placed in the last element of the list created. The opposite for `repeatl`. For example:

```
(define x 0)
(define (f)
   (set! x (+ x 1))
   x
)

(repeatl f 3)            => (1 2 3)
(repeatr f 3)            => (6 5 4)
```

2. CGI scripts are typically invoked in one of two ways. Either as a *post* or as a *get*. *Posting* is the typical method for submitting forms and *getting* is the method for retrieving data for some query. For example, if you send email on the web (E.g. with Hotmail), you are invoking their CGI with a *post*. On the other hand, if you do a web search (E.g. with *Google*) you are doing a *get*. The HTML form looks something like this for a *post*:

```
...
<FORM ACTION=emailform.cgi METHOD=POST>
<INPUT TYPE=TEXT NAME=TO SIZE=40>
<INPUT TYPE=TEXT NAME=SUBJECT SIZE=40>
<TEXTAREA NAME=TEXT COLS=60 ROWS=10>
</TEXTAREA>
<INPUT TYPE=SUBMIT VALUE="Send Email">
</FORM>
...
```

The HTML form for a *get* looks like this:

```
...
<FORM ACTION=websearch.cgi METHOD=GET>
```

```
<INPUT TYPE=TEXT NAME=QUERY SIZE=40>
<INPUT TYPE=SUBMIT VALUE="Search">
</FORM>
...
```

The following CGI script prints out "Hello World" in a web browser when invoked:

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

(display "Content-type: text/html\n")       ; Header: Output is HTML.
(newline)                                   ; End of headers.

(display "<BLINK>Hello World</BLINK>")
```

Some discussion of the above CGI program:

- The first two lines are a bit of UNIX magic that turns your guile file into an executable program. If the first two characters of an executable text file are `#!` then the remainder of the first line is executed with the contents of the file as its standard input. Here we take this file and run it as the standard input to the command `guile -s`. The `-s` command-line switch to Guile tells it to run as a script. That is, instead of doing the read-eval-print loop, it just does a read-eval loop. In order to make your Guile script executable, you must make sure to run `chmod +x` on it. This changes the permissions so that it can actually be run like any other UNIX program. This `#!` magic notation is standard for writing shell scripts in UNIX. It works because `#` is the comment character for most shell scripting languages (E.g. sh, tcsh, perl, tcl). So when we execute the shell on the contents of the file, the first line is commented out. In Scheme, the `#` character is not a comment character. To make this work in a consistent way (and not be just a hack so that Guile programs can be run as shell scripts), Guile has added the comment pattern, `#! this is a comment !#`, which works in a similar way to C's comments, `/* this is a comment */`. This comment pattern is special to Guile and is not part of standard Scheme.
- CGI scripts send their output back to the web server (and then to the web browser) via their standard output. The second two lines in this Guile CGI program tell the web server (and browser) that the content being returned is an HTML web page. These are the *script headers*. If you ever get an error trying to run your CGI script that says:

  > The server encountered, like, an internal error or misconfiguration and was unable to complete your request "..." Bummer!

  It means that your program was not executable or that it tried to print stuff out that did not look like the CGI script headers. There will be a link at the bottom of the error message that will sometimes give you more details. Usually it will say *premature end of script headers*. Typically when a Guile program crashes (one of the reasons for getting a premature end of script headers) it prints out messages to standard error. These messages can be looked at by viewing the system web server error log. One way to view this error log is to type, `tail /www/htdocs/logs/error_log`, from the `cubist` prompt. This prints out the end of the error log file.

- The last line in this script prints out the actual content of the web page being returned.
- This CGI script is available for testing at `http://cubist.cs.washington.edu/~hartline/hello.cgi`. This is located in user `hartline`'s `www` directory. CGI scripts must end with the `.cgi` suffix so that the web server knows that the script should be run. For example, the URL `http://cubist.cs.washington.edu/~hartline/hello.txt` just returns the text contents of the script because it does not end in `.cgi`.

CGI output can get fancier than this *hello world* program, but most CGI is not. CGI input is communicated from the web server to the CGI program through UNIX environment variables and sometimes the standard input. In Guile, you can get the value of a UNIX environment variable with the function `getenv` which works as follows (`getenv` is not a standard Scheme function):

```
guile> (getenv "USER")
"hartline"
guile> (getenv "SHELL")
"/usr/bin/tcsh"
```

You can tell whether the CGI program is invoked as *get* or *post* by looking at the environment variable `REQUEST_METHOD`. It will have value `"GET"` or `"POST"` respectively.

POST If the method is *post*, the environment variable `CONTENT_LENGTH` will contain number of characters in the input. This input will be sent to the CGI program on its standard input.

GET If the method is *get*, the entire CGI input is stored in the environment variable `QUERY_STRING`.

Modify the *hello world* CGI script above to read in the CGI input and print it out. You may use `<PRE>...</PRE>` tags in your output HTML so that the output is printed with a fixed width font. See `http://cubist.cs.washington.edu/~hartline/test-forms.html`.

Part V: (Due Monday, 01/29/01. Turn this in.)

Complete the following. You do not have to handle errors in input gracefully; however, if you want to you may use the `error` primitive function; e.g. `(error "error in input")`. Do not forget to turn in a print out of your source code as well as example executions of your code on test cases.

Some of the functions you are asked to implement in this assignment should call functions your wrote in Assignment 1. You may use either your own solutions from Assignment 1 or the posted solutions. However, please include these in your electronic turnin.

1. (5 points) We wrote `foldr` in class by abstracting `sum`. Write `foldl`, the left associative fold, by abstracting the tail-recursive version of `sum`. Use `foldl` to define `reverse`, the function that takes a list are returns a reversed copy of it. For Example:

   ```
   (foldl + 0 '(1 2 3))          => 6
   (reverse '(1 2 3))            => (3 2 1)
   ```

2. (10 points) Recall our own definition of *cons cells* using `lambdas` from lecture:

   ```
   (define (my-cons the-car the-cdr)
      (lambda (which)
         (cond ((eq? which 'car) the-car)
               ((eq? which 'cdr) the-cdr)
               (else (error "cons cell usage error"))
         )
      )
   )

   (define (my-car the-pair)
      (the-pair 'car)
   )

   (define (my-cdr the-pair)
      (the-pair 'cdr)
   )
   ```

   Define `my-set-car!` and `my-set-cdr!`. You may make any modifications to the above definitions that you need (but you may not use primitives `car`, `cdr`, `cons`, `set-car!`, or `set-cdr!`).

3. (5 points) Input is supplied to the CGI program as a string of text of the form:

   `NAME1=VALUE1&NAME2=VALUE2&NAME3=VALUE3`

   Here `NAME1`, `VALUE1`, etc. are arbitrary strings that are encoded such that:

   - Strange symbols (including `+`, newline, and tab) are converted to their hexadecimal value and prepended with the `%` character. For example `"+"` is encoded as `"%2b"`.
   - Whitespace is replaced with the `+` character. For example `"this is a test"` is encoded as `"this+is+a+test"`.

You'll note that the `decode-hex-string` from Assignment 1 will correctly decode `NAME1`, `VALUE1`, etc.

Write a function `parse-cgi` that takes a string of the above form as its only input and returns an *association list* (R^4RS 6.3). That is a list of lists of the form:

```
((NAME1 VALUE1) (NAME2 VALUE2) ...)
```

For example:

```
(parse-cgi "FOO=this+is+a+test&B+A+R=In+hex+5+%2b+5+is+%27a%27.")
   => (("FOO" "this is a test") ("B A R" "In hex 5 + 5 is 'a'."))
```

Your implementation of `parse-cgi` should not be recursive (nor should your write any recursive helper functions besides `decode-hex-string` and `split` that you wrote in Assignment 1). Instead, use `map`.

4. (15 points) Now create a web guest-book application. The minimum requirements are that users can enter their name and a brief comment into a form, and that a user can also view all the other comments already submitted. You have the freedom to do this however you choose; however, the following is recommended:

   - Write one CGI program that reads form data with the user's name and comment and appends it to a file.
   - Write another CGI script that just reads the names and comments from the file and creates a HTML document for the user to view.

   See `http://gurgle.cs.washington.edu/~hartline/cgi/show-guestbook.cgi` for an example implementation.

   To open a file for appending, use the Guile builtin function `(open-file <file-name> "a")`. This returns a file handle much like `(open-output-file <file-name>)` does, except that new data written to the file will go at the end of the file instead of replacing its current contents (`open-file` is not a standard Scheme function).

   You will be graded on the style; your organization of scheme code, appropriate use of side-effects, and use of higher order functions; as well as the correctness of your implementation.