CS 351 Programming Assignment 5

Due: November 6, 2014

Web pages are written in a markup language called HTML. One common element of Web pages is the **form**. Forms are usually used by a Web server to collect data. Note that the data sent to the server is a long string of text that shows the "field_name=value" pair for each field, and an & is used to separate fields. Spaces in the form data are represented by +'s, and the character '+' itself is represented by its ASCII code in hexadecimal %2B, % by %25, / by %2F, etc. For example, if a software registration form has the following fields and the corresponding values:

Field Name	Value
FirstName	Leh-Sheng
LastName	Tang
Address 1	Western New England University
Address 2	1215 Wilbraham Rd.
City	Springfield
State	MA
ZIP	01119
Country	United States
Product	Visual C++.Net
Date	10/30/2012
Discount	30%
Comments	It is very difficult to use the software.

then the string sent to the server is

```
FirstName=Leh-Sheng&LastName=Tang&Address+1=Western+New+England
+University&Address+2=1215+Wilbraham+Rd.&City=Springfield&State=
MA&ZIP=01119&Country=United+States&Product=Visual+C%2B%2B.Net
&Date=10%302F2005&Discount=30%25&Comments=It+is+very+difficult
+to++use+the+software.
```

Here we show the data in several lines, but the actual data is a long string of characters. Note that even though HTML allows you to create forms, it cannot process the data received from the form. One way to process the data is to send it to a program, called a **CGI script**, on the server. Two most commonly used languages for CGI programs are C and Perl. The **CGI** (**Common Gateway Interface**) will act on the data and perform certain task. To extract data from the string, the CGI program has to parse the string.

In this assignment you are to write a C library, let's call it **cgilib.h**, that allows users to parse such strings and extract data.

The program consists of three files.

1. **cgilib.h**

   The cgilib.h library contains some type declarations and three functions:

```
typedef char *string;    /* dynamic array type */

typedef struct
{
     string fieldName;
     string value;
}oneField;

struct element {
     oneField       data;
     struct element *next;
};

typedef struct element   node;
typedef node             *link;

link tokenize(string input_string);
string cgi_val(link head, string field);
void print_table(link head);
```

2. **cgilib.c**

   This file contains the implementation of three library functions.

   (1) **link tokenize (string input_string);**

   This function is used to parse an input string, and returns a linked list of tokens (oneFields). The linked list is maintained by the alphabetical order of the field name, and the data stored in each node is a record representing one field.

   Note that both fieldName and value are **dynamic** strings of characters. That is, both can be strings of any length. (**How??**)

In order to put strings of any length in the linked list, you must dynamically generate the string. But **how??** You should write a function that extracts one word at a time:

```
string next_word(string input_string, int *pos);
```

This function will extract a word from the input string, starting at the given position specified by pos. If such a word exists, the function will dynamically create an array and put that word in the array, and return the pointer to the word. If the end of string is reached, the function should return NULL, so the caller knows that no more words can be found. Note that after you put all characters in the output string, **you must place the NULL character ('\0') at the end of the string.** Also, after reading the current word, **pos should be set to the proper position, so the next read will start at that position.**

But how do we extract and create a word of any size? You need to step through the input string twice. The first time is to scan the string from the current position, character by character, to count the number of characters of the next word. **Don't forget that certain characters are stored as 3 characters.** Assume the next word has n characters (after the conversion). You now create the word, a string of n+1 (**???**) characters, using the **calloc** function. You then go back to pos, and extract the next word. **Don't forget to do the conversion, such as '+' to a space, %2B to a '+', etc. Finally put the NULL character ('\0') at the end of word.** Before you return from the function, don't forget to set pos to the proper position.

(2) **string cgi_val(link head, string field);**

This function will step through the linked list pointed to by head, find the node containing the given field, and return its value. Note that the function returns a dynamic string. Once you find the field, use strlen to determine the length of its value, then use calloc to dynamically create a string of size length+1 (**???**), copy over the string, and finally return that pointer. If you cannot find the field, return the NULL pointer.

Note that your code should effectively use the fact that the linked list is ordered by the field name.

(3) **void print_table(link head);**

This function simply steps through the linked list, and prints a table that shows all fields and the corresponding values.

3. **The main program**

Your main program should have a function that reads the input string:

```
string get_input(void);
```

The function should prompt the user to enter the input data file name, then read the input string from the data file. Note that the input string can also be of any length. So, use the same technique above to read the file twice, first to determine the number of characters in the input string, so you can dynamically create a string of proper size, then the second time to actually put characters in the string. Note that between two reads, you should close the file, then re-open the file, or alternatively, **rewind** the file. Again, don't forget to put the NULL character at the end. (You don't need to manually put the NULL character if you read the whole string in by any of the standard input functions.)

Your main program should then print a table of all fields and the corresponding values. Finally, your program will interactively ask the user to enter a field name, you then call cgi_val to find its value, and print the value or an error message if the field is invalid. Continue this process until the user decides to quit.

The **cgilib.h** header file and the data file, **Assignment5.data**, that you will be using are on the Kodiak. Another file, called **Assignment5.c**, which reads one character at a time from a text file is also available on Kodiak.

```
/*                    Assignment5.c                                  */

/*  This program will read from a text file, one character at a  */
/*  time, display each character, until the end of file is       */
/*  reached.                                                     */


#include <stdio.h>
#include <stdlib.h>    /*  It contains the function prototype of
                           "exit".  */



void main()
{
     int  c;
     char   file_name[21];
     FILE   *text;


     printf("\n\nEnter the file name -->  ");
     scanf("%s",file_name);
     text = fopen(file_name, "r");  /*  open the file for reading  */

     if (!text)
     {
          printf("\n\nError: The file \"%s\" does not exist.\n",
                                        file_name);
          exit(1);         /*  terminate the program with an error  */
     }


     while ((c = getc(text)) != EOF)   /* "getc" is used to read a
                                          character from a file.   */

          /*  Note that c was declared to be an int rather than a
              char.  For details, see C Example 6.  */

          putchar(c);

     fclose(text);
}
```