The University of Chicago
Department of Computer Science

CMSC 16200 – Honors Introduction to Computer Science 2
Winter Quarter 2007
Lab #6 (02/12/2007)
Due: 02/14 (Wed) at 5pm

Name:	
Student ID:	
Lab Instructor:	Borja Sotomayor

Do not write in this area

1	2	3	4	EC	**TOTAL**

Maximum possible points: 40 + 5

One of the exercises in this lab has to be done in a group of 4. In particular, you will be asked to do exercise 1 individually, and then provide feedback on the solutions other members of the group come up with.

Please include the names and student IDs of the students who reviewed your solution:

Name	***Student ID***

Using lex/yacc to parse configuration files

One of the (many) uses of `lex` and `yacc` is to parse configuration files. A configuration file is a text file containing a set of options for our program, laid out in a very specific format. This format is usually complex enough that using `sed`, `awk`, or plain C would involve way too much code. `lex`/`yacc` provide an easy way of interpreting that format and, furthermore, of including the parsing code in other C programs.

In this lab, we will use an ad hoc configuration file format similar to that used by the BIND DNS server. We will start by building a simple parser that accepts valid files and rejects files that are improperly formatted (for extra sanity, this simple parser will also print some of the file's content to standard output). Then, we will progressively add more features until we end up with a powerful configuration file parser.

Exercise 1 <<10 points>>

We wish to parse configuration files such as this one:

```
global {
        num_daemons = 4
        max_out_bandwidth = 10.0
        max_in_bandwidth = 5.5
        etc = /etc/ftpd
        hostkey = /etc/ftpd/hostkey.pem
};

host "ftp-1.foobar.com" {
        ftproot = /var/ftp/server1
        max_out_bandwidth = 20.7
};

host "ftp-2.foobar.com" {
        ftproot = /var/ftp/server2
        exclude = /var/ftp/server2/private
};
```

We are given the following informal description:

- This configuration file specifies options for an FTP server. The options can be global (affecting the entire server) or host-specific.
- The file must always contain a global section, even if it is empty. This section always comes first.
- The file can contain zero or several host sections. These sections, in turn, can be empty.
- Options are specified by an *option identifier*, followed by the equals sign, followed by the *option value*. Option identifiers are alphanumerical strings (the underscore character is also allowed) beginning with a letter. Option values can be integers,

real numbers, or strings. String option values can contain letters, numbers, the underscore _ character, the hyphen – character, the slash / character, and the dot . character, and must begin with a letter or a slash character.

- Host names (in the host sections) can contain the same characters as option values.
- The rest should be easy to infer from the configuration file.

You are provided with a set of files that your parser should accept, along with a set of files that your parser should *not* accept. Do not assume anything about the configuration file format: if there are any ambiguities, ask the instructor.

If your parser accepts the file, it should print the file's content to standard output like this:

```
GLOBAL CONFIG
-------------
num_daemons=4
max_out_bandwidth=10.0
max_in_bandwidth=5.5
etc=/etc/ftpd
hostcert=/etc/ftpd/hostkey.pem

HOST 'ftp-1.foobar.com' CONFIG
------------------
ftproot=/var/ftp/server1
max_out_bandwidth=20.7

HOST 'ftp-2.foobar.com' CONFIG
------------------
ftproot=/var/ftp/server2
exclude=/var/ftp/server2/private
```

If the parser rejects the file, you are allowed to print out the contents of the file up to the point when the error is detected (we will improve this in Exercise 3).

Note: This exercise is meant to be done using both yacc *and* lex. You must use lex to first tokenize the file (the informal description of the configuration file indirectly suggests what the different tokens should be: option identifier, option value, ...). Although you could also express the format of the file using *only* a yacc grammar, that would be overkill. Using lex to write the lexer (yylex()), makes things much easier.

Exercise 2 (Documentation) <<10 points>>

It is not common to comment `lex`/`yacc` code, as it tends to be self-explanatory. However, you should make sure to comment any non-obvious code (e.g. if you are using some clever trick). Besides that, you must provide feedback on your peers' solutions to exercise 1 *only*. When sending your code to your group, you should state what sample files are correctly recognized by your parser and which ones are not. Then, for each member of your group, answer the following questions (your answers to these questions should be concise and to the point):

- Is the lexer design correct? (e.g. should the lexer consider more tokens? less?)
- If the parser accepts a bad file, or rejects a good file, identify an error in the `yacc` file that is causing (or contributing) to the problem. Do *not* try to identify every single error in your classmate's file.

Finally, briefly describe (1-2 paragraphs) how the feedback you received from the group affected your own solution.

You must write up this documentation in a text file called `DOCUMENTATION.txt`, which must be included in your lab submission.

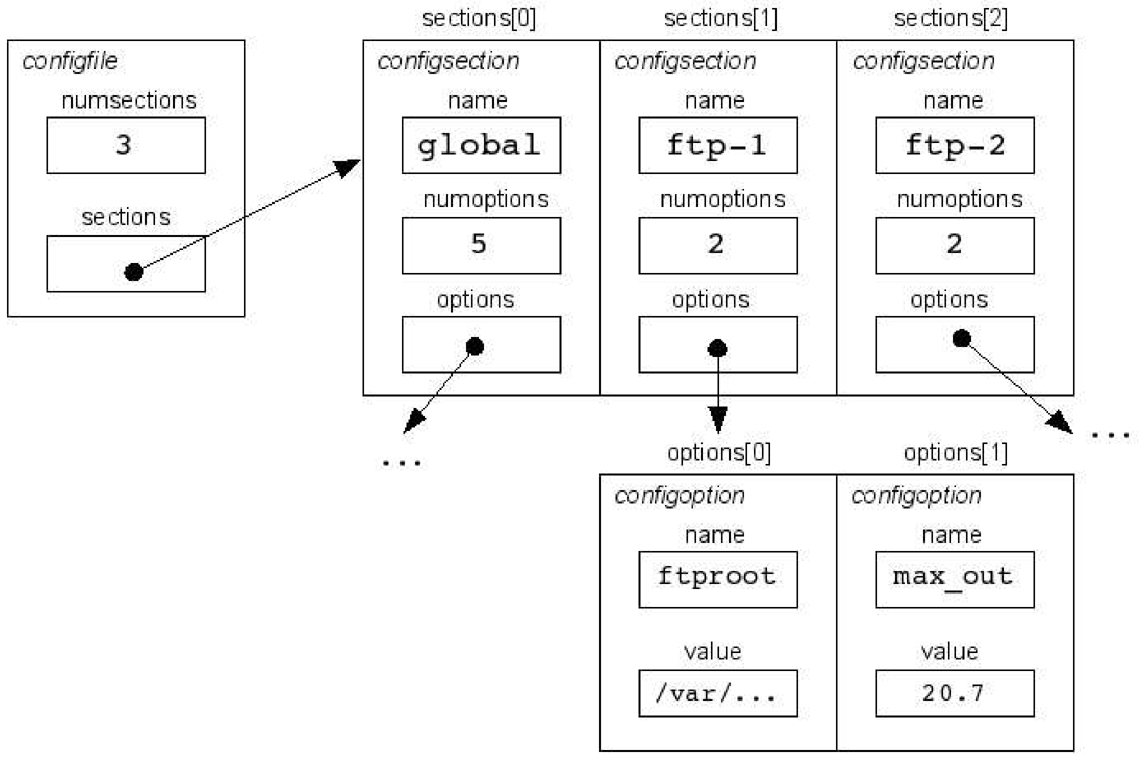
Exercise 3 <<10 points>>

In this exercise, we will make the information parsed from the file available to a C program. We will do this by loading the information into a set of data structures in memory, instead of simply printing out the information to standard output in each of the parser's actions.

These data structures have already been designed for you, and are the following:

```
struct configoption {
       char *name;
       char *value;
};

typedef struct configoption configoption;

struct configsection {
       char *name;
       unsigned int numopts;
       configoption *options;
};

typedef struct configsection configsection;

struct configfile {
       unsigned int numsections;
       configsection *sections;
};

typedef struct configfile configfile;
```

You are ***not*** allowed to modify the specification of these structures. To give you a better idea of how you are supposed to use them, the following diagram shows how a configuration file could be represented in memory:



Once the parsing is done (after `yyparse` has been called), the configuration file should be contained in a configfile variable. For example, you main program could look like this:

```
configfile *cf;

int main (void)
{
        cf = malloc(sizeof(configfile));
        // Initialize values in cf
        yyparse ();
        // Do something with variable cf
}
```

To test your program, you will use the `cf` variable to write out the values of the options. The output of your program should be something like this:

```
SECTION global
-------------
num_daemons=4
max_out_bandwidth=10.0
max_in_bandwidth=5.5
etc=/etc/ftpd
hostcert=/etc/ftpd/hostkey.pem

SECTION ftp-1.foobar.com
-------------
ftproot=/var/ftp/server1
max_out_bandwidth=20.7

SECTION ftp-2.foobar.com
-------------
ftproot=/var/ftp/server2
exclude=/var/ftp/server2/private
```

Note: To make the implementation simpler, we suggest you implement the following functions, and simply make calls to them from your parsing actions:

```
// Initialize a configfile structure (0 sections)
void init_file(configfile *cf);
// Initialize a configsection structure with a name (and 0 options)
void init_sec(configsection *sec, char *name);
// Add a section to a configfile
void add_sec(configfile *cf, configsection *sec);
// Initialize a configoption structure with a given name and value
void init_opt(configoption *opt, char *name, char *value);
// Add an option to a configsection
void add_opt(configsection *sec, configoption *opt);
// Print the contents of a configfile structure
void print_cf(configfile *cf);
```

Extra Credit <<5 points>>

Both the parser and the program treat all option values as strings (type `char*`). Modify your parser and the provided data structures so integers are represented internally as `int`'s, real numbers are represented as `double`'s, and strings are represented as `char*`'s.

Exercise 4 <<10 points>>

Modify the code from exercise 3 so your parser will be able to handle *variable substitutions*. For example, notice how the following configuration file has option values that refer to previous options:

```
global {
        num_daemons = 4
        max_out_bandwidth = 10.0
        max_in_bandwidth = 5.5
        etc = /etc/ftpd
        hostkey = %etc%/hostkey.pem
};

host "ftp-1.foobar.com" {
        ftproot = /var/ftp/server1
        max_out_bandwidth = 20.7
};

host "ftp-2.foobar.com" {
        ftproot = /var/ftp/server2
        exclude = %ftproot%/private
};
```

You must perform the variable substitution at the parser level, *not* at the application level. In other words, you can't take your code from Exercise 3 and simply go through the `configfile` structure in search of substrings that begin and end with %. You will need to modify your grammar to accommodate these references, and make sure that, when a new option is added to the `configfile` structure, its value has no variable references in it (i.e., you must perform all variable substitutions before storing the option value). For simplicity, we will assume that (1) you can only refer to variables within the section where the variable substitution takes place and (2) you can only refer to options that have already been parsed.

Directories, Makefiles, and executable file name

In this lab, you must include two directories:

- **`ex1`**: This directory will contain your implementation up to exercise 1.
- **`ex34`**: In exercises 3 and 4 (and the extra credit question), you will build on the implementation of exercise 1. This directory will contain your improved implementation. You must include a README file briefly describing how complete your implementation is (i.e., tell me if you implemented exercise 3, exercise 4, the extra credit problem, or some combination of these)

Each directory must contain a Makefile to compile the code in that directory (the Makefile must also handle all necessary calls to lex and yacc). The Makefile will have the default **`Makefile`** name (i.e., it should be possible to build your code by running **`make`** without specifying any extra parameters).

Your generated executable file *must* be called **`configparse`**

You must observe these naming conventions.

Failure to do so will result in a 30% point deduction.