A Compiler Project

The goals of this project is to design a simple but Turing Computable programming language and to implement a compiler for the language. For project 1, the compiler will scan, parse and report if the program is legal or not.

Policies:
This project will use the C programming language. It will be a recursive descent parser.

Click this link for the template parser

Your compiler should take input of the source code from a file.

At a minimum, you must implement the following language features:

- A main method
- Variable declarations (with a policy: do all declarations come first or can they be intermingled with executable statements)
- An iterator
- A selector
- An assignment statement
- Evaluation of nested arithmetic expressions
- A general solution with regard to comparison operators
- Provision of an integer data type
- Provision for numeric constants
- A syntactic element to place comments in the code

Project 1 requires a symbol table. The template version has a minimal, simple table with:

1. An abstract data type comprised of:

```
struct entry {

  char *lexptr;
  int token;
};
struct entry symtable[SYMMAX];
char lexemes[STRMAX];
```

2. three (very typical) functions:

```
void init();
int lookup(char s[]);
int insert(char s[], int tok)
```

Modify this symboltable to create an externally chained hash table for insertion and lookup. Modify struct entry as needed. Eliminate use of char lexemes[STRMAX]. The signatures of insert() and lookup() will remain the same, but will have different semantics. Specifically, they will now return success or failure (essentially booleans). You will add accessor functions to get the current token type or lexeme. Note that this change will have ripple effects in the lexer, parser, and emitter. `tokenval` will continue to hold the value of a numeric literal.

A word about identifiers:

A legal `<identifier>` will start with a letter followed by combinations of letters, numbers, and underscores, subject to the following two rules:

1. You cannot have consecutive underscores
2. An identifier cannot end with an underscore.

These are valid identifiers:

```
e123, e, qwert5yuio, a_b_7
```

These are not:

```
e__7, abc_
```

You are certainly invited but not required to extend the basic capabilities of your compiler. Here are some examples of what you might add:

- Additional data types such as float, char, etc
- Subprograms
- An indexing mechanism to handle arrays.
- Pointers and dynamic allocation (we are allowed to make some assumptions regarding memory management)

Input and Output
There will be no user interaction with the program. The program will be invoked 8 times from a shell script as demonstrated in class (see below in deliverables). Input will be via commandline parameter of the file to be parsed, specified in the script file. Output will be to the console reporting either a successful parse or a syntax error and the line on which the error was detected.

Other Policies
All projects will be individual.

Deliverables:

You will submit all your work bundled up in WinZip (on PC) to the DropBox for the course on `elearning.uwf.edu`. Please make your name is very clear on the zipfile you upload and inside all files contained within the zipfile. Place all your files in a single directory named `<firstInitial><lastname>-p<projNumber>`. For instance, my submission would be `jcoffey-p1`. zip that folder into a .zip file and upload it to the appropriate dropbox.

So, submit a single zip file containing a single folder, with the following items in the folder:

- The grammar for your language
- Source code for the lexical analyzer and parser portion of your program
- Design documents for this part of the project
- A User's manual regarding exactly what this submission does and how to run it
- Four examples each of both *legal* and *illegal* programs. The programs should be of increasing complexity, with the last demonstrating all features of your language.
- A script that compiles all legal and illegal programs as a batch, providing an indication that the compile was a success or appropriate error messages.
- A `makefile` to build your project
